

# Simplifying the analysis of software design variants with a colorful Alloy

Chong Liu, Nuno Macedo, and Alcino Cunha

INESC TEC and Universidade do Minho, Portugal

**Abstract.** Formal modeling and automatic analysis are essential to achieve a trustworthy software design prior to its implementation. Alloy and its Analyzer are a popular language and tool for this task. Frequently, rather than a single software artifact, the goal is to develop a full *software product line* (SPL) with many variants supporting different features. Ideally, software design languages and tools should provide support for analyzing all such variants (e.g., by helping pinpoint combinations of features that could break a property), but that is not currently the case. Even when developing a single artifact, support for multi-variant analysis is desirable to explore design alternatives. Several techniques have been proposed to simplify the implementation of SPLs. One such technique is to use background colors to identify the fragments of code associated with each feature. In this paper we propose to use that same technique for formal design, showing how to add support for features and background colors to Alloy and its Analyzer, thus easing the analysis of software design variants. Some illustrative examples and evaluation results are presented, showing the benefits and efficiency of the implemented technique.

**Keywords:** Formal software design · Variability · Alloy

## 1 Introduction

Formal methods are crucial in the development of high-assurance software. Their role in early development phases is well established, for example to check the consistency of formally specified requirements before proceeding to design, or that a formal model of the intended design satisfies desirable properties before proceeding to implementation. Among the myriad of formal methods proposed, *lightweight* ones – which rely on automatic analyses to verify (often partial) specifications – are quite popular. That is the case of model checkers like NuSMV [4] or SPIN [13], for verifying temporal logic properties of (behavioral) designs (modeled as transition systems), or model finders like Alloy [14], more geared towards verifying first-order properties of structural designs specified at a high level of abstraction (using simple mathematical concepts like sets and relations).

When developing large-scale software systems it is common to adopt the paradigm of *feature-oriented software development* [2], which organizes software around the key concept of *feature*, a unit of functionality that satisfies some of the requirements and that originates a configuration option. If the implementation is

properly decomposed, it is possible to deliver many variants of the system just by selecting the desired features. The set of all those variants is usually called a *software product line* (SPL). Ideally, the design of SPLs should already explicitly take features into account, and formal methods should be adapted to support such *feature-oriented design* [3]. In fact, even when developing a single software product, it is still convenient to explicitly consider features and multi-variant analysis during design to support the exploration of different design alternatives.

Most techniques to organize software implementation around features fall into one of two categories: *compositional* approaches implement features as distinct modules, and some sort of module *composition* technique is defined to generate a specific software variant; *annotative* approaches implement features with explicit (or sometimes implicit) annotations in the source code, that dictate which code fragments will be present in a specific variant. The former are well suited to support coarse-grained extensions, for example adding a complete new class to implement a particular feature, but not to support fine-grained extensions, for example adding a sentence to a method or change the expression in a conditional, to affect the way a code fragment works with different features [15]. Annotative approaches are much better suited for such fine-grained variability points.

Unfortunately, explicit support for feature-oriented design in formal methods, providing a uniform formalism for feature, architectural and behavioral modeling as advocated for SPL engineering [19], is still scarce. Support for features in model checking has been proposed, namely a compositional approach for the SMV modeling language of NuSMV [18] and an annotative approach for the Promela modeling language of SPIN [5]. For structural design, a compositional approach has been proposed to explicit support features in Alloy [3]. Typically, modeling and specifying in Alloy is done at high levels of abstraction, and adding a feature can require only minimal and very precise changes (e.g., adding one new relation to the model or changing part of the specification of a desired property). Compositional approaches such as [3] are not well suited for these fine-grained extensions. This paper addresses precisely this problem, proposing an annotative approach to add explicit support for features to Alloy and its Analyzer.

A classic annotative approach for source code is the use of `#ifdef` and `#endif` C/C++ compiler preprocessor directives to delimit code fragments that implement a specific feature. Unfortunately, such annotation style obfuscates the code and makes it hard to understand and maintain, leading to the well-known `#ifdef` hell [11]. To alleviate this problem, while retaining the advantages of annotative approaches, Kästner et al [15] proposed to annotate code fragments associated with different features with different background colors, which was later shown to clearly improve SPL code comprehension and be favored by developers [11]. Given these results, we propose to use such annotative technique to support features in Alloy. Our main contribution is thus a colorful extension to Alloy and its Analyzer, that allows users to annotate model and specification fragments with different background colors (denoting different features), and run analysis commands to verify either a particular variant, or several variants at once, of the design, simplifying the detection of feature combinations that may

```

1  fact FeatureModel {
2    // ④ requires ③
3    ④ ③ some none ③ ④ }
4
5  sig StoredModel {
6    ① derivationOf : lone StoredModel ①,
7    public       : lone Link,
8    ② secret      : lone Link ②,
9    ③ command     : lone Command ③ }
10
11  ② sig Secret in StoredModel {} ②
12
13  sig Link {}
14
15  ③ sig Command {} ③
16
17  ③ ④ sig Instance {
18    instanceOf : one Command,
19    model      : set StoredModel,
20    link       : one Link } ④ ③
21
22  fact {
23    // Links are not shared between artifacts
24    all l : Link | one (public+② secret ②+③ ④ link ④ ③).l
25    // All models have public links, unless commands are stored
26    ③ all m : StoredModel | one m.public ③
27    // The model derivations form a forest
28    ① no m : StoredModel | m in m.^derivationOf ①
29    // Private and public links must be different
30    ② all m : StoredModel | m.public != m.secret ②
31    ... }
32
33  run {some command} with ③ for 3
34  run {some command} with exactly ②, ③ for 3
35
36  assert OneDerivation {
37    // Stored models without a public link can have at most one derivation
38    ① all m : StoredModel | no m.public implies lone derivationOf.m ① }
39  check OneDerivation with ④ for 3

```

Fig. 1: Alloy4Fun specification in colorful Alloy.

fail to satisfy the desired specification. To the best of our knowledge, this is the first color-based annotative approach for feature support in a formal method.

The paper is organized as follows. The next section presents an overview of the proposed approach using a simple case study. Section 3 formally presents the new language extension, including the typing rules for the annotations, and Section 4 the extensions to the Alloy engine to support multi-variant analysis. Section 5 evaluates the flexibility of the language and the efficiency of the new multi-variant analysis engine using various examples. Section 6 discusses related work. Finally, Section 7 concludes the paper, presenting ideas for future work.

## 2 Overview

Alloy4Fun [17] is a web-platform<sup>1</sup> developed by our team for learning Alloy and sharing models. Besides the online creation, analysis and sharing of models,

<sup>1</sup> alloy4fun.inesctec.pt

Alloy4Fun has two additional goals: to provide a kind of auto-grading feature by marking certain parts of the model as secret, and to collect information regarding usage patterns and typical pitfalls when modeling in Alloy.

