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, instead of a single software artifact, the goal is to develop a full *software product line* (SPL) with many variants that support different features. Ideally, software design languages and tools should provide support for analyzing all such variants, for example helping pinpoint combinations of features that could break a desired property, but that is not the case nowadays. Even when developing a single artifact, support for multi-variant analysis is desirable when exploring design alternatives. Several techniques have been proposed to simplify the programming of SPLs. One such technique is to use background colors to identify the fragments of the code that implement each feature. In this paper we propose to use that very same technique for formal design, namely show how to add support for features and background colors to Alloy and its Analyzer, to ease the analysis of software design variants. Some illustrative examples and preliminary evaluation results are presented, showing the benefits and efficiency of the implemented technique.

Keywords: Formal software design · Variability · Alloy

1 Introduction

Formal methods are essential in the development of high-assurance software. Their role in the early software development phase is well established, for example to check the consistency of a set of formally specified requirements before proceeding to design, or that a formal model of the intended design satisfies some desirable properties before proceeding to implementation. Among the myriad of formal methods, lightweight ones that rely on automatic analysis tools to verify (sometimes partial) specifications are quite popular. That is the case of model checkers like NuSMV [4] or SPIN [10], for verifying temporal logic properties of (behavioral) software designs modeled using transition systems, or model finders like Alloy [11], more geared towards verifying first order properties of structural designs specified at a high level of abstraction (using very 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]. The key idea is to organize the software around the key concept of *feature*, a unit of functionality that satisfies some of the requirements and that originates a potential 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, when developing SPLs, software design should already explicitly take features into account, and formal methods should be adapted to support such *feature-oriented design* [3]. Even if the goal is to develop a single software product, at the design phase it is still very convenient to explicitly consider features and multi-variant analysis to support the exploration of different design alternatives.

There are plenty techniques to organize the software implementation around features. Most 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. Compositional approaches are well suited to support coarse-grained extensions, for example adding a complete new class to implement a particular feature, but are not so good 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 [12]. Annotative approaches are much better suited for such fine-grained extension.

Unfortunately, explicit support for feature-oriented design in formal methods is still quite rare. Some support for features in model checking has been proposed, namely a compositional approach for the SMV modeling language of NuSMV [15] 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]. Being compositional, it is not well suited for fine-grained extension, and frequently when modeling and specifying at a high-level of abstraction, as is typically the case with Alloy, adding support for a feature can require only minimal and very precise changes, for example adding just one new relation to the model or change part of the specification of a desired property. In this paper we address precisely this problem, and propose an annotative approach to add explicit support for features to Alloy and its Analyzer.

The most classic example of 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 style of annotations causes problems, namely it obfuscates the code, making it hard to understand and difficult to maintain, leading to the well-known `#ifdef` hell [9]. To alleviate this problem, but still retain the advantages of annotative approaches, Kästner et al [12] proposed to annotate code fragments associated with different features with different background colors, a technique that was later showed to clearly improve SPL code comprehension and be favored by developers [9]. Given these results, we propose to use this very same annotative technique to support features in Alloy, to ease the analysis of software design variants. Our main contribution is thus a colorful extension to Alloy and its Analyzer, that allows users to annotate different model and specification fragments with different background colors (denoting different features), and run analysis commands to verify either a particular variant of the design or a full set of variants at once, to simplify the detection of particular combinations of features that may 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 to illustrate its advantages. Section 3 formally presents the new language extension, including the new typing rules to check the correctness of the annotations. Section 4 presents the extensions to the analysis engine of Alloy to support multi-variant analysis. Section 5 presents the evaluation of the proposed technique, using a series of examples to evaluate the efficiency of the new multi-variant analysis engine. Section 6 discusses some related work. Finally, Section 7 concludes the paper, namely presenting some ideas for future work.

2 Overview

Alloy4Fun [14] is a web-platform[1] developed by our team for learning Alloy and sharing models. Besides the online creation, analysis and sharing of models, Alloy4Fun had two additional goals: to provide some kind of auto-grading feature through the codification of secrets, and to collect information regarding usage patterns and typical pitfalls. To explore these features, we used Alloy itself to explore the design Alloy4Fun.

Modeling The structure of an Alloy[2] 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 reflexive 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 proprieties which are to be checked through *asserts*. In colorful Alloy provides, 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. Following the results of [9], 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 model simply allows models to be stored when shared by the user, and is thus comprised by models (**`sig`** `Model`, l. 5) assigned to at most one public link (**`sig`** `Link`, l. 11, through field `public` with **`lone`** multiplicity, l. 7).

