Clafer: a Unified Language for Class and Feature Modeling

Kacper Bąk
Generative Software Development Lab
University of Waterloo
Canada
kbak@gsd.uwaterloo.ca

ABSTRACT

This paper presents Clafer, a class modeling language with first class support for feature modeling. In the work we identify key differences between class and feature models and show how to unify the two notations. Our language offers simple, yet powerful constraint notation to restrict models and define mappings between features and classes. In the paper, we describe how to use Clafer to express problem and solution spaces and specify mappings between them. We also present a Clafer-to-Alloy translator, our primary tool that gives precise semantics to Clafer.

Categories and Subject Descriptors

D.3 [**Software**]: Programming Languages; D.3.2 [**Programming Languages**]: Language Classifications—*Design languages*

General Terms

Design, Languages

Keywords

class modeling, feature modeling, constraints

1. INTRODUCTION

Feature modeling is a technique used for capturing commonalities and variabilities in software product lines. It was introduced by Kang *et al.* as a part of *Feature-Oriented Domain Analysis* methodology [12]. Original FODA feature models, are tree structures supplemented with additional constraints to specify dependencies among features.

The primary purpose of feature models is to describe variation space in terms of *user-relevant* products characteristics. Feature models naturally fit into the *problem space* (see Fig. 1), as they determine what products are available in the particular software product line. The *solution space*, on the other hand, is usually expressed in terms of class models. It defines components, connectors and additional constraints that make up product line architectures.

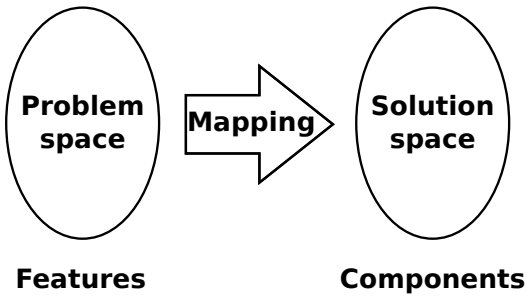

Figure 1: Problem and solution spaces

The two modeling notations were designed with different purposes in mind and model different types of variability. Feature models capture selections from predefined fixed (tree) structure, thus we obtain a subset of available features. This is not usually the case with class models, as they grow and become more complex. Class models support making new structures by inheritance, creating multiple instances of classes and connecting them via object references.

There have been attempts to merge feature with class modeling. It was achieved either by extending feature models so that they become class models or by clumsily simulating containment hierarchy in class models. Unfortunately, both approaches tend to make feature modeling notation more complex and less intuitive. Our belief that simplicity of the original FODA notation is an important advantage has been confirmed in a recent study [13].

In this paper, we present *Clafer* (class, feature, reference), a class modeling language with first-class support for feature modeling. Our textual language provides a uniform syntax and semantics to class and feature models. It tries to minimize the number of underlying concepts. Both feature and class models can be naturally mixed and coupled via constraints and inheritance.

Our recent work on Clafer focused on polishing existing constructions, adding Alloy-like constraints, providing name resolution strategy and developing a Clafer-to-Alloy translator. Thus, the language semantics is mostly defined in terms of Alloy constructions, but complete specification includes rules of semantic analysis, which are part of the translator.

The rest of the paper is organized as follows. We relate class to feature modeling in Sect. 2. We show how both

modeling notations are unified in Clafer in Sect. 3. We then describe the structure of our translator and function of each component in Sect. 4. We argue that Clafer satisfies its design goals in Sect. 5. We outline future directions for Clafer design in Sect. 6, compare the language with existing solutions in Sect. 7 and conclude in Sect. 8.

2. UNIFYING CLASS AND FEATURE MODELING

Class and feature modeling capture different abstractions. The former gives a good high-level picture of the application's functionality, while the latter is a preferable notation for representing implementational models.

The two complimentary formalisms cannot be easily combined in the existing class modeling notations, such as UML and Alloy, since they do not provide first-class support for feature modeling. It is possible to express feature models in terms of class models by using composition, inner classes and OCL constraints, but the result carries undesirable notational complexity.