*Modeling.* The structure of an Alloy<sup>2</sup> model is defined by declaring *signatures* and *fields* (of arbitrary arity) within them, possibly with multiplicity constraints. A hierarchy can be imposed on signatures, either through extension or inclusion. These are combined into relational expressions using standard relational and transitive closure operators. Basic formulas over these expressions either compare them or perform simple multiplicity tests, and are combined using first-order logic operators. *Predicates* and *functions* represent re-usable formulas and expressions, respectively. Additional axioms are introduced through *facts*, and properties which are to be checked through *asserts*. In colorful Alloy, parts of the model can be associated with a positive annotation – selecting the feature introduces the element, colored background – or a negative annotation – selecting a feature removes the element, colored strike-through. Both positive and negative annotations can be nested, in which case the colors are mixed. Following the results of [11], we chose colors that would reduce visual fatigue.

Figure 1 depicts an excerpt of the encoding of the Alloy4Fun design variants in colorful Alloy. The base variant (i.e., the parts without annotations) simply stores models when shared by the user, and is thus comprised by stored models (**sig** `StoredModel`, l. 5) assigned to at most one public link (**sig** `Link`, l. 11, through field `public` with **lone** multiplicity, l. 7). Two additional constraints are enforced (by a **fact**, l. 22): a link is assigned to exactly **one** stored model (l. 24) and every stored model has a public link assigned to it (l. 26). To this base, 4 features can be added:

- ① to collect usage patterns, the derivation tree of the stored models is registered, introducing a new field (`derivationOf`, l. 6) and an additional constraint to avoid cyclic dependencies (using transitive closure, l. 28);
- ② stored models can have secrets defined (**sig** `Secret` as a subset of `StoredModel`, l. 11), and as a consequence, have private links generated when shared so that they can be recovered (field `secret`, l. 8); a new constraint forces public and private links to be distinct (l. 30) and the existing constraint on links is relaxed to allow links to be private (l. 24);
- ③ models are also stored when commands are executed (**sig** `Command`, l. 14), rather than just when shared by the user, allowing finer data collection; the command that originated such models is also stored (field `command`, l. 9); moreover, the constraint on the existence of public links is removed (l. 26), since stored models created through command execution are shared;
- ④ instances (**sig** `Instance`, l. 17) resulting from command execution can also be stored and shared; the constraint on links (l. 24) is relaxed, which may now point to stored instances (field `link`, l. 20) rather than just stored models.

Feature-oriented software development is usually accompanied by a *feature model* denoting which feature combinations are acceptable. In our example, instances

<sup>2</sup> For a thorough presentation of the Alloy language and the Analyzer, please see [14].

are associated to the commands that created them, so feature 4 requires feature 3. This is enforced by a fact (l. 1) that guarantees that invalid variants are unsatisfiable: if feature 4 without feature 3, an inconsistent formula is imposed (**some none**, since Alloy does not support Boolean constants). Feature models are easily translated into propositional logic [9], and are expected to be simple and easy to encode as this kind of facts at the level of abstraction that (colorful) Alloy is employed. Exploring whether dedicated support to encode feature models is needed is left for future work.

*Analysis.* Once the colorful Alloy model is defined, the user can instruct the Analyzer to generate scenarios for which a certain property holds through *run* commands, or instruct it to find counter-examples to a property expected to hold through *check* commands. These are assigned a scope, denoting the maximum size to be considered for the model’s signatures. The colorful Analyzer supports an additional scope on the variants that should be explored: a set of (positive and negative) feature presence conditions is provided, and analysis will either consider all variants or the smallest variant for which those conditions hold.

In our example, the run in l. 33 allows the user to explore scenarios with stored commands in all 8 variants where feature 3 is selected (without feature 3 command does not exist, so a type error would be thrown). This will generate an arbitrary instance for one of those variants; the user can then ask for succeeding solutions which may vary in variant or in scenario (more controlled enumeration is work in progress). A tighter scope provides a finer control on the explored variants, as the run in l. 34 for the 4 variants with features 2 and 3, which the Analyzer will report as unsatisfiable. This was due to a bug in the constraint on secret links (l. 30), which does not hold when stored models are created from command execution (a possible fix is to enforce instead **all m : StoredModel | no m.public&m.secret**). Such issue could go unnoticed without variant-focused analysis since it arises from the interaction of 2 features.

After exploring scenarios, the user can start checking desirable properties. The assert in l. 36 specifies a property that could be expected to hold for all variants that allow derivation trees: if a stored model has no public link, then at most one stored model is derived from it. Through the check in l. 39 for all variants with feature 1, the Analyzer quickly shows that that is not the case, generating a counter-example in a variant with features 1, 2 and 3, where command execution allows two stored models to derive the same ancestor. An analysis focused on individual variants could miss this possible issue arising from the interaction of features 1 and 3. Figure 2 presents an overview of the colorful Alloy Analyzer for this scenario, with its editor with feature annotations, and its visualizer with the counter-example and a panel denoting to which variant it belongs.

*Discussion.* Using regular Alloy, the user would have two alternative ways of encoding these design variants. One would be to try to encode in a single Alloy model all conditional structures and behaviors that the design may have, using a signature (e.g., **Variant**) to denote which features are under consideration. This

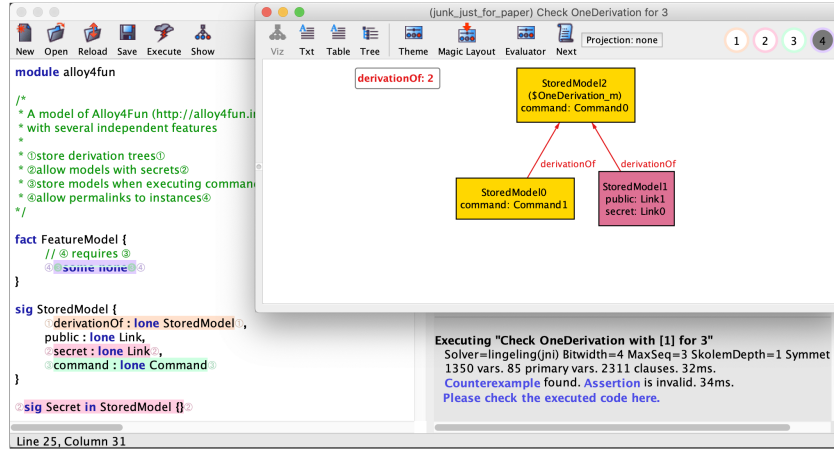


Fig. 2: Counter-example to OneDerivation.

quickly renders the Alloy model intractable, particularly in annotations of small granularity. For instance, the expression in l. 24 could be encoded like

```
public + (F2 in Variant implies secret else none → none) +
         (F3 + F4 in Variant implies link else none → none)
```

which can quickly become impenetrable and unmanageable.