[1] `alloy4fun.inesctec.pt`

[2] For a thorough presentation of the Alloy language and the Analyzer, please see [11].

Two additional constraints are enforced (inside a **`fact`**, l. 22): a link is assigned to exactly **`one`** model (l. 24) and all models have a public link assigned to them (l. 26). To this model, 4 features can be added:

① to collect usage patterns, the derivation tree of the models is stored, introducing a new field (`derivationOf`, l. 6) and an additional constraint to avoid cyclic dependencies (using reflexive closure, l. 28);
② models can have secrets defined (**`sig`** `Secret` as a subset of `Model`, 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 stored when commands are executed (**`sig`** `Command`, l. 14), rather than just when shared by the user, allowing finer data collection; such models store the command that originated them (field `command`, l. 9); moreover, the previous constraint on the existence of public links is removed (l. 26), since models created through command execution do not create links;
④ instances (**`sig`** `Instance`, l. 17) resulting from command execution can also be stored and shared; the constraint on links (l. 24) must be relaxed, since they may now point to an instance (field `link`, l. 20) rather than to a model.

Feature-oriented software development is usually accompanied by a feature model denoting which feature combinations are acceptable. In our example, instances are associated to the commands that created them, so the selection of feature 4 requires feature 3. To enforce that constraint, a fact is introduced (l. 1) that guarantees that variants for which it fails are unsatisfiable: if feature 4 is present but not feature 3, then an inconsistent formula is imposed (expression **`some none`**, since Alloy does not support Boolean constants). We expect that at the level of abstraction that (colorful) Alloy is employed, feature models will be simple and easily encoded as facts of this kind. Exploring whether that is the case or language extensions are needed is left for future work.

Analysis Once the model is defined, the user can instruct the Analyzer to generate scenarios for which a certain property hold through *run* commands, or instruct it to find counter-examples to a property expected to hold through *check* commands. These are provided with 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 for which those conditions hold, or smallest variant for which those conditions hold.

In our example, the run in l. 33 allows the user explore scenarios with commands stored in all 8 variants where feature 3 is selected (without feature 3 `command` does not exist, and a type error would be thrown). This will generate an arbitrary instance for one of the possible 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 that will generate instances under the 4 valid

```
1   fact FeatureModel {
2     // ④ requires ③
3     ④③~~some none~~③④ }
4
5   sig Model {
6     ①derivationOf : lone Model①,
7     public       : lone Link,
8     ②secret       : lone Link②,
9     ③command      : lone Command③ }
10
11  ②sig Secret in Model {}②
12
13  sig Link {}
14
15  ③sig Command {}③
16
17  ③④sig Instance {
18    instanceOf : one Command,
19    model      : set Model,
20    link       : one Link }④③
21
22  fact {
23    // Links are not shared between artifacts
24    all l : Link | one (public+②secret②+③④link④③).l
25    // All models have public links, unless commands are stored
26    ③~~all m : Model | one m.public~~③
27    // The model derivations form a forest
28    ①no m : Model | m in m.^derivationOf①
29    // Private and public links must be different
30    ②all m : Model | m.public != m.secret②
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    // Models without a public link can have at most one derivation
38    ①all m : Model | no m.public implies lone derivationOf.m① }
39  check OneDerivation with ① for 3
```

Fig. 1: Alloy4Fun specification in colorful Alloy.

variants with features 2 and 3, which the Analyzer will say is unsatisfiable. This was due to a bug on the constraint on secret links (l. 30), which does not hold when models are created from command execution (a possible fix would be to state `all m : Model | no m.public & m.secret`). Such issue could go unnoticed without variant-focused analysis commands.

