Feature and Class Models in Clafer: Mixed, Specialized, and Coupled

University of Waterloo Technical Report CS-2010-10

Kacper Bąk[1], Krzysztof Czarnecki[1], and Andrzej Wąsowski[2]

[1] Generative Software Development Lab, University of Waterloo, Canada,
{kbak,kczarnec}@gsd.uwaterloo.ca
[2] IT University of Copenhagen, Denmark, wasowski@itu.dk

Abstract. We present *Clafer*, a class modeling language with first-class support for feature modeling. We designed Clafer as a concise notation for class models, feature models, mixtures of class and feature models (such as components with options), and models that couple feature models and class models via constraints (such as mapping feature configurations to component configurations). Clafer also allows arranging models into multiple specialization and extension layers via constraints and inheritance. We identified four key mechanisms allowing a class modeling language to express feature models concisely and show that Clafer meets its design objectives using a sample product line.

1 Introduction

Both feature and class modeling have been used in software product line engineering to model variability. Feature models are tree-like menus of mostly Boolean—but sometimes also integer and string—configuration options, augmented with cross-tree constraints [15]. These models are typically used to show the variation of *user-relevant* characteristics of products within a product line. In contrast, class models, as supported by the Unified Modeling Language (UML), have been used to represent the components and connectors of *product line architectures* and the valid ways to connect them. Thus, the nature of variability expressed by each type of models is different: feature models capture simple selections from predefined (mostly Boolean) choices within a fixed (tree) structure; and class models support making new structures by creating multiple instances of classes and connecting them via object references.

Over the last eight years, the distinction between feature and class models has been blurred somewhat in the literature due to feature modeling extensions, such as *cardinality-based feature modeling* [10, 3], or attempts to use composition hierarchies to express feature models in UML class models [7, 11]. A key driver behind these two developments has been the desire to express components and configuration options in a single notation (e.g., see [9]). Cardinality-based feature modeling achieves this goal by extending feature modeling with multiple instantiation and references. Class modeling, while natively supporting multiple

instantiation and references, can also support feature modeling by a stylized use of composition hierarchy and UML profiling mechanisms.

Both developments have notable drawbacks, however. An important advantage of feature modeling as originally defined by Kang et al. [15] is its simplicity; several respondents to a recent survey confirmed this view [16]. Extending feature modeling with multiple instantiation and references diminishes this advantage by introducing additional complexity. Further, models that contain significant amounts of multiply-instantiatable features and references can be hardly called feature models in the original sense; they are more of class models. On the other hand, whereas the model parts requiring multiple instantiation and references are naturally expressed as class models, the parts that have feature-modeling nature cannot be expressed elegantly in class models, but only clumsily simulated using composition hierarchy and certain modeling patterns.

We present *Clafer* (class, feature, reference), a class modeling language with first-class support for feature modeling. The language was designed to naturally express class models, feature models, mixtures of class and feature models (such as components with options), and models that couple feature models with class models via constraints (such as mapping feature configurations to component configurations). Clafer also allows arranging models into multiple specialization and extension layers via constraints and inheritance. We show that Clafer meets its design objectives using a sample product line. Clafer's design identifies four key mechanisms allowing a class modeling language to express feature models concisely.

We developed a translator from Clafer to Alloy [13], a class modeling language with a powerful constraint notation. The translator gives Clafer precise (translational) semantics and allows analyzing Clafer models using the Alloy Analyzer. In particular, we show how the analyzer can be used to instantiate our sample product line. In future, we will investigate model-specific translations to a range of reasoners in order to provide efficient support for operations on Clafer models and model instances, such as consistency checks, configuration, refactoring, and specialization.

The paper is organized as follows. We introduce our running example in Sect. 2. We discuss the challenges of representing the example using either only class modeling or only feature modeling and define a set of design objectives for Clafer in Sect. 3. We then present Clafer in Sect. 4 and demonstrate that it satisfies these objectives. We revisit the objectives in Sect. 5 and discuss Clafer's current status and future work in Sect. 6. After comparing Clafer with related work in Sect. 7, we conclude in Sect. 8.

2 Running Example: Telematics Product Line

Vehicle telematics systems integrate multiple telecommunication and information processing functions in an automobile, such as navigation, driving assistance, emergency and warning systems, hands-free phone, and entertainment functions, and present them to the driver and passengers via multimedia displays. Figure 1