The other alternative would be to rely on the Alloy module system and define variants as separate modules that import common elements. Modules are completely included when imported, meaning that existing elements cannot be removed or changed. As a consequence, a base module with the non-annotated parts cannot be extended with fields in existing signatures, requiring less intuitive workarounds. For instance, a module for feature 2 could add signature `Command` with the (arguably less intuitive) inverted field `secret` declared in it; for feature 1, adding field `derivationOf` would require the introduction of a “dummy” singleton signature with a field of type `StoredModel → StoredModel`. Negative annotations are even more troublesome, since they would have to be added by all modules except the ones that would remove them, which quickly becomes unmanageable. These issues are handled by the compositional approach implemented in `FeatureAlloy` [3], that allows the insertion of fields in existing signatures, as well as replacing existing paragraphs. However, handling variability points of fine granularity is still difficult to manage. For instance, the constraint in l. 24 not only would have to be fully replaced by features 2 and 4, but a new feature merging those two features would have to be created due to feature interaction.

### 3 Language

One of the reasons behind the initial proposal of color annotations was to avoid obfuscating the code with additional constructs [15]. There, colors are internally

handled by the IDE developed for that purpose, which hinders saving, sharing and editing models, particularly when dealing with simple, single-file, models as is typical in Alloy. Our approach aims at a middle ground, using minimal annotations that are colored when using the Analyzer, but that can still be saved and interpreted as a pure text file. Additionally, unlike [15], our language allows elements to be marked with the absence of features. Thus, although not allowing full propositional formulas, elements can be assigned a conjunction of positive/negative literals.

The colorful Alloy language is thus a minimal extension to regular Alloy mainly by allowing, first, elements to be associated with the presence or absence of features; and second, analysis commands to focus on particular sets of features. Features are identified by circled symbols,  $\textcircled{c}$  and  $\textcircled{\bar{c}}$ , denoting the presence and absence of a feature, respectively, for  $1 \leq c \leq 9$  (throughout the paper,  $\textcircled{c}$  will denote either  $\textcircled{c}$  or  $\textcircled{\bar{c}}$ ). This allows models with at most 9 distinct features, which we believe to be adequate since the colorful approach is known to be better suited for models with a small number of features [11], and our own experiments never relied on more than 6 features. Figure 3 presents the syntax of colorful Alloy, highlighting changes with regard to the regular Alloy language.

Features are associated to model elements by using feature marks as delimiters surrounding those elements. An element within a positive delimiter  $\textcircled{c}$  will only exist in variants where  $c$  is selected, while those within negative delimiters  $\textcircled{\bar{c}}$  only exist if  $c$  is absent from the variant. Color annotations can be nested, denoting the conjunction of presence conditions (e.g.,  $\textcircled{4}\textcircled{\bar{3}}\text{some none}\textcircled{\bar{3}}\textcircled{4}$  in Fig 1, l. 3, **some none** will be present in any variant with feature 4 selected but not feature 3). Likewise [15], only elements of the Alloy AST can be annotated. Thus, features cannot be assigned, for instance, to operators. Another consequence is that the model stripped down of its color marks is still a valid Alloy AST itself.<sup>3</sup> In general, any node whose removal does not invalidate the AST can be marked with features, including all global declarations (i.e., signatures, fields, facts, predicates, functions and asserts) and individual formulas within blocks. The marking of local declarations (i.e., predicate and function arguments, and quantified variables) is left as future work. One exception to the AST validity rule is allowed for binary expressions with a neutral element, in which case the sub-expressions can be annotated even if the whole binary expression is not. For instance,  $\text{public}+\textcircled{2}\text{link}\textcircled{2}$  is interpreted as  $\text{public}+(\text{none} \rightarrow \text{none})$  in variants where feature 2 is not selected.

Run and check commands can then be instructed to focus, possibly **exactly**, on certain features using a **with** scope: if not exact, commands will consider every variant where the scope features are present/absent; otherwise, exactly the variant with the presence features will be considered (negative features are spurious in that case). For instance, **run {} with**  $\textcircled{1},\textcircled{\bar{2}}$  will consider every variant with feature 1 selected but not feature 2, while **run {} with exactly**  $\textcircled{1},\textcircled{\bar{2}}$  will only consider the variant with exactly feature 1 selected. An additional feature

<sup>3</sup> We actually force the stripped model to be a valid Alloy model, forbidding, for instance, declarations with the same identifier associated with different feature sets.

```

spec ::= module qualName [ [ name,+ ] ] import* paragraph*
import ::= open qualName [ [ qualName,+ ] ] [ as name ]
paragraph ::= colPara | cmdDecl
colPara ::=  $\textcircled{c}$  colPara  $\textcircled{c}$  | 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 ::=  $\textcircled{c}$  colDecl  $\textcircled{c}$  | 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 (  $\Rightarrow$  | implies ) expr else expr
      | quant decl,+ blockOrBar | ( expr ) | block | { decl,+ blockOrBar }
colExpr ::=  $\textcircled{c}$  colExpr  $\textcircled{c}$  | expr
const ::= none | univ | iden
unOp ::= ! | not | no | mult | set |  $\sim$  | * | ^
binOp ::=  $\Leftrightarrow$  | iff |  $\Rightarrow$  | implies | - | ++ | <: | := | .
colBinOp ::= || | or |  $\&\&$  | and | + |  $\&$ 
arrowOp ::= [ mult | set ]  $\rightarrow$  [ 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 | [  $\otimes$  |  $\textcircled{c}$  ],+
qualName ::= [ this/ ] ( name/ )* name

```

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

mark  $\otimes$  denotes the empty variant (no features selected), and can be used to analyze every possible variant (if not exact, the default behavior if no color scope is provided) or solely the base variant (if exact).