After exploring scenarios, the user usually starts 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 model has no public link, than at most one model is derived from it. Through the check in l. 39 for all variants with feature 1 selected, the Analyzer quickly shows that that is not the case, generating a counter-example in a variant with features 1, 2 and 3, where the execution of commands allows two models to derive the same one. An analysis focused on individual variants could miss this possible issue arising from the interaction fo feature 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 model all conditional structures and behaviors that the model may encode, using an signature to denote which features are under consideration. This quickly renders the 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)
```

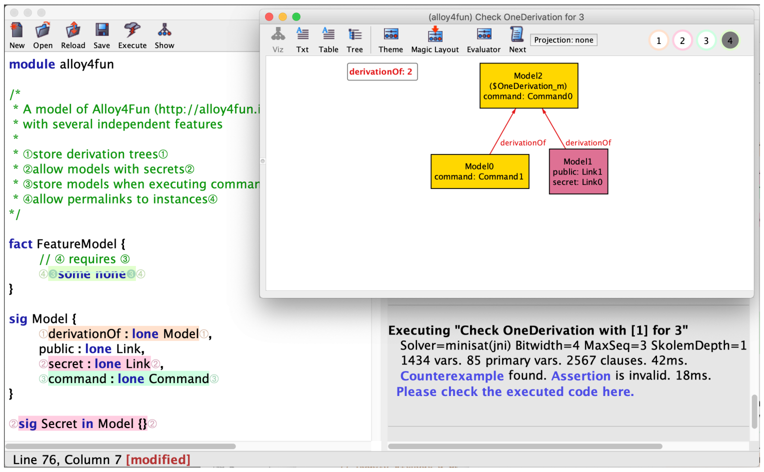
which can quickly become impenetrable and unmanageable.

The other alternative would be to rely on the Alloy module system and define variants as separate modules that import common elements. Opening a module imports the complete module, meaning that existing elements cannot be removed or changed. As a consequence, a base module with all parts not annotated 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 `Model → Model`. Negative annotations are even more troublesome, since they would have to be added by all models 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 [12]. 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 can still be saved and interpreted as a pure text file. Additionally, unlike [12] 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



```
module alloy4fun

/*
 * A model of Alloy4Fun (http://alloy4fun.i
 * with several independent features
 *
 * ①store derivation trees①
 * ②allow models with secrets②
 * ③store models when executing comman
 * ④allow permalinks to instances④
*/

fact FeatureModel {
	// ④ requires ③
	④❸some none❸④
}

sig Model {
	①derivationOf : lone Model①,
	public : lone Link,
	②secret : lone Link②,
	③command : lone Command③
}

②sig Secret in Model {}②
```

```
Executing "Check OneDerivation with [1] for 3"
   Solver=minisat(jni) Bitwidth=4 MaxSeq=3 SkolemDepth=1
   1434 vars. 85 primary vars. 2567 clauses. 42ms.
   Counterexample found. Assertion is invalid. 18ms.
  Please check the executed code here.
```