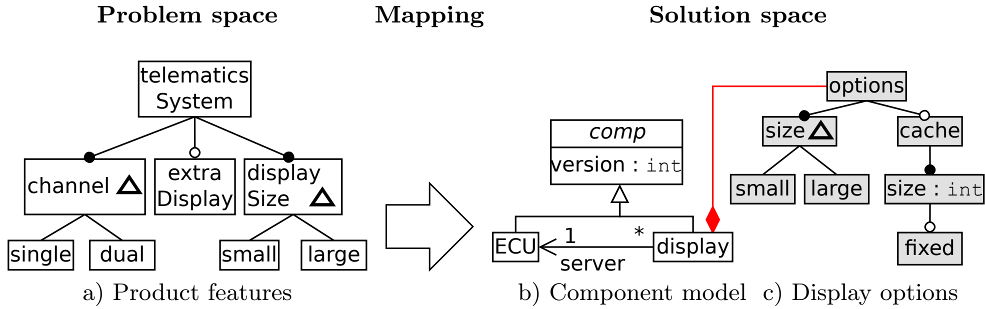


Fig. 1. Telematics product line

presents a variability model of a sample telematics product line, which we will use as a running example. The product line offers the features summarized in Fig. 1a, the *problem-space* feature model. A concrete telematics system can support either a single or two independently controllable channels; two channels afford independent programming for the driver and the passengers. The choice is represented as the xor-group channel, marked by the small triangle. By default, each channel has one associated display; however, we can add one extra display per channel, as indicated by the optional feature extraDisplay. Finally, we can choose large or small displays (displaySize).

Figure 1b shows the actual components that make up a telematics system, represented by a class model. There are two types of components: ECUs (electronic control units) and displays. Each display has exactly one ECU as its server. Further, all components have a version.

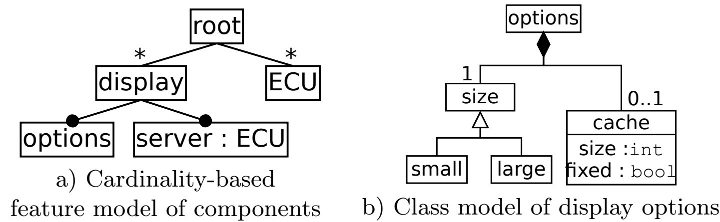
Components may have configuration options themselves. In our example, we can configure the display size and enable a display cache (see Fig. 1c). We can also specify the cache size and decide whether the size is fixed or can be updated at runtime. Thus, the *solution space* consists of a class model of component types and a feature model of component options.

Finally, the variability model requires a mapping from the problem-space feature configurations to the component and option configurations in the solution space. A big arrow in Fig. 1 indicates this mapping; we will specify it completely and precisely in Sect. 4.3.

3 Feature vs. Class Modeling

The solution space in Fig. 1 contains a class and a feature model. To capture our intention, the models are connected via UML composition. Since the precise semantics of such notational mixture are not clear, this connection should be understood only informally for now.

We have at least two choices to represent both the components and the options in a single notation. The first choice is to show the entire solution space model using cardinality-based feature modeling. Figure 2a shows the component



a) Cardinality-based feature model of components b) Class model of display options

Fig. 2. Feature model as class model and vice versa

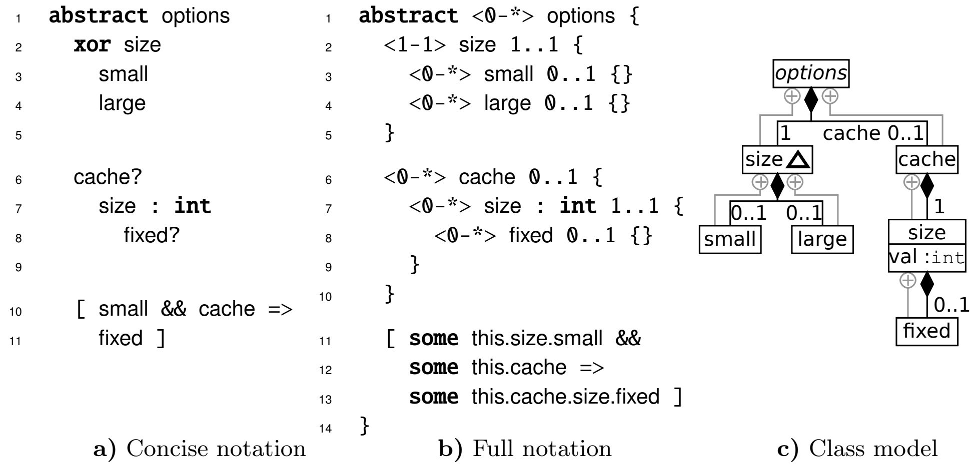
part of the model (the subfeatures of options are elided). The model introduces a synthetic root feature; display and ECU can be multiply instantiated; and display has server subfeature representing a reference to instances of ECU. Versions could be added to both display and ECU to match the class model (Fig. 1b) or we could extend the notation with inheritance. Doing the latter would bring the cardinality-based feature modeling notation very close to class modeling, posing the question whether a class modeling notation should be used for the entire solution space model instead.

We explore the class modeling alternative in Fig. 2b. The figure shows only the options model, as the component model remains unchanged (as in Fig. 1b). Subfeature relationships are represented as UML composition and feature cardinalities correspond to composition cardinalities at the part end. The xor-group is represented by inheritance and cache size and fixed as attributes of cache.