In contrast to class diagrams, there is no one standard notation for feature models. The whole variety of notations is discussed in the literature [11]. Some of them are tree-like structures, where each non-root feature has one parent. Other notations do not impose this restriction and allow more general Directed Acyclic Graphs to form feature models. We believe that further extensions, such as cardinality-based feature diagrams [8] and attributes [5] bring feature modeling much closer to class modeling than to the original FODA modeling notation.

The two modeling notations can be elegantly unified by providing 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 with overlapping constructions.

3. CLAFER: CLASS, FEATURE, REFERENCE

We introduce Clafer by presenting a running example: simple 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 2 presents a variability model of a sample telematics product line.

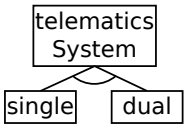

Figure 2: Problem space: product features

3.1 Feature Modeling

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 telematicsSystem. The corresponding Clafer model is presented in Fig. 3.

```
1 xor telematicsSystem
2   single
3   dual
```

Figure 3: Feature model in Clafer

In general, 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. The enclosing type provides a separate name space for this content (so it is similar to nesting inner classes in UML class diagrams).

A type definition can contain one or more *features*. Features are slots that can contain one or more instances or references to instances. Mathematically, features are binary relations. Original FODA feature models incorporate only *containment features*, i.e., features that contain instances. All features visible in Fig. 3 are of this type. An instance can be contained by only one feature, and no cycles in instance containment are allowed. The parent-children hierarchy is indicated by simply indenting subfeatures under parent feature.

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 single (line 2) in Fig. 3 defines both the feature single, and, implicitly, the type single. The new type is nested in the type telematicsSystem.

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 $m \in \mathbb{N}, n \in \mathbb{N} \cup \{*\}, m \leq n$. As conciseness is an important goal for Clafer, we provide syntactic sugar for common constructions. For example, feature cardinalities resemble syntax of regular expressions: `?` (optional) denote $0..1$; `*` denote $0..*$; and `+` denote $1..*$. Feature cardinality specification follows the feature name or its reference type, if any. No fature cardinality specified denotes $1..1$ (mandatory) by default (e.g. telematicsSystem in Fig. 3), modulo three exceptions explained next.

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

⟨1–∗⟩; and `mux` denotes ⟨0–1⟩; 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 telematicsSystem (line 1) in Fig. 3 states that only one child instance of either single or dual is allowed. Because the two subfeatures single and dual have no explicit cardinality attached to them, they are both optional. No explicit group cardinality stands for ⟨0–∗⟩, except when it is inherited as illustrated later.

3.2 Class Modeling

So far we expressed our problem space in Clafer. Let us move on to the solution space. Figure 4 shows components of a generic 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.

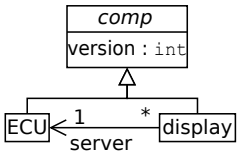

Figure 4: Solution space: component model

The corresponding Clafer representation is visible in Fig. 5. Again, the model is a set of type definitions, features and constraints. The `abstract` modifier indicates that no instance of the type will be created, unless extended by a concrete type. Now features correspond to attributes or role names of association or composition relationships in UML.

```
1  abstract comp
2    version : integer
3
4  abstract ECU extends comp
5
6  abstract display extends comp
7    server : ECU
8    [version >= server.version]
```

Figure 5: Class model in Clafer

For example, in Fig. 5, the feature version (line 2) corresponds to the attribute of the class comp in Fig. 4; and the feature server (line 7) corresponds to the association role name next to the class ECU in Fig. 4. Features declared using the colon notation and having no subfeatures, like in server : ECU, are *reference features*, i.e., they hold references to instances. The reference feature server points to an existing ECU instance.

Constraints are a significant aspect of Clafer, as they 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 set expressions. The value of a set 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 before Boolean expression stands for `some`, meaning that the relation cannot be empty (presented later).

The constraint defined in the context of display states that display's version cannot be lower than server's version. Although version is itself just a reference, Clafer is instructed to compare the actual integer values and not just references.

Each feature in Clafer introduces a local namespace, which is rather different from namespaces in popular programming languages. Name resolution is required to identify: target of a reference feature, and features used in constraints. In both cases, names are path expressions (similar to navigation 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 a feature and a constraint.