```
Line 76, Column 7 [modified]
```

Fig. 2: Counter-example to `OneDerivation`.

of features; and second, analysis commands to focus on particular sets of features. Features are identified by circled symbols, ⓒ and 🅒, denoting the presence and absence of a feature, respectively, for $1 \leq c \leq 9$ (ⓒ denotes either ⓒ or 🅒). 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 [9], and in 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 ⓒ will only exist in variants where c is selected, while those within negative delimiters 🅒 only exist if c is absent from the variant. Color annotations can be nested, denoting the conjunction of presence conditions (e.g., ④❸**`some none`**❸④ in Fig 1, l. 3, **`some none`** will be present in any variant with feature 4 selected but not feature 3). Likewise [12], 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.[3] 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.

[3] 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 ::= ⓒ 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).

For instance, `public+②link②` is interpreted as `public+(none→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, it instructs commands to consider every variant where the scope features are present/absent; otherwise, exactly that variant will be considered (negative features are spurious in that case). For instance, **`run {} with`** ①,❷ will consider every variant with feature 1 selected but not feature 2, while **`run {} with exactly`** ①,❷ will only consider the variant with exactly feature 1 selected. An additional feature mark ⊗ 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. 6 and 7 in the appendix for a kernel of expressions and paragraphs, respectively, to which the remainder language features can be converted (except comprehension, omitted for simplicity). For a mark, $\neg\textcircled{c}$ converts between the positive and negative version; $\mathbf{c}$ denotes a set marks; $\lfloor \mathbf{c} \rfloor$ expands $\mathbf{c}$ with all marks 🅒 such that ⓒ $\notin \mathbf{c}$; and $+\!\!+$ denotes the overriding of mappings. The context of the type rules is a mapping Γ from variables to the color of their declaration (and arity), and a set of color marks $\mathbf{c}$. An expression *e* of arity n is well-typed if $\Gamma, \mathbf{c} \vdash_n e$ (arity 0 denotes formulas), and the declaration *d* of i variables if $\Gamma, \mathbf{c} \vdash_{n_1 \ldots n_i} d$. A set of marks $\mathbf{c}$ is valid if $\vdash \mathbf{c}$. A paragraph *p* is well-typed under a similar context if $\Gamma, \mathbf{c} \vdash p$. The complete colorful model *m* comprised by paragraphs $p_1 \ldots p_i$, is well-typed if $\vdash m\ p_1 \ldots p_i$. This requires the prior collection of all global elements and the annotations under which they were declared, as calculated by function decls:[4]

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

Type rules check mainly for two kinds of issues. First, calls to global elements must occur in a context that guarantees its existence. For instance, signature `Secrets`, declared with feature 1, may not be called in a plain `fact { some Model }` 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 directly annotated, their context being instead defined by the color scope (for exact color scopes, the context is expanded with the negation of all features not present in it). 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 ①❶*e*❶①, *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;

[4] Function arity is an oversimplification, since calculating the arity of a bounding expression requires the arity of other declared sigs and fields.

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 projected 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 4 and 5 present the translation of the colorful model into the amalgamated version for paragraphs and expressions, respectively. The models are assumed to be well-typed at this stage,[5] and it is assumed that all unique colorful marks $\mathbf{c}_0$ that occur in the model 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 or negative color marks from a set, respectively. For a model $m\ p_1\ \ldots\ p_i$, the translation $[\![m\ p_1\ \ldots\ p_i]\!]_{\mathbf{c}_0}$ 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.[6] 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 relevant features are present/absent. 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 8a in the appendix presents an except of the amalgamated Alloy model for the Alloy4Fun colorful model.