Representing a feature model as a UML class model worked reasonably well for our small example; however, the approach has several drawbacks. First, the feature model showed fixed as a property of size by nesting; this intention is lost in the class model. As solution would be to create a separate class size, containing the size value and fixed; thus, adding a subfeature to a feature represented as a class attribute requires refactoring. Another issue is that the name of the new class size would clash with the class size representing the display size; thus, we would have to rename one of them. Further, converting an xor-group to an or-group in feature modeling is simple: the empty triangle needs to be replaced by a filled one. For example, displaySize (Fig. 1a) could be converted to an or-group in a future version of the product line to allow systems with both large and small displays simultaneously. Such change is tricky in UML class models: we would have to either allow one to two objects of type displaySize and write an OCL constraint forbidding two objects of the same subtype (small or large) or use overlapping inheritance (i.e., multiple classification). Thus, the representation of feature models in UML, rather than a dedicated notation, incurs additional complexity. Similar argument can be made for Alloy.

The examples in Fig. 2 lead us to the following two conclusions:

(1) *"Cardinality-based feature modeling" is a misnomer.* Cardinality-based feature modeling encompasses multiple instantiation and references, mechanisms characteristic of class modeling, and could even be extended further towards class modeling, e.g., with inheritance; however, the resulting notation can be hardly



a) Concise notation **b)** Full notation **c)** Class model

Fig. 3. Feature model in Clafer and corresponding UML class model

still referred to as "feature modeling', as it would go clearly beyond the original scope of feature modeling [15].

(2) *Existing class modeling notations such as UML and Alloy do not offer first-class support for feature modeling.* Feature models can still be represented in these languages; however, the result carries undesirable notational complexity.

The solution to these two issues is to design a *class modeling language with first-class support for feature modeling*. We postulate that such a language should satisfy the following design goals:

1. *Provide a concise notation for feature modeling*
2. *Provide a concise notation for class modeling*
3. *Allow mixing feature models and class models*
4. *Use minimal number of concepts and have uniform semantics*

The last goal expresses our desire that the new language should unify the concepts of feature and class modeling as much as possible, both syntactically and semantically. In other words, we do not want a hybrid language.

4 Clafer: Class Modeling with First-Class Support for Feature Modeling

We explain the meaning of Clafer models by relating them to their corresponding UML class models. Figure 3 shows the display options feature model in Clafer (a) and the the corresponding UML model (c). Figure 4 shows the component class model in Clafer; Fig. 1b has the corresponding UML model.

A Clafer model is a set of type definitions, features, and constraints. A type can be understood as a class or feature type; the distinction is immaterial. Figure 3a contains options as single top-level type definition. The definition contains

a hierarchy of features (lines 2-8) and a constraint (lines 10-11); the enclosing type provides a separate name space for this content. The `abstract` modifier indicates that no instance of the type will be created, unless extended by a concrete type.

A type definition can contain one or more *features*; the type options has two (direct) features: size (line 2) and cache (line 6). Features are slots that can contain one or more instances or references to instances. Mathematically, features are binary relations. They correspond to attributes or role names of association or composition relationships in UML. For example, in Fig. 4, the feature version (line 2) corresponds to the attribute of the class comp in Fig. 1b; and the feature server (line 6) corresponds to the association role name next to the class ECU in Fig. 1b. Features declared using the colon notation and having no subfeatures, like in server : ECU, are *reference features*, i.e., they hold references to instances.

Features that do not have their type declared using the colon notation, such as size (line 2) and cache in Fig. 3a, or have subfeatures, such as size (line 7) in Fig. 3a, are *containment features*, i.e., features that contain instances. An instance can be contained by only one feature, and no cycles in instance containment are allowed. These features correspond to role names at the part end of composition relationships in UML. For example, the feature cache in Fig. 3a corresponds to the role name cache next to the class cache in Fig. 3c. By a UML convention, the role name at the association or composition end touching a class is, if not specified, same as the class name.

A containment feature definition creates a feature and, implicitly, a new concrete type, both located in the same name space. For example, the feature definition cache (line 6) in Fig. 3a defines both the feature cache, corresponding to the role name in Fig. 3c, and, implicitly, the type cache, corresponding to the class cache in Fig. 3c. The new type is nested in the type options; in UML this nesting means that the class cache is an inner class of the class options, i.e., its full name is options::cache. Figure 3c shows UML class nesting relations in light color. Class nesting permits two classes named size in a single model, because each enclosing class defines an independent name scope.

The feature size (line 7) in Fig. 3a is a containment feature of general form, the implicitly defined type is a structure containing a reference, here to `int`, and a subfeature, fixed. This type corresponds to the class cache::size in Fig. 2b.