3.3 Specializing and Extending the Class Model

The class model presented in Fig. 5 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 6 shows such a template for our example. We specialize the generic metamodel via inheritance and constraints. In our example, a concrete product must have at least one ECU (ECU1) and can optionally (indicated by a question mark in line 7) have another ECU (ECU2). The two ECU's extend the plaECU stating that each ECU contains a display. The *quotation* notation (line 2 in Fig. 6) is a syntactic sugar for inheritance.

Syntactically, quotation is just a name of abstract type preceded by left quote (‘), which in the example is expanded as display `extends` display. The first name indicates a new feature, and the second refers to the abstract type. Semantically, this notation creates a containment feature display with a new concrete type plaECU.display, which extends the top-level abstract type display from Fig. 6. The concrete type inherits group cardinality and features of its supertype.

Besides, we need to constrain the server reference in a display, so that it points to its associated ECU. The reference this points to the current instance of plaECU. Also, ECU2 extends the base type with master, pointing to ECU1 as the main control unit.

3.4 Gluing Class and Feature Models

We already defined our high-level representation of a product line and a template architectural model. Mapping be-

```
1  abstract plaECU extends ECU
2    'display
3    [display.server = this]
4
5  ECU1 extends plaECU
6
7  ECU2 extends plaECU ?
8    master : ECU1
```

Figure 6: Architectural template

tween the two models is still missing. A correct configuration shall always have at least one concrete ECU (ECU1) and optionally ECU2. This mapping is provided by adding a constraint to the original feature model (Fig. 3) resulting in a constrained feature model (Fig. 7). The constraint says that the dual feature is present if and only if ECU2 is present. Thus, we defined a mapping from the problem space to the the solution space.

```
1  xor telematicsSystem
2    single
3    dual
4    [dual <=> ECU2]
```

Figure 7: Feature model with mapping constraint

Constraints allow us restricting a model to a single or dual configuration. Figure 8 shows a top-level constraint defining a single product, with two ECUs and all components in version 1. Such a global constraint removes all variability from the feature and class model.

```
1  -- concrete product
2  [dual && comp.version == 1]
```

Figure 8: Constraints determining a single product

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 more than 7 times). Although Alloy can detect some equivalent solutions, it does not perform full graph isomorphisms detection.

4. CLAFER-TO-ALLOY TRANSLATOR

Clafer is designed simultaneously with the clafer2alloy translator. It takes a Clafer model and transforms it to corresponding Alloy model. The translation gives precise semantics to our language by performing semantic analysis and establishing a mapping to Alloy. The translator enables us to experiment with new language constructions, detect cross-cutting concerns and dependencies.

The software is written in Haskell and comprises several chained modules: lexer, layout resolver, parser, desugarer, semantic analyzer, and code generator. Lexer and parser were generated from a labeled BNF grammar by BNFC [14], which is a multilingual front-end generator. The rest of the code was hand-coded in a purely functional style with occasional use of monads [17].

Translation is not a straightforward process because of possible dependencies among features and substantial amount of syntactic sugar offered by Clafer. We present our software step-by-step by describing each module's purpose and operation. For the sake of clarity, the input Clafer model from Sect. 3 is summarized in Fig. 9.

```
1  abstract comp
2    version : integer
3
4  abstract ECU extends comp
5
6  abstract display extends comp
7    server : ECU
8    [version >= server.version]
9
10 abstract plaECU extends ECU
11   'display
12   [display.server = this]
13
14 ECU1 extends plaECU
15
16 ECU2 extends plaECU ?
17   master : ECU1
18
19 xor telematicsSystem
20   single
21   dual
22   [dual <=> ECU2]
23
24 [dual && comp.version == 1]
```

Figure 9: Telematics PLA: Clafer model

4.1 Layout Resolver

Layout resolver makes the use of braces grouping subfeatures optional. The relation parent-child can be expressed simply by indenting subfeatures further. This way of structuring code makes a model shorter, and arguably, easier to read.

The resolver is a fairly simple module that processes a stream of tokens before they reach the parser. It automatically inserts brackets when necessary, so that the output stream complies to parser rules generated from the context-free BNF grammar. Braces should only be placed before and after a collection of subfeatures (e.g. compare lines 1 and 3 in Fig. 9 and Fig. 10).