The projected 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 yet natively support feature models, the $2^{\#\mathbf{c}}$ projected models must be generated and analyzed (although the process can be stopped once one of those models is found to be satisfiable). It is worth noticing, however, that the workaround proposed in Section 2 actually renders variants invalid in the feature model 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 8b in the appendix presents an except of

[5] This amalgamated version is the reason why we cannot rely on the Alloy type checker for the expanded versions, since references to supposedly absent elements would not be detected.

[6] To avoid collisions with the identifiers of the colorful model, the translation actually uses obfuscated identifiers these signatures.

$$\llbracket m\ p_1\ \dots\ p_i \rrbracket_{k\dots l} \equiv \begin{array}{l} m \\ \textbf{abstract sig}\ \texttt{Feature \{\}} \\ \textbf{one sig}\ \texttt{F}k\texttt{, ..., F}l\ \textbf{extends}\ \texttt{Feature \{\}} \\ \textbf{sig}\ \texttt{Variant}\ \textbf{in}\ \texttt{Feature \{\}} \\ \llbracket p_1 \rrbracket_\emptyset\ \dots\ \llbracket p_i \rrbracket_\emptyset \end{array}$$

$$\llbracket a\ m\ \textbf{sig}\ n\ x\ \{\ d_1, \dots, d_i\ \}\ b \rrbracket_{\mathbf{c}} \equiv \begin{array}{l} a\ \textbf{sig}\ n\ x\ \{\ \llbracket d_1 \rrbracket_{\mathbf{c}}, \dots, \llbracket d_i \rrbracket_{\mathbf{c}}\ \}\ \llbracket b \rrbracket_{\mathbf{c}} \\ \textbf{fact}\ \{\ (\llbracket \mathbf{c}^+ \rrbracket\ \textbf{in}\ \texttt{Variant}\ \textbf{and}\ \llbracket \mathbf{c}^- \rrbracket\ \textbf{not in}\ \texttt{Variant})\ \textbf{implies}\ m\ n \\ \qquad \textbf{else no}\ n\ \} \\ \textbf{fact}\ \{\ \mathsf{trans}(d_1)\ \dots\ \mathsf{trans}(d_i)\ \} \end{array}$$

where

$$\mathsf{trans}(\textcircled{c}\ d\ \textcircled{c}) = \mathsf{trans}(d)$$

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

$$\llbracket \textbf{pred}\ n\ [\ d\ ]\ b \rrbracket_{\mathbf{c}} \equiv \textbf{pred}\ n\ [\ \llbracket d \rrbracket_{\mathbf{c}}\ ]\ \llbracket b \rrbracket_{\mathbf{c}}$$

$$\llbracket \textbf{fun}\ n\ [\ d\ ]\ :\ e\ b \rrbracket_{\mathbf{c}} \equiv \textbf{fun}\ n\ [\ \llbracket d \rrbracket_{\mathbf{c}}\ ]\ :\ \llbracket e \rrbracket_{\mathbf{c}}\ \llbracket b \rrbracket_{\mathbf{c}}$$

$$\llbracket \textbf{fact}\ n\ b \rrbracket_{\mathbf{c}} \equiv \textbf{fact}\ n\ \llbracket b \rrbracket_{\mathbf{c}}$$

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

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

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

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

$$\llbracket \textcircled{c}\ p\ \textcircled{c} \rrbracket_{\mathbf{c}} \equiv \llbracket p \rrbracket_{\mathbf{c} \cup \{\textcircled{c}\}}$$

$$\llbracket v\ :\ m\ e \rrbracket_{\mathbf{c}} \equiv v\ :\ \textbf{set}\ \llbracket e \rrbracket_{\mathbf{c}}$$

$$\llbracket v\ :\ e_1\ m_1 \rightarrow m_2\ e_2 \rrbracket_{\mathbf{c}} \equiv v\ :\ \llbracket e_1 \rrbracket_{\mathbf{c}}\ \textbf{set} \rightarrow \textbf{set}\ \llbracket e_2 \rrbracket_{\mathbf{c}}$$

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

$$\llbracket \textcircled{c} \rrbracket \equiv \texttt{F}c$$

$$\llbracket \otimes \rrbracket \equiv \textbf{none}$$

Fig. 4: Paragraph translation into the amalgamated model with variability.

a projected Alloy model for the Alloy4Fun colorful model under an exact scope ②,③.

5 Evaluation

Our evaluation aimed to answer two questions regarding the feasibility of the approach, prior to developing more advances analysis procedures: *i*) is the analysis through the amalgamated model feasible? And if so, *ii*) does it always pay off in relation to the iterative analysis of 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.

$$\llbracket k \rrbracket_{\mathbf{c}} \equiv k$$
$$\llbracket n \rrbracket_{\mathbf{c}} \equiv n$$
$$\llbracket \square e \rrbracket_{\mathbf{c}} \equiv \square \llbracket e \rrbracket_{\mathbf{c}}$$
$$\llbracket e_1 \square e_2 \rrbracket_{\mathbf{c}} \equiv \begin{cases} \llbracket e_1 \rrbracket_{\mathbf{c}} \square \llbracket e_2 \rrbracket_{\mathbf{c}} & \text{if } \square \not\in \{+, \&, \textbf{or}, \textbf{and}\} \\ \mathsf{trans}(e_1) \square \mathsf{trans}(e_2) & \text{otherwise} \end{cases}$$