The grammar of the language restricts which elements can be annotated, but additional type checking rules must be employed to guarantee consistent colorful models. These are formalized in Figs. 4 and 5 for a kernel of expressions and paragraphs, respectively, to which the remainder language features can be converted (except comprehension, omitted for simplicity). Expression arity is also considered, since this is the only rule enforced by the type system of regular Alloy. For a mark,  $\neg\textcircled{c}$  converts between the positive and negative version;  $\mathbf{c}$  denotes a set of marks; and  $[\mathbf{c}]$  expands  $\mathbf{c}$  with all marks  $\textcircled{c}$  such that  $\textcircled{c} \notin \mathbf{c}$ . The context of the type rules is a mapping  $\Gamma$  from variables to the color (and arity) of their declaration, and a set of marks  $\mathbf{c}$  under which it is being evaluated. We denote a singleton mapping as  $v \mapsto (\mathbf{c}, n)$  for a variable  $v$ , marks  $\mathbf{c}$  and arity  $n$ , while  $++$  denotes the overriding of mappings. An expression (or declaration)  $e$  of arity  $n$  is well-typed if  $\Gamma, \mathbf{c} \vdash_n e$  (arity 0 denotes formulas), and a paragraph  $p$  if



$$\begin{array}{c}
\frac{\square \in \{\mathbf{none}, \mathbf{univ}\}}{\Gamma, \mathbf{c} \vdash_1 \square} \quad \frac{}{\Gamma, \mathbf{c} \vdash_2 \mathbf{idem}} \quad \frac{v \mapsto (r, n) \in \Gamma \quad r \subseteq \mathbf{c}}{\Gamma, \mathbf{c} \vdash_n v} \\
\\
\frac{\Gamma, \mathbf{c} \vdash_n e \quad n > 0}{\Gamma, \mathbf{c} \vdash_n \wedge e} \quad \frac{\Gamma, \mathbf{c} \vdash_2 e}{\Gamma, \mathbf{c} \vdash_2 \sim e} \quad \frac{\Gamma, \mathbf{c} \vdash_0 e}{\Gamma, \mathbf{c} \vdash_0 \mathbf{not} e} \quad \frac{\Gamma, \mathbf{c} \vdash_0 e_1 \quad \Gamma, \mathbf{c} \vdash_0 e_2}{\Gamma, \mathbf{c} \vdash_0 e_1 \mathbf{and} e_2} \\
\\
\frac{\Gamma, \mathbf{c} \vdash_n e_1 \quad \Gamma, \mathbf{c} \vdash_n e_2 \quad n > 0}{\Gamma, \mathbf{c} \vdash_0 e_1 \mathbf{in} e_2} \quad \frac{\Gamma, \mathbf{c} \vdash_n e_1 \quad \Gamma, \mathbf{c} \vdash_n e_2 \quad n > 0 \quad \square \in \{\&, +, -\}}{\Gamma, \mathbf{c} \vdash_n e_1 \square e_2} \\
\\
\frac{\Gamma, \mathbf{c} \vdash_n e_1 \quad \Gamma, \mathbf{c} \vdash_m e_2 \quad k = n + m - 2 \quad n, m, k > 0}{\Gamma, \mathbf{c} \vdash_k e_1 \cdot e_2} \\
\\
\frac{\Gamma, \mathbf{c} \vdash_n e_1 \quad \Gamma, \mathbf{c} \vdash_m e_2 \quad k = n + m \quad n, m > 0}{\Gamma, \mathbf{c} \vdash_k e_1 \rightarrow e_2} \\
\\
\frac{\Gamma, \mathbf{c} \vdash_1 d \quad \Gamma, \mathbf{c} \dashv\vdash v \mapsto (\emptyset, 1) \vdash_0 e}{\Gamma, \mathbf{c} \vdash_0 \mathbf{all} v : d \mid e} \quad \frac{\Gamma, \mathbf{c} \vdash_n e \quad \neg \odot \notin \mathbf{c}}{\Gamma, \mathbf{c} \cup \{\odot\} \vdash_n \odot e \odot}
\end{array}$$

Fig. 4: Type rules for kernel expressions.

$$\begin{array}{c}
\frac{\Gamma, \mathbf{c} \vdash_{n_1} d_1 \quad \dots \quad \Gamma, \mathbf{c} \vdash_{n_i} d_i \quad \Gamma, \mathbf{c} \vdash_0 b \quad \Gamma, \mathbf{c} \vdash_1 h \quad n_1 \dots n_i > 0}{\Gamma, \mathbf{c} \vdash a \mathbf{sig} n \times h \{d_1, \dots, d_i\} b} \\
\\
\frac{\Gamma, \mathbf{c} \vdash_{n_1} d_1 \quad \dots \quad \Gamma, \mathbf{c} \vdash_{n_i} d_i \quad \Gamma, \mathbf{c} \vdash_n e \quad \Gamma, \mathbf{c} \vdash_n b \quad n, n_1 \dots n_i > 0}{\Gamma, \mathbf{c} \vdash \mathbf{fun} n [d_1, \dots, d_i] : e b} \\
\\
\frac{\Gamma, \mathbf{c} \vdash_{n_1} d_1 \quad \dots \quad \Gamma, \mathbf{c} \vdash_{n_i} d_i \quad \Gamma, \mathbf{c} \vdash_0 b \quad n_1 \dots n_i > 0}{\Gamma, \mathbf{c} \vdash \mathbf{pred} n [d_1, \dots, d_i] b} \quad \frac{\Gamma, \mathbf{c} \vdash_0 b}{\Gamma, \mathbf{c} \vdash \mathbf{fact} n b} \\
\\
\frac{\Gamma, \mathbf{c} \vdash_0 b \quad \square \in \{\mathbf{run}, \mathbf{check}\} \quad \vdash \mathbf{c}}{\Gamma, \emptyset \vdash \square n b \mathbf{with} \mathbf{c} s} \quad \frac{\Gamma, [\mathbf{c}] \vdash_0 b \quad \square \in \{\mathbf{run}, \mathbf{check}\} \quad \vdash \mathbf{c}}{\Gamma, \emptyset \vdash \square n b \mathbf{with} \mathbf{exactly} \mathbf{c} s} \\
\\
\frac{\Gamma, \mathbf{c} \vdash p \quad \neg \odot \notin \mathbf{c} \quad \vdash \mathbf{c} \quad \neg \odot \notin \mathbf{c}}{\Gamma, \mathbf{c} \cup \{\odot\} \vdash \odot p \odot} \quad \frac{\vdash \mathbf{c} \quad \neg \odot \notin \mathbf{c}}{\vdash \odot, \mathbf{c}} \quad \frac{\Gamma, \mathbf{c} \vdash_n e \quad n > 0}{\Gamma, \mathbf{c} \vdash_n v : e} \quad \frac{\Gamma, \mathbf{c} \vdash_n d \quad \neg \odot \notin \mathbf{c}}{\Gamma, \mathbf{c} \cup \{\odot\} \vdash_n \odot d \odot} \\
\\
\frac{\Gamma = \mathbf{decls}(\emptyset, p_1, \dots, p_n) \quad \Gamma, \emptyset \vdash p_1 \quad \dots \quad \Gamma, \emptyset \vdash p_i}{\vdash^m p_1 \dots p_i}
\end{array}$$

Fig. 5: Type rules for kernel paragraphs.