Clafer is not a typical programming language, thus requires some customization of the popular layout resolution algorithm. For example, we do not want to modify constraints or strings spanning over multiple lines. Another significant difference concerns lack of explicit keywords for indicating a block of code. It makes the analysis slightly more complex since the resolver needs to find out whether a line-break occurred within a feature hierarchy or in other construction (e.g. enumeration).

4.2 Desugarer

Clafer is composed of two languages: the core and the full language. The first one is a minimal language with well-

```
1  abstract comp {
2    version : integer
3  }
4
5  abstract ECU extends comp
6
7  abstract display extends comp {
8    server : ECU
9    [version >= server.version]
10 }
11
12 abstract plaECU extends ECU {
13   'display
14   [display.server = this]
15 }
16
17 ECU1 extends plaECU
18
19 ECU2 extends plaECU ? {
20   master : ECU1
21 }
22
23 xor telematicsSystem {
24   single
25   dual
26   [dual <=> ECU2]
27 }
28
29 [dual && comp.version == 1]
```

Figure 10: Telematics PLA: resolved layout

defined translational semantics. The latter is built on top of the core language and provides large amount of syntactic sugar. Separation of the two layers makes the language not only more elegant, but also simplifies semantic analysis and code generation stages.

Desugarer removes syntactic sugar from the input model. The most straightforward solution would be to implement it as a text file preprocessor. Yet we decided to perform an Abstract Syntax Tree transformation instead. Large part of the code is generated by BNFC so our work limits to constructing the actual tree rewriting rules.

The desugarer works as follows. First, it converts enumerations to a bunch of abstract features (not present in the running example). Then sets implicit clafer supertype for classes/features (e.g. line 1 in Fig. 11). Thanks to one common supertype we can easily perform set operations (e.g. union, relational join) on them.

In the next step attribute navigation is converted to a set expression – this AST transformation is not reflected in the code but simplifies code generation by unifying AST nodes. Sample navigational expressions include target features of reference features (e.g. ECU in line 8 in Fig. 11).

Finally, feature cardinalities are transformed to their numerical values (e.g. lines 2 and 19 in Fig. 11). This transformation does not, however, update cardinalities of abstract feature definitions (e.g. line 1 in Fig. 11) because we do not impose any feature cardinality restrictions on type definitions. If needed, this kind of constraint could be expressed by additional constraints.

The transition from input code to desugared code does not seem to be a big change. On the other hand, the role of desugarer will certainly grow if Clafer is used as a base language for more customized notations. In that case, a domain-specific notation shall be provided and desugared by external tools.

```
1  abstract comp extends clafer {
2    version extends clafer : integer 1..1 {}
3  }
4
5  abstract ECU extends comp {}
6
7  abstract display extends comp {
8    server extends clafer : ECU 1..1 {}
9    [version >= server.version]
10 }
11
12 abstract plaECU extends ECU {
13   display extends display 1..1 {}
14   [display.server = this]
15 }
16
17 ECU1 extends plaECU 1..1 {}
18
19 ECU2 extends plaECU 0..1 {
20   master extends clafer : ECU1 1..1 {}
21 }
22
23 xor telematicsSystem extends clafer 1..1 {
24   single extends clafer 0..1 {}
25   dual extends clafer 0..1 {}
26   [dual <=> ECU2]
27 }
28 [dual && comp.version == 1]
```

Figure 11: Telematics PLA: desugared

4.3 Semantic Analyzer

There are certain semantical differences between Clafer and Alloy. They mostly come from properties of feature models. For example, Alloy does not allow signature nesting and thus all names shall be resolved by the analyzer to avoid name clashes. Furthermore, in Clafer it is possible to overwrite inherited group cardinality. It is especially useful when we want to extend a feature and constrain the number of all subfeatures. In Alloy such an operation is impossible because we cannot ignore constraints of the supertype.

For the above reasons significant part of translation is devoted to semantic analysis. This stage distinguishes our translation from direct mapping of metamodels. Similarly to the desugarer, semantic analysis is an AST rewriting process. However, to transform an AST node we need a context, i.e. the transformation depends on other nodes and could not be easily performed by a preprocessor.