Features have *feature cardinalities*, which constrain the number of instances or references that a given feature can contain. Cardinality of a feature is specified by an interval $m..n$, where $m \in \mathbb{N}, n \in \mathbb{N} \cup \{*\}, m \leq n$. Feature cardinality specification follows the feature name or its reference type, if any.

```
1 abstract comp
2   version : int
3
4 abstract ECU extends comp
```

```
5 abstract display extends comp
6   server : ECU
7   'options
8   [ version >= server.version ]
9
```

Fig. 4. Class model in Clafer

Conciseness is an important goal for Clafer; therefore, we provide syntactic sugar for common constructions. Figures 3a and 3b show the same Clafer model; the first one is written in concise notation, while the second one is completely desugared code with resolved names in constraints.

Clafer provides syntactic sugar similar to syntax of regular expressions: `?` or `lone` (optional) denote 0..1; `*` or `any` denote 0..∗; and `+` or `some` denote 1..∗. For example, `cache` (line 6) in Fig. 3 is an optional feature. No feature cardinality specified denotes 1..1 (mandatory) by default, modulo three exceptions explained shortly. For example, `size` (line 7) in Fig. 3a is mandatory.

Features and types have *group cardinalities*, which constrain the number of child instances, i.e., the instances contained by subfeatures. Group cardinality is specified by an interval $\langle m{-}n \rangle$, with the same restrictions on m and n as for feature cardinalities, or by a keyword: `xor` denotes $\langle 1{-}1 \rangle$; `or` denotes $\langle 1{-}* \rangle$; and `mux` denotes $\langle 0{-}1 \rangle$; further, each of the three keywords makes subfeatures optional by default. If any, a group cardinality specification precedes a feature or type name. For example, `xor` on `size` (line 2) in Fig. 3a states that only one child instance of either `small` or `large` is allowed. Because the two subfeatures `small` and `large` have no explicit cardinality attached to them, they are both optional (cf. Fig. 3b). No explicit group cardinality stands for $\langle 0{-}* \rangle$, except when it is inherited as illustrated later.

Constraints are a significant aspect of Clafer, because they can express dependencies among features or restrict string or integer values. Constraints are always surrounded by square brackets and are a conjunction of first-order logic expressions. We modeled constraints after Alloy; the Alloy constraint notation is elegant, concise, and expressive enough to restrict both feature and class models. Logical expressions are composed of terms and logical operators. Terms either relate values (integers, strings) or are navigational expressions. The value of navigational expression is always a relation, therefore each expression must be preceded by a *quantifier*, such as `no`, `one`, `lone` or `some`. However, lack of explicit quantifier (Fig. 3a) stands for `some` (Fig. 3b), meaning that the relation cannot be empty.

Each feature in Clafer introduces a local namespace, which is rather different from namespaces in popular programming languages. Name resolution is important in two cases: 1) resolving type names used in feature and type definitions and 2) resolving feature names used in constraints. In both cases, names are path expressions, used for navigation like in OCL or Alloy, where the dot operator joins two relations. A name is resolved in a context of a feature in up to four steps. First, it is checked to be a special name like `this`. Secondly, the name is looked up in subfeatures in breadth-first search manner. If it is still not found, the algorithm searches in the top-level definition that contains the feature in its hierarchy. Otherwise, it searches in other top-level definitions. If the name cannot be resolved or is ambiguous within a single step, an error is reported.

Clafer supports single inheritance. In Fig. 4, the type `ECU` inherits features and group cardinality of it supertype. The type `display` extends `comp` by adding

two features and a constraint. The reference feature server points to an existing ECU instance. The meaning of ‘options notation is explained in Sect. 4.1.

The constraint defined in the context of display states that display's version cannot be lower than server's version. To dereference the server feature, we use dot, which then returns version. Although version is itself just a reference, Clafer automatically infers that it should compare the actual integer values and not just references.

4.1 Mixing via Quotes and References

Mixing class and feature models in Clafer is achieved via *quotation* (see line 7 in Fig. 4) or references. Syntactically, quotation is just a name of abstract type preceded by left quote (‘), which in the example is expanded as options extends options. The first name indicates a new feature, and the second refers to the abstract type. Semantically, this notation creates a containment feature options with a new concrete type display.options, which extends the top-level abstract type options from Fig. 3a. The concrete type inherits group cardinality and features of its supertype. By using quotation only type is shared, but not instances. References, on the other hand, are used for sharing instances.

The following example highlights the difference:

```
displayOwningOptions *
  ‘options -- shorthand for options extends options
```

In the above snippet, each instance of displayOwningOptions will have its own instance of type options. Note that Clafer assumes the existence of an implicit *root object*; thus, a feature definition, such as displayOwningOptions above, defines both a subfeature of the root object and a new top-level concrete type.

