Feature and Meta-Models in Clafer: Mixed, Specialized, and Coupled

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 meta-modeling language with first-class support for feature modeling. We designed Clafer as a concise notation for meta-models, feature models, mixtures of meta- and feature models (such as components with options), and models that couple feature models and meta-models via constraints (such as mapping feature configurations to component configurations or model templates). Clafer also allows arranging models into multiple specialization and extension layers via constraints and inheritance. We identify four key mechanisms allowing a meta-modeling language to express feature models concisely and show that Clafer meets its design objectives using a sample product line. We evaluated Clafer and how it lends itself to analysis on sample feature models, meta-models, and model templates of an E-Commerce platform.

1 Introduction

Both feature and meta-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 [22]. These models are typically used to show the variation of *user-relevant* characteristics of products within a product line. In contrast, meta-models, as supported by the Meta Object Facility (MOF) [28], represent concepts of—possibly domain-specific—modeling languages, used to represent more detailed aspects such as behavioral or architectural specification. For example, meta-models are often used to represent the components and connectors of *product line architectures* and the valid ways to connect them. 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 meta-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 models and meta-models (represented as class models) has been blurred somewhat in the literature due to 1) feature modeling extensions, such as *cardinality-based feature modeling* [15, 4], or 2) attempts to express feature models as class models in Unified Modeling Language (UML) [11, 16]; note that MOF is essentially the class modeling subset of UML. A key driver behind these developments has been the

desire to express components and configuration options in a single notation [14]. Cardinality-based feature modeling achieves this by extending feature models with multiple instantiation and references. Class modeling, which natively supports multiple instantiation and references, enables feature modeling by a stylized use of composition and the profiling mechanisms of MOF or UML.

Both developments have notable drawbacks, however. An important advantage of feature modeling as originally defined by Kang et al. [22] is its simplicity; several respondents to a recent survey confirmed this view [23]. 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 meta-modeling language with first-class support for feature modeling. The language was designed to naturally express meta-models, feature models, mixtures of meta- and feature models (such as components with options), and models that couple feature models with meta-models and their instances via constraints (such as mapping feature configurations to component configurations or to *model templates* [13]). Clafer also allows arranging models into multiple specialization and extension layers via constraints and inheritance, which we illustrate using a sample product line.

We developed a translator from Clafer to Alloy [19], a class modeling language with a modern constraint notation. The translator gives Clafer precise translational semantics and enables model analyses using Alloy Analyzer. Different strategies are applied for distinct model classes. They all preserve meaning of the models, but speed up analysis by exploiting the Alloy constructions.

We evaluate Clafer analytically and experimentally. The analytic evaluation argues that Clafer meets its design objectives. It identifies four key mechanisms allowing a meta-modeling language to express feature models concisely. The experimental evaluation shows that a wide range of realistic feature models, meta-models, and model templates can be expressed in Clafer and that useful analyses can be run on them within seconds. Many useful analyses such as consistency checks, element liveness, configuration completion, and reasoning on model edits can be reduced to instance finding by combinatorial solvers [7, 9, 12]; thus, we use instance finding and element liveness as representatives of such analyses.

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 evaluate the language analytically and experimentally in Sect. 5. We conclude in Sect. 7, after having compared Clafer with related work in Sect. 6.

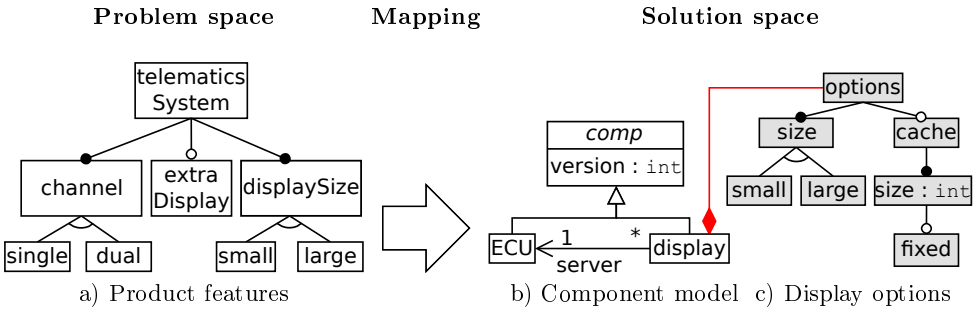


Fig. 1. Telematics product line

2 Running Example: A 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 presents a variability model of a sample telematics product line, which we will use as a running example. The features offered are summarized in the *problem-space* feature model (Fig. 1a). A concrete telematics system can support either a single or two channels; two channels afford independent programming for the driver and the passengers. The choice is represented as the xor-group channel, marked by the arch between edges. 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).

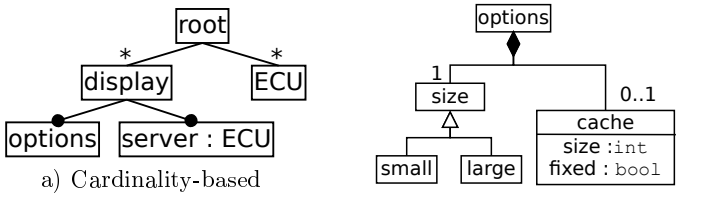
Figure 1b shows a meta-model of components making up a telematics system. There are two types of components: ECUs (electronic control units) and displays. Each display has exactly one ECU as its server. All components have a version.