$\Gamma, \mathbf{c} \vdash p$ . A set of marks  $\mathbf{c}$  is valid,  $\vdash \mathbf{c}$ , if it does not contain contradictory marks. A complete colorful model  $m$  comprised by paragraphs  $p_1 \dots p_i$ , is well-typed if  $\vdash^m p_1 \dots p_i$ . This requires the prior collection of the declarations of global elements, as calculated by function  $\mathbf{decls}$ :<sup>4</sup>

<sup>4</sup> Function  $\mathbf{arity}$  is an oversimplification, since calculating the arity of a bounding expression requires the arity of other declared sigs and fields.

$$\begin{aligned}
\text{decls}(\mathbf{c}, d_1, \dots, d_n) &= \text{decl}(\mathbf{c}, d_1) \# \dots \# \text{decl}(\mathbf{c}, d_n) \\
\text{decl}(\mathbf{c}, \textcircled{\mathbf{c}} d \textcircled{\mathbf{c}}) &= \text{decl}(\mathbf{c} \cup \{\textcircled{\mathbf{c}}\}, d) \\
\text{decl}(\mathbf{c}, a \textit{ m } \mathbf{sig} \ n \ x \ \{ d \} \ b) &= n \mapsto (\mathbf{c}, 1) \# \text{decls}(\mathbf{c}, d) \\
\text{decl}(\mathbf{c}, v : e) &= v \mapsto (\mathbf{c}, \text{arity}(e)) \\
\text{decl}(\mathbf{c}, \mathbf{pred} \ n \ [ \ d \ ] \ b) &= n \mapsto (\mathbf{c}, \#d) \\
\text{decl}(\mathbf{c}, \mathbf{fun} \ n \ [ \ d \ ] : e \ b) &= n \mapsto (\mathbf{c}, \#d + \text{arity}(e)) \\
\text{decl}(\mathbf{c}, \mathbf{run} \ n \ b \ w \ f) &= \emptyset \\
\text{decl}(\mathbf{c}, \mathbf{check} \ n \ b \ w \ f) &= \emptyset
\end{aligned}$$

Type rules check mainly for two kinds of coloring issues. First, calls to global elements must occur in a context that guarantees its existence. For instance, signature `Secret`, declared with feature 1, may not be called in a plain `fact { some Secret }` since that fact will be present in variants where secrets do not exist (those without feature 1). This applies to calls in expressions, the class hierarchy (the parent signature must exist in every variant that the children do), and calls to predicates/asserts in `run/check` commands. Commands are not annotated, their context being instead defined by the color scope (for exact color scopes, the context is expanded with the negation of all marks not present in it,  $[\mathbf{c}]$ ). Second, the nesting of negative and positive annotations of the same feature is forbidden, since this conjunction of conditions is necessarily inconsistent (i.e., under  $\textcircled{1}\textcircled{1}e\textcircled{1}\textcircled{1}$ ,  $e$  will never exist). This also applies to color scopes of commands, where the presence and absence of a feature would allow no variant.

## 4 Analysis

Analysis of colorful Alloy models is achieved through the translation into regular Alloy. There are two main alternative ways to do this: *i*) through the generation and analysis, for every feature combination, of a projected version of the model; *ii*) through the generation of an ‘amalgamated’ Alloy model that encompasses the alternative behaviors of the model family. In order to compare their performance (see Section 5), we have implemented both translations in our prototype. Since this preliminary evaluation is inconclusive, the current version of the colorful Analyzer relies on the amalgamated translation for non-exact color scopes, and on the projection one otherwise (i.e., for the analysis of a single variant). The resulting models are also provided to the user, which can be used to better understand the model family under development (particularly, the projected versions allow the user to inspect concrete variants).

Figures 7 and 8 present the translation of the colorful model into the amalgamated version for paragraphs and expressions, respectively. It is assumed that the colorful model is well-typed at this stage, and that all unique colorful marks  $\mathbf{c}_0$  that occur in it have been collected during that process (i.e., the features relevant for this family of models). Expressions  $\mathbf{c}^+$  and  $\mathbf{c}^-$  filter the positive

```

abstract sig Feature {}
one sig F1,F2,F3,F4 extends Feature{}
sig Variant in Feature {}

fact FeatureModel {
  (F3 not in Variant and F4 in Variant) implies
  some none }

sig StoredModel {
  derivationOf : set StoredModel,
  ...,
  command      : set Command }
sig Link, Command {}

fact {
  F3 in Variant implies
  command in StoredModel  $\rightarrow$  lone Command else no command
  F3 not in Variant implies no Command
  ... }

fact Links {
  all l : Link | one (public+
  (F2 in Variant implies secret else none  $\rightarrow$  none)+
  (F3+F4 in Variant implies link else none  $\rightarrow$  none)).l
  F3 not in Variant implies all m : StoredModel | one m.public
  ... }

```

```

sig StoredModel{
  public: lone Link,
  secret: lone Link,
  command: lone Command }

sig Secret in StoredModel{}

sig Link{}

sig Command{}

fact Links {
  all l : Link |
  one (public+secret).l
  all m : StoredModel |
  m.public != m.secret
  ... }

```

(b) ②,③ projection.

(a) Amalgamated translation.

Fig. 6: Excerpts of the translations for the Alloy4Fun colorful model.

or negative color marks from a set, respectively. For a model  $m \rho_1 \dots \rho_i$ , the translation  $\llbracket m \rho_1 \dots \rho_i \rrbracket_{c_0}$  starts by introducing an abstract signature `Feature`, that is extended exactly by singleton signatures that represent each of the relevant features. Signature `Variant`, a sub-set of `Feature`, represents particular feature combinations under consideration.<sup>5</sup> Its acceptable valuations are restricted by facts introduced during the translation of the color scope of the commands. To control the existence of structural elements (signatures and fields), their multiplicity is relaxed and additional facts only enforce them if the associated features are present/absent. Even though these elements are always declared, the colorful type checking rules guarantee that they are not referenced in invalid variants. In the kernel language, only sub-expressions of binary operators may be associated with features (blocks of formulas have been converted into binary conjunctions). Depending on the presence/absence of the relevant features, either the expression or its neutral element is returned. Commands are also expanded depending on their color scope, so that only relevant variants are considered. Figure 6a presents an excerpt of the amalgamated Alloy model for the Alloy4Fun colorful model.