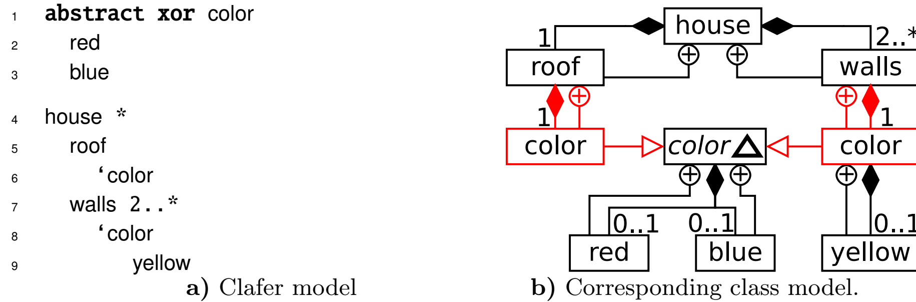
Now consider the following code:

```
options *
  -- content as in options in Fig. 3a

displaySharingOptions *
  sharedOptions : options
```

Each instance of displaySharingOptions has a reference named sharedOptions pointing to an instance of options. Although there can be many references, they might all point to the same instance living somewhere outside displaySharingOptions.

Clafer tries to minimize the number of language concepts. For example, it does not have a built-in enumeration type. There are several ways of defining enumerations in Clafer; we present one of them in Fig. 5. Quotation can simulate enumerations in the following way. First, we define an abstract feature color, with xor group cardinality. Next, the feature is quoted as a subfeature of another type, house in our example. Each house contains one roof and at least two walls. A roof can be painted in one of previously defined colors; however, a wall can be painted



a) Clafer model **b)** Corresponding class model.

Fig. 5. Simulating enumerations in Clafer using `xor` and quotation

in one of three colors since we extended the original set of two with an additional color. The corresponding UML model is shown in Fig. 5b. The abstract type *color* is shared between roof and walls via inheritance; however, no instances of type *color* are shared. Elements introduced by quotation are marked in red.

4.2 Specializing via Inheritance and Constraints

Let us go back to our telematics product line example. The class model of architecture as presented in Fig. 4 is a very generic metamodel, representing infinitely many different products. We would like to *specialize* and *extend* it to create a particular *architectural template*. A template makes most of the architectural structure fixed, but leaves some points of variability.

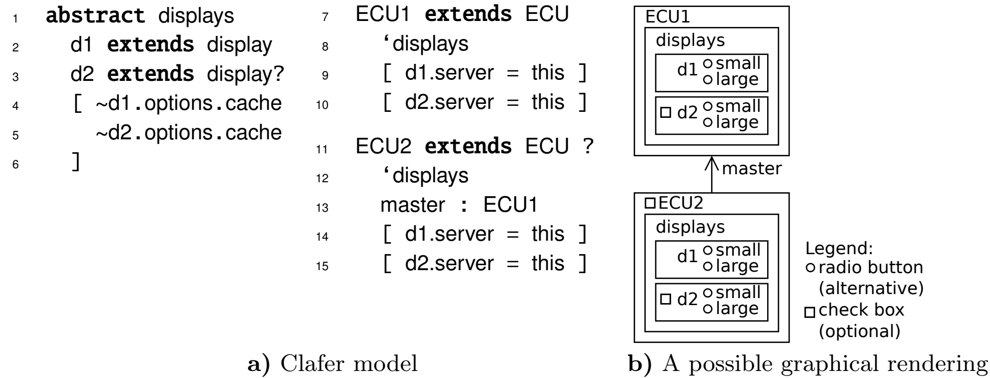
Figure 6a shows such a template for our example. We achieve specialization via inheritance and constraints. In our example, a concrete product must have at least one ECU (ECU1) and can optionally have another ECU (ECU2). Similarly, each ECU has either one display (d1) or two displays (d1 and d2), but none of the displays has cache. Besides, we need to constrain the server reference in each display, so that it points to its associated ECU. The reference this points to the current instance of ECU1. Also, ECU2 extends the base type with master, pointing to ECU1 as the main control unit.

Figure 6b visualizes the template in a domain-specific notation, showing both the fixed parts, e.g., mandatory ECU1 and d1, and the variable parts, e.g., alternative display sizes (radio buttons) and optional ECU2 and d2 (checkboxes).

The template represents knowledge that was not available in the metamodel. We could push forward this approach until there is no variability and a concrete product is defined. Moreover, we could use multiple layers of specialization or extension. It is all possible to express using inheritance and constraints.