Semantic analysis starts with determining feature's inherited artifacts: group cardinality and attribute (reference). Group cardinality is inherited by default (e.g. line 5 in Fig. 12) unless specified explicitly (e.g. line 23 in Fig. 12, where xor is expanded as $\langle 1{-}1 \rangle$).

An attribute is also inherited from the supertype if not specified otherwise. Explicit specification of an attribute never overwrites the inherited one. It can only further restrict

the type a reference points to. This design decision goes along with Liskov substitution principle. We should note that if attribute types are disjoint then the attribute yields an empty set.

Each feature introduces a new namespace in Clafer model. In contrast, Alloy namespace within a file is mostly flat. To deal with this issue the analyzer keeps track of Clafer names and determines absolute path in set expressions. For example, the dual name (line 28 in Fig. 11) is resolved as @telematicsSystem.@dual (line 28 in Fig. 12). Each name preceded by @ indicates that the feature will no longer be resolved.

Clafer constraints are not limited to Boolean expressions. They include relational operators to compare string and integer values. Internally these primitive types are features extending the clafer supertype. By wrapping primitive types we achieve more uniformity among language concepts. Features containing primitive types differ from other features in the sense that they contain the actual value field of primitive type. The value field is attached by the analyzer whenever value-comparing operators are used (e.g. in line 9 in Fig. 12 @val is attached).

A quick look at the initial model in Fig. 9 and the model in Fig. 12 reveals verbosity of the second representation. It comes mainly from assuming certain *defaults* about feature/class modeling constructions. In our opinion assuming intuitive defaults improves usability and clarity of the language.

Semantic analyzer worked fairly well for most of the models we tested. Nevertheless, the name resolution algorithm still partly relies on Alloy name-lookup strategy. In some cases it results in name clash and errors from the Allow Analyzer. We acknowledge that more precise semantics for Clafer name resolution is needed.

4.4 Code Generator

The code generator transforms the core language into Alloy. The input Clafer model is assumed to have fully resolved dependencies and expanded convenient syntactic constructions. The generator traverses input Abstract Syntax Tree and for each node creates corresponding string with Alloy code. The output code (in Fig. 13) is much more verbose than the original Clafer model (in Fig. 9). Our rough approximations show that Alloy models are 2-3 times longer in terms of lines of code, and about four times longer in terms of number of characters. Alloy code is necessarily longer despite of certain optimizations performed by the translator.

Let us now go through the Alloy model in Fig. 13. The translated file starts with a standard header. The two first lines are required to instantiate an Alloy model (we do not check any property here). The line 4 defines a supertype of a feature, and declares that each feature has a parent. Then we create a feature holding integer value and for the sake of convenience specify that it is its own parent. Line 7 and 8 contain a function that for each class/feature returns their all subfeatures.

The next part contains the translated model. Features in Fig. 12 are in different order than corresponding Alloy sig-

```
1  abstract <0-*> comp extends clafer {
2    <0-*> version extends clafer : @integer 1..1 {}
3  }
4
5  abstract <0-*> ECU extends comp {}
6
7  abstract <0-*> display extends comp {
8    <0-*> server extends clafer : @ECU 1..1 {}
9    [this.@version.@val >= this.@server.@version.@val]
10 }
11
12 abstract <0-*> plaECU extends ECU {
13   <0-*> display extends display 1..1 {}
14   [this.@display.@server = this]
15 }
16
17 <0-*> ECU1 extends plaECU 1..1 {}
18
19 <0-*> ECU2 extends plaECU 0..1 {
20   <0-*> master extends clafer : @ECU1 1..1 {}
21 }
22
23 <1-1> telematicsSystem extends clafer 1..1 {
24   <0-*> single extends clafer 0..1 {}
25   <0-*> dual extends clafer 0..1 {}
26   [this.@dual <=> @ECU2]
27 }
28 [@telematicsSystem.@dual && @comp.@version.@val == 1]
```

Figure 12: Telematics PLA: core Clafer model

natures in Fig. 13. In our description we follow the Alloy file. Lines 10–14 in Fig. 13 correspond to lines 7–10 in Fig. 12. An abstract feature is translated to an abstract signature. Supertype is the same in both models. Next we place the server subfeature as a field. The field points to ECU as specified in the Alloy constraint. Finally the constraint from line 9 in Fig. 12 is almost the same in the Alloy model. Please note that in abstract signatures we never specify the parent feature, since they are just type definitions and indeed do not have a well-defined parent.