The projection translation is straight-forward: given a concrete variant, it projects away elements not relevant in that variant. Paragraphs not associated with a particular variant are completely removed, as are branches of marked binary expressions. Since colorful Alloy does not natively support feature models,

<sup>5</sup> To avoid collisions with the identifiers of the colorful model, the translation actually uses obfuscated identifiers these signatures.

```


$$\llbracket m \ p_1 \ \dots \ p_i \rrbracket_{k\dots t} \equiv \begin{array}{l} \text{abstract sig Feature } \{\} \\ \text{one sig } Fk, \dots, Fl \text{ extends Feature } \{\} \\ \text{sig Variant in Feature } \{\} \\ \llbracket p_1 \rrbracket_{\emptyset} \ \dots \ \llbracket p_i \rrbracket_{\emptyset} \end{array}$$


$$\llbracket a \ m \ \text{sig } n \times \{ d_1, \dots, d_i \} \ b \rrbracket_c \equiv \begin{array}{l} a \ \text{sig } n \times \{ \llbracket d_1 \rrbracket_c, \dots, \llbracket d_i \rrbracket_c \} \ \llbracket b \rrbracket_c \\ \text{fact } \{ (\llbracket c^+ \rrbracket \text{ in Variant and } \llbracket c^- \rrbracket \text{ not in Variant}) \text{ implies } m \ n \\ \quad \text{else no } n \} \\ \text{fact } \{ \text{trans}(d_1) \ \dots \ \text{trans}(d_i) \} \end{array}$$

where

$$\text{trans}(\odot \ d \ \ominus) = \text{trans}(d)$$


$$\text{trans}(v : e) = (\llbracket c^+ \rrbracket \text{ in Variant and } \llbracket c^- \rrbracket \text{ not in Variant}) \text{ implies } v \text{ in } n \rightarrow e \\ \text{else no } v$$


$$\llbracket \text{pred } n \ [ \ d \ ] \ b \rrbracket_c \equiv \text{pred } n \ [ \ \llbracket d \rrbracket_c \ ] \ \llbracket b \rrbracket_c$$


$$\llbracket \text{fun } n \ [ \ d \ ] : e \ b \rrbracket_c \equiv \text{fun } n \ [ \ \llbracket d \rrbracket_c \ ] : \llbracket e \rrbracket_c \ \llbracket b \rrbracket_c$$


$$\llbracket \text{fact } n \ b \rrbracket_c \equiv \text{fact } n \ \llbracket b \rrbracket_c$$


$$\llbracket \text{run } n \ b \ \text{with } c \ s \rrbracket_{\emptyset} \equiv \text{run } n \ \{ (\llbracket c^+ \rrbracket \text{ in Variant and } \llbracket c^- \rrbracket \text{ not in Variant}) \text{ and } \llbracket b \rrbracket_c \} \ s$$


$$\llbracket \text{run } n \ b \ \text{with exactly } c \ s \rrbracket_{\emptyset} \equiv \text{run } n \ \{ \llbracket c^+ \rrbracket = \text{Variant and } \llbracket b \rrbracket_c \} \ s$$


$$\llbracket \text{check } n \ b \ \text{with } c \ s \rrbracket_{\emptyset} \equiv \text{check } n \ \{ (\llbracket c^+ \rrbracket \text{ in Variant and } \llbracket c^- \rrbracket \text{ not in Variant}) \text{ implies } \llbracket b \rrbracket_c \} \ s$$


$$\llbracket \text{check } n \ b \ \text{with exactly } c \ s \rrbracket_{\emptyset} \equiv \text{check } n \ \{ \llbracket c^+ \rrbracket = \text{Variant implies } \llbracket b \rrbracket_c \} \ s$$


$$\llbracket \odot \ p \ \ominus \rrbracket_c \equiv \llbracket p \rrbracket_{c \cup \{c\}}$$


$$\llbracket v : m \ e \rrbracket_c \equiv v : \text{set } \llbracket e \rrbracket_c$$


$$\llbracket v : e_1 \ m_1 \rightarrow m_2 \ e_2 \rrbracket_c \equiv v : \llbracket e_1 \rrbracket_c \ \text{set} \rightarrow \text{set } \llbracket e_2 \rrbracket_c$$


$$\llbracket \textcircled{i}, \dots, \textcircled{j} \rrbracket \equiv \llbracket \textcircled{i} \rrbracket, \dots, \llbracket \textcircled{j} \rrbracket$$


$$\llbracket \textcircled{c} \rrbracket \equiv Fc$$


$$\llbracket \textcircled{\times} \rrbracket \equiv \text{none}$$


```

Fig. 7: Paragraph translation into the amalgamated model with variability.

the  $2^{\#c}$  projected models must be generated and analyzed (although the process can be stopped once one of those models is found to be satisfiable). However, the codification of feature models proposed in Section 2 actually renders invalid variants trivially unsatisfiable and instantaneously discharged: the projection of the model for such variants will end up with a fact enforcing **some none**, which is detected during the translation into SAT before the solving process is even launched. Figure 6b presents an excerpt of a projected Alloy model for the Alloy4Fun colorful model under an exact scope  $\textcircled{2}, \textcircled{3}$ .

## 5 Evaluation

Our evaluation aimed to answer two questions regarding the feasibility of the approach, prior to developing more advanced analysis procedures: *i*) is the analysis

$$\begin{aligned}
\llbracket k \rrbracket_c &\equiv k \\
\llbracket n \rrbracket_c &\equiv n \\
\llbracket \square e \rrbracket_c &\equiv \square \llbracket e \rrbracket_c \\
\llbracket e_1 \square e_2 \rrbracket_c &\equiv \begin{cases} \llbracket e_1 \rrbracket_c \square \llbracket e_2 \rrbracket_c & \text{if } \square \notin \{+, \&, \text{or}, \text{and}\} \\ \text{trans}(e_1) \square \text{trans}(e_2) & \text{otherwise} \end{cases} \\
\text{where} & \\
\text{trans}(e) &\equiv \llbracket c^+ \rrbracket \text{ in Variant and } \llbracket c^- \rrbracket \text{ not in Variant implies } \llbracket e \rrbracket_c \text{ else neutral}(\square, \text{arity}(e)) \\
\text{neutral}(+, a) &= \underbrace{\text{none} \rightarrow \dots \rightarrow \text{none}}_a \\
\text{neutral}(\&, a) &= \underbrace{\text{univ} \rightarrow \dots \rightarrow \text{univ}}_a \\
\text{neutral}(\text{or}, a) &= \text{some none} \\
\text{neutral}(\text{and}, a) &= \text{no none} \\
\llbracket \square d \mid e \rrbracket_c &\equiv \square \llbracket d \rrbracket_c \mid \llbracket e \rrbracket_c \\
\llbracket \odot e \odot \rrbracket_c &\equiv \llbracket e \rrbracket_{c \cup \{\odot\}}, \\
\llbracket v : e \rrbracket_c &\equiv v : \llbracket e \rrbracket_c
\end{aligned}$$

Fig. 8: Expression translation into the amalgamated model with variability.

through the amalgamated model feasible? And if so, *ii*) does it outperform a preprocessing approach that iteratively analyzes all projected variants? To answer these questions, we applied our technique to 7 model families with different characteristics, including some rich on structural and others on behavioral properties, and mostly encoding variants of system design. This also allowed us to validate the expressibility and flexibility of the language extension.