4.3 Coupling via Constraints

Having defined the architectural template, we are ready to expose the remaining variability points as a product-line feature model. Figure 7 shows this model (cf. Fig. 1a) along with a set of constraints coupling its features to the variability



a) Clafer model **b)** A possible graphical rendering

Fig. 6. Architectural template

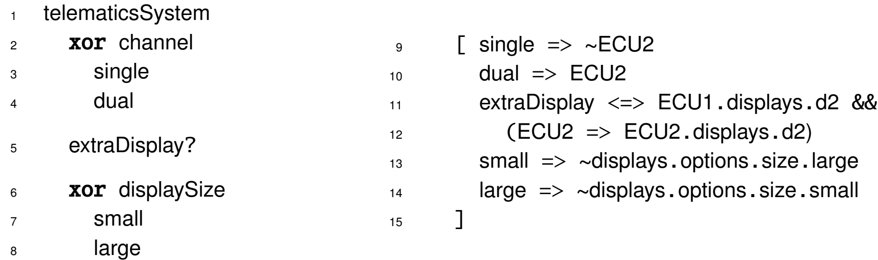


Fig. 7. Feature model with mapping constraints

points of the template. Note that the template allowed the presence of the extra displays (ECU1.d2 and ECU2.d2) and the size of every display to vary independently; however, we further restrict the variability in the feature model, requiring either all present ECUs to have an extra display or all to have no extra display and either all present displays to be small or all to be large.

Constraints allow us restricting a model to a single configuration. Figure 8 shows a top-level constraint defining a single product, with two ECUs, two large displays per ECU, and all components in version 1.

We tested our approach by automatically translating the Clafer model to Alloy and subjecting the resulting code to the Alloy Analyzer. The analyzer generates the expected instance of the product-line, which confirms that the model is not constrained too much. The solution is represented as a graph, where vertexes correspond to signatures and edges correspond to relations in the Alloy model. The output graph is unique up to structural equivalence. The Alloy Analyzer generates multiple instances of our example, but all of them have exactly the same structure within tested scope (each signature instantiated no

```
1  -- concrete product
2  [ dual && extraDisplay && telematicsSystem.size.large && comp.version == 1 ]
```

Fig. 8. Constraints determining a single product

more than 27 times). Although Alloy can detect some equivalent solutions, it does not perform full graph isomorphisms detection.[3]

5 Clafer's Design Goals Revisited

Let us revisit Clafer's design goals from Sect. 3.

(1) Clafer provides a concise notation for feature modeling (e.g., Fig. 3). Our language design reveals four key ingredients allowing a class modeling language to provide a concise notation for feature modeling:

- *Containment features*: A containment feature definition creates both a feature and a type in one step; for example, all features in Figs. 3 and 7 are of this type. Neither UML nor Alloy provide this mechanism.
- *Feature nesting*: Feature nesting is a single construct accomplishing both instance composition and type nesting. UML provides composition; however, type nesting has to be specified separately (cf. Fig. 3c). Alloy has no built-in support for composition and thus requires explicit set-up of parent-child constraints. It also has no signature nesting; signature name clashes need to be avoided differently, e.g., by using prefixes.
- *Group constraints*: Group constraints are defined concisely as intervals. Group constraints can be expressed in OCL or Alloy; however, the resulting encoding can be lengthy since it requires enumerating reference features.
- *Constraints with default quantifiers*: Default quantifiers on relations, such as `some` in Fig. 3, allow us writing constraints that look like propositional logic, even though their underlying semantics is first-order predicate logic.

Let us compare the size of the Clafer and Alloy models of the running example. With similar code formatting (no comments and blank lines), Clafer representation has 43 LOC and the automatically generated Alloy code is over two times longer. Since the Alloy model contains many long lines, let us also compare source file sizes: 1kb for Clafer and over 4kb for Alloy. The code generator favors conciseness of the translation over uniformity of the generated code, but in the worst case the lack of the previously listed constructs makes Alloy models necessarily larger. Additional language differences tip the balance further in favor of Clafer. For example, an abstract type definition in Clafer guarantees that the type will not be automatically instantiated; however, unextended abstract sets can be still instantiated by Alloy Analyzer. Therefore, each abstract signature in Alloy needs to be extended by an additional signature and then constrained.

(2) Clafer provides a concise notation for class modeling (e.g., Fig. 4).

[3] The Clafer model of the running example, the generated Alloy code, and the generated instance are available at `http://www.cs.uwaterloo.ca/~kbak/clafer/`.

(3) Clafer allows mixing feature and class models. Quotations allow reusing feature or class types in multiple model locations; references allow reusing both types and instances. Figure 7 shows how separate feature and class models can be related via constraints.
(4) Clafer tries to use a minimal number of concepts and has uniform semantics. While integrating feature modeling into class modeling, our goal was to avoid creating a hybrid language with duplicate concepts. In Clafer, there is no distinction between class and feature types. Features are relations and thus, besides their obvious role in feature modeling, they also play the role of attributes in class modeling. We also contribute a simplification to the realm of feature modeling: Clafer does not have an explicit feature group construct; instead, every feature can use a group cardinality to constrain the number of its children. We believe that this is an important simplification, as we no longer need to distinguish between "grouping features", i.e., features used purely for grouping, such as menus, and feature groups. In Clafer, the grouping intention and grouping cardinalities are orthogonal: any feature could be marked as a grouping feature via an annotation and any feature may or may not chose to impose grouping constraints on its children. Further, constructions such as enumeration types are built easily from the basic ingredients. Finally, both feature and class modeling have a uniform semantics in Clafer: a Clafer model instance, just like Alloy's, is a set of relations.