where

$$\mathsf{trans}(e) \equiv \llbracket \mathbf{c}^{+} \rrbracket \ \texttt{\textbf{in} Variant \textbf{and}}\ \llbracket \mathbf{c}^{-} \rrbracket\ \texttt{\textbf{not in} Variant \textbf{implies}}\ \llbracket e \rrbracket_{\mathbf{c}}\ \texttt{\textbf{else}}\ \mathsf{neutral}(\square, \mathsf{arity}(e))$$
$$\mathsf{neutral}(+, a) = \underbrace{\texttt{\textbf{none}} \rightarrow \ldots \rightarrow \texttt{\textbf{none}}}_{a}$$
$$\mathsf{neutral}(\&, a) = \underbrace{\texttt{\textbf{univ}} \rightarrow \ldots \rightarrow \texttt{\textbf{univ}}}_{a}$$
$$\mathsf{neutral}(\textbf{or}, a) = \texttt{\textbf{some none}}$$
$$\mathsf{neutral}(\textbf{and}, a) = \texttt{\textbf{no none}}$$
$$\llbracket \square\ d \ | \ e \rrbracket_{\mathbf{c}} \equiv \square \llbracket d \rrbracket_{\mathbf{c}} \ | \ \llbracket e \rrbracket_{\mathbf{c}}$$
$$\llbracket \textcircled{c}\ e\ \textcircled{c} \rrbracket_{\mathbf{c}} \equiv \llbracket e \rrbracket_{\mathbf{c} \cup \{\textcircled{c}\}}$$
$$\llbracket v \ : \ e \rrbracket_{\mathbf{c}} \equiv v \ : \ \llbracket e \rrbracket_{\mathbf{c}}$$

Fig. 5: Expression translation into the amalgamated model with variability.

5.1 Examples

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 [11], the latter is presented in stages to address different concepts, which are distributed as 3 distinct Alloy files. Our base model 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 [7] and models various options for the catalog structure of the platform. The base model is a catalog representing 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 to require feature 1; feature 5 also requires feature 4. The command evaluated tests whether all items are cataloged in every variant.

The *Graph* model is adapted from a compositional version from [1], exploring different classes of graphs. The base model 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) vertex 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. Two properties have been evaluated: whether the graph is connected, all nodes descending from the root, and whether for non-empty graphs there is 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. Another example defined by us is inspired by various vending machine examples commonly used in SPL literature (e.g., [8]). The base model of this dynamic example encodes the process of selecting and serving an item, extensible by introducing 1) the notion of price and payment; and 2) the possibility to select multiple items. The two features are independent. 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 a family of very simple models that we use in classes to explore different classes of relations. Each feature defines relations as 1) injective, 2) functional, 3) total and 4) surjective. Evaluated commands check alternative definitions of injectivity and functionality, as well as whether relations are associative.

5.2 Results

Table 1 depicts the performance times for the examples presented above, for varying scope. For each model, the table presents how many features it has (NF). Then, for each pair command / scope within a model, it presents how many different 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), the speedup of the former in relation to the latter (SU). Additionally, the slowest time for a projected variant (SP) is also presented. All experiments were run on a MacBook with a 2.4 GHz Intel Core i5 and 8GB memory using the MiniSAT SAT solver.

Results show that the amalgamated approach is indeed feasible, and in fact, it shows to always be faster than the iterative analysis except for one particular command of *Alloy4Fun*. The evaluation did however rise some interesting unexpected questions, due to how frequently the slowest analysis of a single projected variant is actually than the full amalgamated analysis. For the *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

Feature-oriented design [3] is a design paradigm where features are considered explicitly, tailored for feature-oriented software development. Several approaches have been proposed for feature-oriented design, but we focus on those that provide

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)
OwnGrandpa	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
E-commerce	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