### 5.1 Evaluation subjects

The *OwnGrandpa* model is based on 2 toy models by Daniel Jackson distributed with the Alloy Analyzer that share certain elements, one modeling genealogical relationships and other solving the “I’m My Own Grandpa” puzzle. In [14], the latter is presented in stages to address different concepts, which are distributed as 3 distinct Alloy files. Our base variant considers basic biological facts, which can be extended by 1) introducing Adam and Eve, who are considered as the first man and woman according to the Bible creation myth; 2) introducing social norms regarding marriage; and 3) forbidding incestuous marriages. The feature model forces feature 3 to require 2. The command evaluated checks whether all persons descend from Adam and Eve in variants with feature 1.

The *E-commerce* platform model is adapted from [8] and models variants for the catalog structure of the platform. In the base variant the catalog is a collection of items, which can be enhanced by 1) allowing items to be classified in categories; 2) allowing a hierarchy on categories; 3) allowing the assignment of multiple categories to items; 4) presenting images of items; and 5) presenting thumbnails summarizing categories. The feature model forces features 2, 3 and 5

Table 1: Evaluation of the amalgamated and iterative approaches for the examples.

Model	NF	Command	NP	NV	Scope	TA(s)	TI(s)	SU	SP(s)	
<i>OwnGrandpa</i>	3	AllDescend	4	4		9	0.3	1.7	5.67	0.9
						10	1.0	10.9	10.90	4.0
						11	7.3	24.1	3.30	13.4
						12	26.1	132.6	5.08	57.1
<i>E-commerce</i>	5	AllCataloged	32	12		10	6.2	13.6	2.19	3.5
						11	15.5	57.1	3.68	17.4
						12	73.7	182.0	2.47	45.2
<i>Graph</i>	6	Connected	32	6		8	3.2	19.2	6.00	3.4
						9	11.9	80.9	6.80	16.7
		SourcesAndSinks	32	10		8	7.0	62.1	6.99	12.9
					9	187.5	1010.2	5.39	166.0	
<i>Alloy4Fun</i>	4	NoCommands	4	4		25	1.8	8.1	4.50	3.1
						30	4.4	17.5	3.98	7.7
		PublicSecretDisjoint	8	6		20	1.1	6.3	5.73	1.9
						25	2.8	19.6	7.00	7.6
					30	5.9	37.9	6.42	11.0	
<i>Vending</i>	2	Stock	4	4		6 but 4 Int	4.6	8.5	1.85	5.0
						8 but 4 Int	5.5	9.7	1.76	4.4
						5 but 5 Int	<b>30.7</b>	<b>26.1</b>	<b>0.85</b>	14.8
						7 but 5 Int	19.3	28.6	1.48	13.1
		Selection	4	4		6 but 4 Int	2.1	2.8	1.33	1.1
						8 but 4 Int	2.4	3.7	1.54	1.7
						5 but 5 Int	4.5	5.7	1.27	2.7
					7 but 5 Int	4.0	9.0	2.25	4.5	
<i>Bestiary</i>	4	Injective	8	8		25	6.9	12.8	1.86	3.0
						30	9.8	49.6	5.05	16.0
		Functional	8	8		25	2.4	11.1	4.59	2.4
						30	10.2	33.6	3.29	8.4
						6	2.8	9.4	3.38	2.5
Associative	8	8		7	52.5	211.9	4.04	62.2		
				8	230.2	891.9	3.88	309.1		

to require feature 1; feature 5 also requires feature 4. The command evaluated tests whether all items are cataloged in every variant.

*Graph* is adapted from a compositional version from [1] that explores different classes of graphs. The base simply defines nodes and edges, which can be extended by forcing the graph to be: 1) a multigraph; 2) undirected; 3) a directed acyclic graph; 4) a tree; 5) edge labeled; and 6) a binary search tree. The feature model declares feature 2 as incompatible with 3, feature 4 requiring 3, and 6 requiring both 4 and 5. The evaluated properties are whether the graph is connected and whether non-empty graphs have at least one source and one sink node.

*Alloy4Fun* has already been thoroughly explored in Section 2. The evaluated commands check whether it is possible to create commands, and whether public

and private links are always disjoint. *Vending* is inspired by various vending machine examples commonly used in SPL literature (e.g., [10]). The base variant of this dynamic example encodes the process of selecting and serving an item, extensible by introducing two independent features, 1) the notion of price and payment; and 2) the possibility to select multiple items. The first command evaluated tests whether the stock is always non-negative, and the second whether only elements with positive stock can be selected (all commands assume scope 15 on *Time*). Finally, *Bestiary* is a family of very simple models that we use in classes to explore different types of relations. Each feature defines relations as 1) injective; 2) functional; 3) total; and 4) surjective. Commands test alternative definitions of injectivity and functionality, as well as whether relations are associative.

## 5.2 Results

Table 1 depicts execution times for the examples presented above, for varying scope. The table presents how many features each model has (NF). Then, for each pair command/scope of a model, it presents how many variants are considered by the color scope (NP), how many of those variants are valid according to the feature model (NV), the analysis time under the amalgamated model (TA), under the iterative analysis of all projected variants (TI), and the speedup of the former in relation to the latter (SU). The slowest time for a projected variant (SP) is also presented. All commands were run 50 times on a MacBook with a 2.4 GHz Intel Core i5 and 8GB memory using the MiniSAT solver.

Results show that the amalgamated approach is indeed feasible, since it proves to be always faster than the iterative analysis except for one particular command of *Alloy4Fun* (highlighted as bold). The evaluation did however raise an interesting unexpected question, due to how frequent the analysis of a single projected variant is slower than the full amalgamated analysis. For *OwnGrandpa* we identified the cause as being related to imposing signature multiplicities through the declaration rather than through a fact. Why this affects the underlying procedure, and whether it can be explored to improve performance, is left as future work.

## 6 Related work

Several approaches have been proposed for feature-oriented design. We focus on those that provide specification languages supported by automated analyses. *fSMV* [18] is a compositional approach for SMV, where a base system in pure SMV can be extended with features, modeled in new textual units. The base behavior may be overwritten by features, integrated automatically by compilation into pure SMV so that normal SMV model checkers can be employed. *fPromela* [5] provides instead an annotative approach for SPIN’s modeling language, where features are introduced by a new user-defined data type structure. Features of a model must be declared as a field of this structure and can be referenced elsewhere by declaring a variable of this type. *fPromela* is accompanied by a language for the specification of feature models, TVL. *FeatureAlloy* [3] introduces a compositional

approach for Alloy. Like fSMV, features are encapsulated and modeled separately from the base system, which are then combined by an external tool in order to produce a final model. This is achieved by recursively superimposing and merging selected features; new fields can be added to signatures, while facts, predicates, or function are overridden, which is unsuited for fine-grained variability points.