6 Current Status and Future Work

Clafer is still in the early stage of development. Although we are mainly focused on semantics of the language, we also spent significant amount of time on defining clean, intuitive syntax. We designed the language simultaneously with the Clafer-to-Alloy translator, which automates many tasks and enables us to experiment with new language constructions. The translator is written in Haskell and comprises several chained modules: lexer, layout resolver, parser, desugarer, semantic analyzer, and code generator. Layout resolver makes brackets grouping subfeatures optional. Feature hierarchies are resolved by means of code indentation. Clafer is composed of two languages: the core and the full language. The first one is a minimal language with well-defined semantics. The latter is built on top of the core language and provides large amount of syntactic sugar. Semantic analyzer is responsible for resolving names and dealing with inheritance. The code generator translates the core language constructions into Alloy.

Clafer-to-Alloy translator is able to generate feature models with simple constraints. Current work concerns mostly cross-tree constraints, providing support for operations on native types, such as integers or strings, and polishing syntax. In the nearest future, we plan to find out the most suitable name resolution strategies and provide a module system. Moreover, we would like to add type checking and inference rules and see what role they play in our language.

Clafer's applications include variability modeling and metamodeling. Our running example demonstrated the former. In the realm of metamodeling, we

used a language similar to Clafer, though without support for constraints and implemented as a profile on Eclipse's metamodeling language *ecore*, to define metamodels of domain-specific languages (DSLs) for framework APIs [1]. These DSLs specify components with rich property hierarchies as feature models.

Both variability modeling and metamodeling can benefit from a declarative constraint-based language supported by reasoners. Reasoners can help uncover flaws in models and assist in model evolution and model instantiation (i.e., configuration). For example, Alloy analyzer helped us discover that our original Clafer code was missing several constraints (specifically lines 9-10 and 14-15 in Fig. 6a and line 14 in Fig. 7). Some software platforms already provide configuration tools using reasoners; for example, Eclipse uses a SAT solver to help users select valid sets of plug-ins [17].

In future, we plan exploring different target reasoners and translation strategies for Clafer. We envision syntactic analyzers that classify Clafer models as belonging to specific sublanguages and using this classification to use the most efficient reasoner and encoding for each model. We also plan providing translations from variability modeling languages used in practice, such as KConfig [18] or CDL [5], to Clafer, and thus making these languages accessible to reasoners. In a sense, we hope to use Clafer as a pivot language, connecting different specialized variability modeling languages and different reasoners.

7 Related Work

Asikainen and Männistö present Forfamel, a unified conceptual foundation for feature modeling [3]. The basic concepts underlying Forfamel and Clafer are similar; Forfamel also includes subfeature, attribute, and subtype relations. The main difference is that Clafer's focus is to provide concise concrete syntax, such as being able to define feature, feature type, and nesting all just by stating an indented feature name. Also, the conceptual foundations of Forfamel and Clafer differ in many respects; e.g., features in Forfamel correspond to Clafer's instances, but features in Clafer are relations. Also, a feature instance in Forfamel can have one or more parents; in Clafer, an instance can have at most one parent. These differences likely stem from the difference in perspective: Forfamel takes a feature modeling perspective and aims at providing a foundation unifying the many existing extensions to feature modeling; on the other hand, Clafer limits feature modeling to its original FODA scope [15], but integrates it into class modeling. Finally, Forfamel considers a constraint language as out of scope, hinting at OCL. Clafer's goal is to provide a concise constraint notation.

TVL is a textual feature modeling language [6]. It favors the use of explicit keywords, which some software developers may prefer. The language covers Boolean features and features of other primitive types such as integer or string. The key difference is that Clafer is also a class modeling language with multiple instantiation, references, and inheritance. It would be interesting to provide a translation from TVL to Clafer. The opposite translation is likely impossible.

Nivel is a *metamodeling* language, which was applied to define feature and class modeling languages [2]. It supports deep instantiation, enabling concise definitions of languages with class-like instantiation semantics. Clafer's purpose is different: to provide a concise notation for combining feature and class models within a single model. Nivel could be used to define the abstract syntax of Clafer, but it would not be able to naturally support our concise concrete syntax.

Clafer builds on our several previous works, including encoding feature models as UML class models with OCL [11]; a Clafer-like graphical profile for ecore, having a bidirectional translation between an annotated ecore model and its rendering in the graphical syntax [19]; and the Clafer-like notation used to specify framework-specific modeling languages [1]. None of these works provided a proper language definition and implementation like Clafer; also, they lacked Clafer's concise constraint notation.

Gheyi et al. [12] pioneered translating feature models into Alloy; their translation targets Boolean feature models, which is a small subset of Clafer.