Components themselves may have options, like the display size or cache (Fig. 1c). We can also specify the cache size and decide whether it is fixed or can be updated dynamically. Thus, the *solution space* model consists of a class model of component types and a feature model of component options.

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

3 Feature vs. Meta-Modeling

The solution space in Fig. 1 contains a meta- 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.



a) Cardinality-based feature model of components

b) Meta-model of display options

Fig. 2. Feature model as meta-model and vice versa

We have at least two choices to represent components and options in a single notation. The first is to show the entire solution space model using cardinality-based feature modeling [15]. Figure 2a shows the component 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 meta-model in Fig. 1b or we could extend the notation with inheritance. The latter would bring the cardinality-based feature modeling notation very close to meta-modeling based on class modeling, posing the question whether class modeling should not 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, it does have several drawbacks. First, the feature model showed fixed as a property of size by nesting; this intention is lost in the class model. A solution would be to create a separate class size, containing the size value and a class fixed; thus, adding a subfeature to a feature represented as a class attribute requires refactoring. 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, or use nested classes, which further complicates the model. Moreover, converting an xor-group to an or-group in feature modeling is simple: the empty arch 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 incurs additional complexity.

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

(1) *"Cardinality-based feature modeling" is a misnomer.* It 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 result can hardly be called 'feature modeling', as it clearly goes beyond the original scope of feature modeling [22].
(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-based) meta-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 meta-modeling*
3. *Allow mixing feature models and meta-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: Meta-Modeling with First-Class Support for Feature Modeling

We explain the meaning of Clafer models by relating them to their corresponding UML class models.[3] Figure 3 shows the display options feature model in Clafer (a) and the the corresponding UML model (c). Figure 4 shows the component meta-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 prohibits creating an instance of the type, 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 arrow notation and having no subfeatures, like in server -> ECU, are *reference features*, i.e., they hold references to instances. Note that we model integral features, like version

[3] For more precise documentation including meta-models see gsd.uwaterloo.ca/sle2010

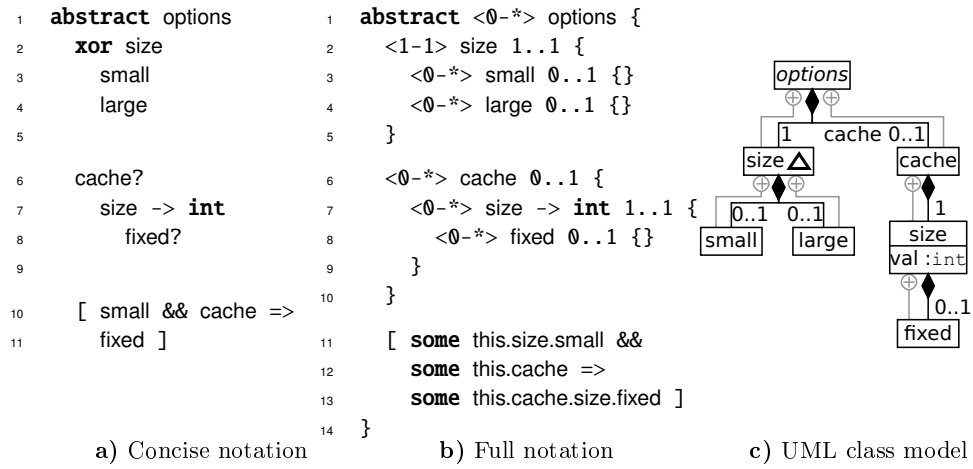


Fig. 3. Feature model in Clafer and corresponding UML class model

(line 2) in Fig. 4, as references. Clafer has only one object representing a given number, which speeds up automated analyses.

Features that do not have their type declared using the arrow 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.

```
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

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.

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 four 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 $\langle 1{-}* \rangle$; `opt` denotes $\langle 0{-}* \rangle$; 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. 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), signifying 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 its 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.

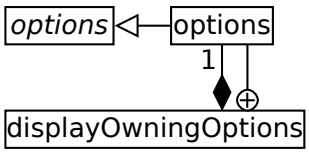
4.1 Mixing via Quotes and References

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 the quoted type is shared, but no instances. References, on the other hand, are used for sharing instances.

The following example highlights the difference:

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

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

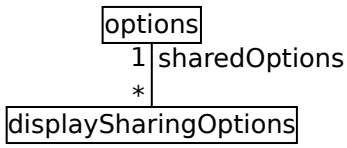


In the above snippet, each instance of displayOwningOptions will have its own instance of type options, as depicted in the corresponding UML diagram. Other types could also quote options to reuse it. Note that Clafer assumes the existence of an implicit *root object*; thus, a feature definition, such as displayOwningOptions above, defines both a subfeature of the root object and a new top-level concrete type.

Now consider the following code with corresponding UML diagram:

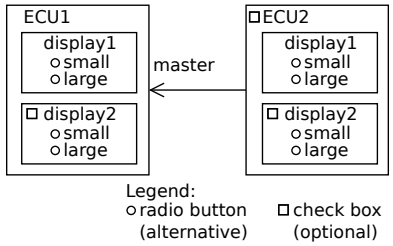
```
options *
  -- content as in options in Fig. 3a