Non-abstract features are translated in a very similar fashion. However, their signatures are not preceded by the abstract keyword, and the parent is stated explicitly in attached constraints (e.g line 47 in Fig. 13).

Group constraints are translated as proper Alloy constraints (e.g. the one from line 23 in Fig. 12 is written in line 43 in Fig. 13). They are reasonably short thanks to the children function. However, it does not deal with reference features, and if we were to include them in the subclafers set, we would have to enumerate them explicitly.

Clafer allows to use Boolean constraints even though the underlying semantics is relational logic (compare line 28 in Fig. 12 and lines 51–52 in Fig. 13). This is done by assuming the default some quantifier before a logical term.

The last lines of Alloy model specify feature cardinalities of top-level features (ECU1, ECU2 and telematicsSystem). They are placed outside the rest of the model, because Alloy model cannot contain relational definitions outside signatures.

4.5 Operation

```
pred show {}
run  show for 7

abstract sig clafer {parent : one clafer}
one sig integer extends clafer {val : Int}
{parent = this}
fun children(p : clafer) : set clafer{
  {c : clafer | c != p && c.@parent = p}}

abstract sig display extends comp
{ server : one clafer }
{ server in @ECU
  (((this).(@version)).(@val)) >=
  ((((this).(@server)).(@version)).(@val)) }
sig absdisplay extends display {}
{ no absdisplay }
abstract sig plaECU extends ECU
{ display : one plaECU_display }
{ (((this).(@display)).(@server)) = (this) }
sig absplaECU extends plaECU {} { no absplaECU }
sig plaECU_display extends display
{}
{ parent = (plaECU <: display).this }
abstract sig ECU extends comp
{}
{}
sig absECU extends ECU {} { no absECU }
abstract sig comp extends clafer
{ version : one clafer }
{ version in @integer }
sig abscomp extends comp {} { no abscomp }
sig ECU1 extends plaECU
{}
{ parent = this }
sig ECU2 extends plaECU
{ master : one clafer }
{ parent = this
  master in @ECU1 }
sig telematicsSystem extends clafer
{ single : lone telematicsSystem_single
, dual : lone telematicsSystem_dual }
{ parent = this
  let subclafers = this.children | one subclafers
  (some (this).(@dual)) <=> (some @ECU2) }
sig telematicsSystem_single extends clafer
{}
{ parent = (telematicsSystem <: single).this }
sig telematicsSystem_dual extends clafer
{}
{ parent = (telematicsSystem <: dual).this }
fact { (some (@telematicsSystem).(@dual)) &&
        ((((@comp).(@version)).(@val)) = (1)) }
fact {  one ECU1 }
fact {  lone ECU2 }
fact {  one telematicsSystem }
```

Figure 13: Telematics PLA: Alloy model

The clafer2alloy translator is available as a source code and a binary. The code can be run in a Haskell interpreter (such as GHCi) by loading the clafer2alloy.hs file and invoking the runFile 2 pModule function with specified Clafer model file.

The binary is run by the command:

clafer2alloy sourceFile.cfr

where sourceFile.cfr is a Clafer model. The translator generates three output files: sourceFile.des, sourceFile.ana and sourceFile.als. The first one contains desugared Clafer model, the second one the same model after semantic analysis and the third one the Alloy model.

The Alloy model can then be loaded into the Alloy Analyzer and checked/instantiated. As each analysis requires specifying the number of instances of each signature, it can be updated in the second line of the als file.

5. DISCUSSION

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. 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 (by using inner classes). Alloy has no built-in support for composition and thus requires explicit set-up of parent-child constraints. It also has no signature nesting.
- *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`, allow us writing constraints that look like propositional logic, even though their underlying semantics is first-order predicate logic.

Clafer tries to use a minimal number of concepts and has uniform semantics. In principle, a Clafer model is a set of relations. 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.