Relating problem-space feature models and solution-space models has a long tradition. For example, feature models have been used to configure model templates before [8]. That work considered model templates as superimposed instances of a metamodel and presence conditions attached to individual elements of the instances; however, the solution in Sect. 4.2 implements model templates as specializations of a metamodel. Such a solution allows us treating the feature model, the metamodel, and the template at the same metalevel, simply as parts of a single Clafer model. As another example, Janota and Botterweck show how to relate feature and architectural models using constraints [14]. Again, our work differs from this work in that our goal is to provide such integration within a single language. Such integration is given in Kumbang [4], which is a language that supports both feature and architectural models, related via constraints. Kumbang models are translated to Weight Constraint Rule Language (WCRL), which has a reasoner supporting model analysis and instantiation. Kumbang provides a rich domain-specific vocabulary, including features, components, interfaces, and ports; however, Clafer's goal is a minimal clean language covering both feature and class modeling, and serving as a platform to derive such domain specific languages, as needed. We would like to explore specializing and extending Clafer into Kumbang via a profiling mechanism.

8 Conclusion

The premise for our work are usage scenarios mixing feature and class models together, such as representing components as classes and their configuration options as feature hierarchies and relating feature models and component models using constraints. Representing both types of models in single languages allows us to use a common infrastructure for model analysis and instantiation.

We take the perspective of integrating feature modeling into class modeling, rather than trying to extend feature modeling as previously done in its cardinality-based variant. We propose the concept of a class modeling language

with first-class support for feature modeling and define a set of design goals for such languages. Clafer is an example of such a language, and we demonstrate that it satisfies these goals. The design of Clafer revealed that a class modeling language can provide a concise notation for feature modeling if it supports containment feature definitions, feature nesting, group cardinalities, and constraints with default quantifiers. Our design contributes a precise characterization of the relationship between feature and class modeling and a uniform framework to reason about both feature and class models.

References

1. Antkiewicz, M., Czarnecki, K., Stephan, M.: Engineering of framework-specific modeling languages. IEEE TSE 35(6), 795–824 (2009)
2. Asikainen, T., Männistö, T.: Nivel: a metamodelling language with a formal semantics. Software and Systems Modeling 8(4), 521–549 (2009)
3. Asikainen, T., Männistö, T., Soininen, T.: A unified conceptual foundation for feature modelling. In: SPLC'06. pp. 31–40 (2006)
4. Asikainen, T., Männistö, T., Soininen, T.: Kumbang: A domain ontology for modelling variability in software product families. Adv. Eng. Inform. 21(1), 23–40 (2007)
5. Bart Veer, J.D.: The eCos Component Writer's Guide (2000)
6. Boucher, Q., Classen, A., Faber, P., Heymans, P.: Introducing TVL, a text-based feature modelling language. In: VaMoS'10. pp. 159–162 (2010)
7. Clauß, M., Jena, I.: Modeling variability with UML. In: YRW at GCSE'01 (2001)
8. Czarnecki, K., Antkiewicz, M.: Mapping features to models: A template approach based on superimposed variants. In: GPCE'05. pp. 422–437 (2005)
9. Czarnecki, K., Bednasch, T., Unger, P., Eisenecker, U.: Generative programming for embedded software: An industrial experience report. In: GPCE'02. pp. 156–172 (2002)
10. Czarnecki, K., Helsen, S., Eisenecker, U.: Formalizing cardinality-based feature models and their specialization. Software Process: Improvement and Practice 10(1), 7–29 (2005)
11. Czarnecki, K., Kim, C.H.: Cardinality-based feature modeling and constraints: A progress report. In: OOPSLA'05 Workshop on Software Factories (2005)
12. Gheyi, R., Massoni, T., Borba, P.: A theory for feature models in Alloy. In: First Alloy Workshop. pp. 71–80 (2006)
13. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press (2006)
14. Janota, M., Botterweck, G.: Formal approach to integrating feature and architecture models. In: FASE'08. pp. 31–45 (2008)
15. Kang, K., Cohen, S., Hess, J., Nowak, W., Peterson, S.: Feature-oriented domain analysis (FODA) feasibility study. Tech. Rep. CMU/SEI-90-TR-21 (1990)
16. Kang, K.C.: FODA: Twenty years of perspective on feature modeling. In: VaMoS'10 (Keynote) (2010)
17. Le Berre, D., Rapicault, P.: Dependency management for the Eclipse ecosystem: Eclipse p2, metadata and resolution. In: IWOCE'09. pp. 21–30 (2009)
18. She, S., Lotufo, R., Berger, T., Wasowski, A., Czarnecki, K.: Variability model of the linux kernel. In: VaMoS'10. pp. 45–51 (2010)
19. Stephan, M., Antkiewicz, M.: Ecore.fmp: A tool for editing and instantiating class models as feature models. Tech. Rep. 2008-08, Univeristy of Waterloo (2008)