The idea to compare some kind of “amalgamated” model checking with the iterative analysis of all variants has been explored in [6], where the analysis of fSMV models through a symbolic algorithm for featured transition systems (FTS) was compared with that of the enumeration of regular SMV models and analysis with NuSMV. Evaluation showed that the former often outperformed the enumerative approach. A similar study was performed for fPromela through a semi-symbolic algorithm for FTS [5], which again proved to be more efficient in general than the enumerative approach using SPIN.

As previously mentioned, background colors have been proposed as an annotative approach for feature-oriented software development [15]. These are similar to `#ifdef` statements, with code fragments only being included when the associated features are selected. Similarly to our approach, it offers a direct mechanism for developers to find whether a code fragment is associated with a feature, but in our approach annotations are part of the model itself, instead being handled by the supporting tool. Extensions to this work [11] also support negative annotations, but color them similarly to positive ones. Colors have also been used to highlight feature annotations in graphical editors for models with variability [7,12].

Several approaches have used Alloy to formalize and analyze feature models [20], but not models with variability points. These could complement colorful Alloy, if dedicated support for feature models is deemed useful in the future.

Techniques have been proposed for type-checking SPLs. One such technique is proposed in [16] but is tailored for Java programs. [8] proposes checking the well-formedness of a model template against a feature model by mapping OCL well-formedness rules into propositional formulas verified by a SAT solver.

## 7 Conclusion and future work

This work explores an annotative approach to feature-based design that minimally extends the Alloy language, and its Analyzer, through colorful annotations and variant-focused analysis commands. Two alternative analysis approaches have been explored to execute these commands. A preliminary study has been performed, showing that in general the amalgamated analysis of the model fares better than the enumeration and subsequent analysis of all projected variants.

Future work is planned on several axes. Regarding the language, we intend to expand the support to additional operators, as well as explore whether syntactic features for specifying feature models are needed. Regarding the analysis processes, we plan to continue exploring the relation between the amalgamated and the iterative approach, and whether there is some middle ground that could provide optimal results. We also expect to implement new analysis operations, like run commands that check all variants for consistency.



## Acknowledgments

This work is financed by the ERDF - European Regional Development Fund - through the Operational Programme for Competitiveness and Internationalisation - COMPETE 2020 - and by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within project POCI-01-0145-FEDER-016826. The third author was also supported by the FCT sabbatical grant with reference SFRH/BSAB/143106/2018.

## References

1. Apel, S., Kästner, C., Lengauer, C.: Language-independent and automated software composition: The FeatureHouse experience. *IEEE Transactions on Software Engineering* **39**(1), 63–79 (2013)
2. Apel, S., Kästner, C.: An overview of feature-oriented software development. *Journal of Object Technology* **8**(5), 49–84 (2009)
3. Apel, S., Scholz, W., Lengauer, C., Kästner, C.: Detecting dependences and interactions in feature-oriented design. In: *Proceedings of the IEEE 21st International Symposium on Software Reliability Engineering (ISSRE)*. pp. 151–170. IEEE (2010)
4. Cimatti, A., Clarke, E., Giunchiglia, F., Roveri, M.: NuSMV: A new symbolic model checker. *International Journal on Software Tools for Technology Transfer* **2**(4), 410–425 (2000)
5. Classen, A., Cordy, M., Heymans, P., Legay, A., Schobbens, P.Y.: Model checking software product lines with SNIP. *Software Tools for Technology Transfer* **14**(5), 589–612 (2012)
6. Classen, A., Heymans, P., Schobbens, P.Y., Legay, A., Raskin, J.F.: Model checking lots of systems: Efficient verification of temporal properties in software product lines. In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE)*. pp. 335–344. ACM (2010)
7. Czarnecki, K., Antkiewicz, M.: Mapping features to models: A template approach based on superimposed variants. In: *Proceedings of the 4th International Conference on Generative Programming and Component Engineering (GPCE)*. LNCS, vol. 3676, pp. 422–437. Springer (2005)
8. Czarnecki, K., Pietroszek, K.: Verifying feature-based model templates against well-formedness OCL constraints. In: *Proceedings of the 5th International Conference on Generative Programming and Component Engineering (GPCE)*. pp. 211–220. ACM (2006)
9. Czarnecki, K., Wasowski, A.: Feature diagrams and logics: There and back again. In: *Proceedings of the 11th International Conference Software Product Lines (SPLC)*. pp. 23–34. IEEE (2007)
10. Fantechi, A., Gnesi, S.: Formal modeling for product families engineering. In: *Proceedings of the 12th International Conference on Software Product Lines (SPLC)*. pp. 193–202. IEEE (2008)
11. Feigenspan, J., Kästner, C., Apel, S., Liebig, J., Schulze, M., Dachselt, R., Pappendieck, M., Leich, T., Saake, G.: Do background colors improve program comprehension in the #ifdef hell? *Empirical Software Engineering* **18**(4), 699–745 (2013)

12. Heidenreich, F., Kopcsek, J., Wende, C.: FeatureMapper: Mapping features to models. In: Companion Volume of the 30th International Conference on Software Engineering (ICSE Companion). pp. 943–944. ACM (2008)
13. Holzmann, G.J.: The model checker SPIN. *IEEE Transactions on Software Engineering* **23**(5), 279–295 (1997)
14. Jackson, D.: *Software Abstractions – Logic, Language, and Analysis*. MIT Press, revised edn. (2012)
15. Kästner, C., Apel, S., Kuhlemann, M.: Granularity in software product lines. In: Proceedings of the 30th International Conference on Software Engineering (ICSE). pp. 311–320. ACM (2008)
16. Kästner, C., Apel, S., Thüm, T., Saake, G.: Type checking annotation-based product lines. *ACM Transactions on Software Engineering and Methodology* **21**(3), 14:1–14:39 (2012)
17. Macedo, N., Cunha, A., Pereira, J., Carvalho, R., Silva, R., Paiva, A.C.R., Ramalho, M.S., Silva, D.C.: Sharing and learning Alloy on the web. *CoRR* **abs/1907.02275** (2019)
18. Plath, M., Ryan, M.: Feature integration using a feature construct. *Science of Computer Programming* **41**(1), 53–84 (2001)
19. Schaefer, I., Hähnle, R.: Formal methods in software product line engineering. *IEEE Computer* **44**(2), 82–85 (2011)
20. Sree-Kumar, A., Planas, E., Clarisó, R.: Analysis of feature models using Alloy: A survey. In: Proceedings of the 7th International Workshop on Formal Methods and Analysis in Software Product Line Engineering (FMSPLE@ETAPS). EPTCS, vol. 206, pp. 46–60 (2016)