Graph	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
Alloy4Fun	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
Vending	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	**30.7**	**26.1**	**0.85**	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
Bestiary	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
		Associative	8	8	6	2.8	9.4	3.38	2.5
					7	52.5	211.9	4.04	62.2
					8	230.2	891.9	3.88	309.1

specification languages supported by automated analysis tools. *fSMV* [15] is a compositional approach for SMV, where a basic system in pure SMV can be extended with features, modeled in new textual units (dubbed feature constructs). The behavior of base system may be overwritten by features, integrated automatically by means of a tool, called SFI, that compiles into pure SMV so that the verification can be performed by normal SMV model checkers. *fPromela* [5] provides instead an annotative approach, where features are introduced by a new user-defined data type structure named *features*. 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, FeatureHouse, in order to produce a final model. The composition is achieved by recursively superimposing and merging the features selected by a user; new fields can be added to signatures, while facts, predicates, or function are altered by overriding, so fine-grained variability points need a rewrite of the entire paragraph.

The idea to compare some kind of "amalgamated" model checking with the iterative analysis of all variants has been explored by [6], where the analysis of fSMV models through a symbolic algorithm for featured transition systems (FTS) was compared with that of the iteration of regular SMV models and 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.

Background colors [12] have been proposed as an annotative approach for feature-oriented software development. These are similar to the *#ifdef* statements in C compiler, the corresponding code fragments are only included when their associated features are selected. Similar to our approach, it offers a direct mechanism for developers to find whether a code fragment is associated with a feature, but annotations are part of the model itself, instead being handled by the supporting tool.

Techniques have been proposed for the type-checking of SPLs. One such technique is proposed in [13] but is tailored for Java programs. [7] presents a technique for checking the well-formedness of a model template against a feature model by mapping OCL well-formedness rules into to 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 of all projected variants.

Future work is planned one 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 Trans. Software Eng. **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. 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. pp. 335–344. ACM (2010)
7. 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. pp. 211–220. ACM (2006)
8. Fantechi, A., Gnesi, S.: Formal modeling for product families engineering. In: SPLC. pp. 193–202. IEEE (2008)
9. Feigenspan, J., Kästner, C., Apel, S., Liebig, J., Schulze, M., Dachselt, R., Papendieck, M., Leich, T., Saake, G.: Do background colors improve program comprehension in the #ifdef hell? Empirical Software Engineering **18**(4), 699–745 (2013)
10. Holzmann, G.J.: The model checker SPIN. IEEE Transactions on Software Engineering **23**(5), 279–295 (1997)
11. Jackson, D.: Software Abstractions - Logic, Language, and Analysis. MIT Press, revised edn. (2012)
12. Kästner, C., Apel, S., Kuhlemann, M.: Granularity in software product lines. In: Proceedings of the 30th International Conference on Software Engineering. pp. 311–320. ACM (2008)
13. 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)

14. Macedo, N., Cunha, A., Pereira, J., Carvalho, R., Silva, R., Paiva, A.C.R., Ramalho, M.S., Silva, D.: Sharing and learning Alloy on the web (2019), submitted, available at `https://nmacedo.github.io/pubs/Alloy4Fun.pdf`
15. Plath, M., Ryan, M.: Feature integration using a feature construct. Science of Computer Programming **41**(1), 53–84 (2001)

A Type rules and translation outputs

$$\dfrac{\Box \in \{\mathtt{none}, \mathtt{univ}\}}{\Gamma, \mathbf{c} \vdash_1 \Box} \qquad \dfrac{}{\Gamma, \mathbf{c} \vdash_2 \mathtt{iden}} \qquad \dfrac{v \mapsto (\mathbf{r}, n) \in \Gamma \qquad \mathbf{r} \subseteq \mathbf{c}}{\Gamma, \mathbf{c} \vdash_n v}$$