6. 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.

There are several directions that we would like to explore in the nearest future. Clafer requires a lot of work as a language to become more usable. For example, the language cannot handle strings although they are used in practical variability modeling languages, such as KConfig [15] or CDL [4].

We would like to support programmers with better type checking/inference mechanisms. This is especially useful if default values (integers, strings) can be specified by the programmer. Type inference would eliminate explicit enumeration of type when it can be inferred from the value.

Besides, to improve scalability and reusability we plan on adding a modularization mechanism. We expect it to be slightly different than in popular languages, since we observed that a Clafer library would consist mostly of abstract features instead of instances. Thus intuitive defaults are to be designed.

Another direction concerns chaining Clafer translator with related tools. Therefore, 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.

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 [12], 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 [7]. It favors the use of explicit keywords, which some software developers may prefer. The language covers Boolean features and features of other types such as integer or enumerations. 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 [9]; a Clafer-like graphical profile for ecore, having a bidirectional translation between an annotated ecore model and its rendering in the graphical syntax [16]; 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. [10] pioneered translating feature models into Alloy; their translation targets Boolean feature models, which is a small subset of Clafer.

8. CONCLUSIONS

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.

9. ACKNOWLEDGMENTS

This paper describes our work on Clafer carried out in the Generative Software Development Lab. The research was done with support of Krzysztof Czarnecki and Andrzej Wasowski. Some parts of the text come from our technical report [6].

10. REFERENCES

[1] M. Antkiewicz, K. Czarnecki, and M. Stephan. Engineering of framework-specific modeling languages. *IEEE TSE*, 35(6):795–824, 2009.

[2] T. Asikainen and T. Männistö. Nivel: a metamodelling language with a formal semantics. *Software and Systems Modeling*, 8(4):521–549, 2009.

[3] T. Asikainen, T. Männistö, and T. Soininen. A unified conceptual foundation for feature modelling. In *SPLC'06*, pages 31–40, 2006.

[4] J. D. Bart Veer. *The eCos Component Writer's Guide*. 2000.

[5] D. Benavides, P. Trinidad, and A. Ruiz-Cortés. Automated reasoning on feature models. In *CAISE 2005*, pages 491–503, 2005.

[6] K. Bąk, K. Czarnecki, and A. Wąsowski. Feature and Class Models in Clafer: Mixed, Specialized, and Coupled. Technical Report CS-2010-10, David R. Cheriton School of Computer Science, University of Waterloo, 2010.

[7] Q. Boucher, A. Classen, P. Faber, and P. Heymans. Introducing TVL, a text-based feature modelling language. In *VaMoS'10*, pages 159–162, 2010.

[8] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged configuration through specialization and multi-level configuration of feature models. *Software Process Improvement and Practice*, 10(2):143–169, 2005.

[9] K. Czarnecki and C. H. Kim. Cardinality-based feature modeling and constraints: A progress report. In *OOPSLA'05 Workshop on Software Factories*, 2005.

[10] R. Gheyi, T. Massoni, and P. Borba. A theory for feature models in Alloy. In *First Alloy Workshop*, pages 71–80, 2006.

[11] P. Heymans, P. Y. Schobbens, J. C. Trigaux, Y. Bontemps, R. Matulevicius, and A. Classen. Evaluating formal properties of feature diagram languages. *Software, IET*, 2(3):281–302, 2008.

[12] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Carnegie Mellon University, 1990.

[13] K. C. Kang. FODA: Twenty years of perspective on feature modeling. In *VaMoS'10 (Keynote)*, 2010.

[14] M. Pellauer, M. Forsberg, and A. Ranta. Bnf converter multilingual front-end generation from labelled bnf grammars. Technical Report 2004-09, Chalmers University of Technology and Göteborg University, 2004.

[15] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki. Variability model of the linux kernel. In *VaMoS'10*, pages 45–51, 2010.

[16] M. Stephan and M. Antkiewicz. Ecore.fmp: A tool for editing and instantiating class models as feature models. Technical Report 2008-08, Univeristy of Waterloo, 2008.

[17] P. Wadler. Monads for functional programming. In *Advanced Functional Programming*, pages 24–52, 1995.