$$\dfrac{\Gamma, \mathbf{c} \vdash_n e \qquad n > 0}{\Gamma, \mathbf{c} \vdash_n \mathtt{\hat{}}e} \qquad \dfrac{\Gamma, \mathbf{c} \vdash_2 e}{\Gamma, \mathbf{c} \vdash_2 \sim e} \qquad \dfrac{\Gamma, \mathbf{c} \vdash_0 e}{\Gamma, \mathbf{c} \vdash_0 \mathtt{not}\ e} \qquad \dfrac{\Gamma, \mathbf{c} \vdash_0 e_1 \qquad \Gamma, \mathbf{c} \vdash_0 e_2}{\Gamma, \mathbf{c} \vdash_0 e_1\ \mathtt{and}\ e_2}$$

$$\dfrac{\Gamma, \mathbf{c} \vdash_n e_1 \qquad \Gamma, \mathbf{c} \vdash_n e_2 \qquad n > 0}{\Gamma, \mathbf{c} \vdash_0 e_1\ \mathtt{in}\ e_2} \qquad \dfrac{\Gamma, \mathbf{c} \vdash_n e_1 \qquad \Gamma, \mathbf{c} \vdash_n e_2 \qquad n > 0 \qquad \Box \in \{\&, +, -\}}{\Gamma, \mathbf{c} \vdash_n e_1\ \Box\ e_2}$$

$$\dfrac{\Gamma, \mathbf{c} \vdash_n e_1 \qquad \Gamma, \mathbf{c} \vdash_m e_2 \qquad k = n + m - 2 \qquad n, m, k > 0}{\Gamma, \mathbf{c} \vdash_k e_1\ .\ e_2}$$

$$\dfrac{\Gamma, \mathbf{c} \vdash_n e_1 \qquad \Gamma, \mathbf{c} \vdash_m e_2 \qquad k = n + m \qquad n, m > 0}{\Gamma, \mathbf{c} \vdash_k e_1 \rightarrow e_2}$$

$$\dfrac{\Gamma, \mathbf{c} \vdash_1 d \qquad \Gamma, \mathbf{c} +\!\!+\ v \mapsto (\emptyset, 1) \vdash_0 e}{\Gamma, \mathbf{c} \vdash_0 \mathtt{all}\ v\ :\ d\ |\ e} \qquad \dfrac{\Gamma, \mathbf{c} \vdash_n e \qquad \neg\textcircled{c} \notin \mathbf{c}}{\Gamma, \mathbf{c} \cup \{\textcircled{c}\} \vdash_n \textcircled{c}\ e\ \textcircled{c}}$$

Fig. 6: Type rules for kernel expressions.

$$\dfrac{\Gamma, \mathbf{c} \vdash_{n_1} d_1 \qquad \ldots \qquad \Gamma, \mathbf{c} \vdash_{n_i} d_i \qquad \Gamma, \mathbf{c} \vdash_0 b \qquad \Gamma, \mathbf{c} \vdash_1 h \qquad n_1 \ldots n_i > 0}{\Gamma, \mathbf{c} \vdash a\ m\ \mathtt{sig}\ n\ x\ h\ \{\ d_1, \ldots, d_i\ \}\ b}$$

$$\dfrac{\Gamma, \mathbf{c} \vdash_{n_1} d_1 \qquad \ldots \qquad \Gamma, \mathbf{c} \vdash_{n_i} d_i \qquad \Gamma, \mathbf{c} \vdash_n e \qquad \Gamma, \mathbf{c} \vdash_n b \qquad n, n_1 \ldots n_i > 0}{\Gamma, \mathbf{c} \vdash \mathtt{fun}\ n\ [\ d_1, \ldots, d_i\ ]\ :\ e\ b}$$

$$\dfrac{\Gamma, \mathbf{c} \vdash_{n_1} d_1 \qquad \ldots \qquad \Gamma, \mathbf{c} \vdash_{n_i} d_i \qquad \Gamma, \mathbf{c} \vdash_0 b \qquad n_1 \ldots n_i > 0}{\Gamma, \mathbf{c} \vdash \mathtt{pred}\ n\ [\ d_1, \ldots, d_i\ ]\ b} \qquad \dfrac{\Gamma, \mathbf{c} \vdash_0 b}{\Gamma, \mathbf{c} \vdash \mathtt{fact}\ n\ b}$$

$$\dfrac{\Gamma, \mathbf{c} \vdash_0 b \qquad \Box \in \{\mathtt{run}, \mathtt{check}\} \qquad \vdash \mathbf{c}}{\Gamma, \emptyset \vdash \Box\ n\ b\ \mathtt{with}\ \mathbf{c}\ s} \qquad \dfrac{\Gamma, \lfloor \mathbf{c} \rfloor \vdash_0 b \qquad \Box \in \{\mathtt{run}, \mathtt{check}\} \qquad \vdash \mathbf{c}}{\Gamma, \emptyset \vdash \Box\ n\ b\ \mathtt{with\ exactly}\ \mathbf{c}\ s}$$

$$\dfrac{\Gamma, \mathbf{c} \vdash p \qquad \neg\textcircled{c} \notin \mathbf{c}}{\Gamma, \mathbf{c} \cup \{\textcircled{c}\} \vdash \textcircled{c}\ p\ \textcircled{c}} \qquad \dfrac{\vdash \mathbf{c} \qquad \neg\textcircled{c} \notin \mathbf{c}}{\vdash \textcircled{c}, \mathbf{c}} \qquad \dfrac{\Gamma, \mathbf{c} \vdash_n e \qquad n > 0}{\Gamma, \mathbf{c} \vdash_n v\ :\ e} \qquad \dfrac{\Gamma, \mathbf{c} \vdash_n d \qquad \neg\textcircled{c} \notin \mathbf{c}}{\Gamma, \mathbf{c} \cup \{\textcircled{c}\} \vdash_n \textcircled{c}\ d\ \textcircled{c}}$$

$$\dfrac{\Gamma = \mathsf{decls}(\emptyset, p_1, \ldots, p_n) \qquad \Gamma, \emptyset \vdash p_1 \qquad \ldots \qquad \Gamma, \emptyset \vdash p_i}{\vdash m\ p_1\ \ldots\ p_i}$$

Fig. 7: Type rules for kernel paragraphs.

```
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 Model {
  derivationOf : set Model,
  ...,
  command      : set Command }
sig Link {}
sig Command {}

fact {
  F3 in Variant implies
    command in Model→lone Command else no command
  F3 not in Variant implies no Command
  ... }

fact Links {
  all : Link | one (public+
    (F2 in Variant implies secret else none→none)+
    (F3+F4 in Variant implies link else none→none)).l
  F3 not in Variant implies
    (all m : Model | one m.public)
  ... }
```

(a) Amalgamated translation.

```
sig Model{
  public:  lone Link,
  secret:  lone Link,
  command: lone Command }

sig Secret in Model{}

sig Link{}

sig Command{}

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

(b) ②,③ projection.

Fig. 8: Excerpts of the translations for the Alloy4Fun colorful model.