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.

```
1 abstract plaECU extends ECU
2   'display 1..2
3     [ ~cache
4       server = parent ]
5 ECU1 extends plaECU
6 ECU2 extends plaECU ?
7   master -> ECU1
```



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

Fig. 5. Architectural template

4.2 Specializing via Inheritance and Constraints

Let us go back to our telematics product line example. The architectural meta-model as presented in Fig. 4 is very generic: the meta-model describes infinitely many different products, each corresponding to its particular instance. We would like to *specialize* and *extend* the meta-model to create a particular *template*. A template makes most of the architectural structure fixed, but leaves some points of variability. In previous work, we introduced *feature-based model templates* (FBMT in short) as models (instances of meta-models) with optional elements annotated with Boolean expressions over features known as *presence conditions* [13]. Below, we show how such templates can be expressed in Clafer.

Figure 5a shows such a template for our example. We achieve specialization via inheritance and constraints. In particular, we represent instances of meta-model classes as singleton classes. In our example, a concrete product must have at least one ECU and thus we create ECU1 to represent the mandatory instance. Then, optional instances are represented using classes with cardinality 0..1. Our product line can optionally have another ECU, represented by ECU2. Similarly, each ECU has either one display or two displays, but none of the displays has cache. Besides, we need to constrain the server reference in each display in plaECU, so that it points to its associated ECU. The constraint in line 3 in Fig. 5a is nested under display. The reference parent points to the current instance of plaECU, which is either ECU1 or ECU2. Also, ECU2 *extends* the base type with master, pointing to ECU1 as the main control unit.

Figure 5b visualizes the template in a domain-specific notation, showing both the fixed parts, e.g., mandatory ECU1 and display1, and the variable parts, e.g., alternative display sizes (radio buttons) and optional ECU2 and display2 (check-boxes). Note that model templates such as UML models annotated with presence conditions (e.g., [13]) can be translated into Clafer automatically by 1) representing each model element e by a class with cardinality 0..1 that extends the element's meta-class and 2) a constraint of the form p && c <=> e, with p being e's parent and c being e's presence condition. In our example, we keep these constraints separate from the template (see Sect. 4.3). Further, in contrast to

```
telematicsSystem
  xor channel
    single
    dual
  extraDisplay?
  xor displaySize
    small
    large

[ dual <=> ECU2
  extraDisplay <=> #ECU1.display = 2
  extraDisplay <=>
      (ECU2 <=> #ECU2.display = 2)
  small <=> ~plaECU.display.options.size.large
  large <=> ~plaECU.display.options.size.small
]
```

Fig. 6. Feature model with mapping constraints

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

Fig. 7. Constraint specifying a single product

annotating models with presence conditions, we can use subclassing and constraints to specialize and extend the meta-model in multiple layers.

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 6 shows this model (cf. Fig. 1a) along with a set of constraints coupling its features to the variability points of the template. Note that the template allowed the number of displays (`ECU1.display` and `ECU2.display`) 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 two displays or all to have no extra display and either all present displays to be small or all to be large. Also note that we opted to explain the meaning of each feature in terms of the model elements to be selected rather than defining the presence condition of each element in terms of the features. Both approaches are available in Clafer, however.

Constraints allow us restricting a model to a single instance. Figure 7 shows a top-level constraint specifying a single product, with two ECUs, two large displays per ECU, and all components in version 1. Based on this constraint, we can automatically instantiate the product line using the Alloy analyzer, as described in Sect. 5.2.

5 Evaluation

5.1 Analytical Evaluation

We now discuss to what extent Clafer meets its design goals from Sect. 3.

(1) *Clafer provides a concise notation for feature modeling.* This can be seen by comparing Clafer to TVL, a state-of-the-art textual feature modeling language [8]. Feature models in Clafer look very similar to feature models in TVL,

```
Options group allof {
  Size group oneof { Small, Large },
  opt Cache group allof {
    CacheSize group allof {
      SizeVal { int val; },
      opt Fixed
    }
  },
  Constraint { (Small && Cache) -> Fixed; }
}
```

a) Options feature model in TVL

```
class Comp {
  reference version : Integer
}

class ECU extends Comp{ }

class Display extends Comp {
  reference server : ECU
  attribute options : Options
}
```

b) Component meta-model in KM3

Fig. 8. Our running example in TVL and KM3

except that TVL uses explicit keywords (e.g., to declare groups) and braces for nesting. Figure 8a shows the TVL encoding of the feature model from Fig. 3.

Clafer's 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 (a slot) and a type (the type of the slot); for example, all features in Figs. 3 and 6 are of this kind. Neither UML nor Alloy provide this mechanism; in there, a slot and the class used as its type are declared separately.
- *Feature nesting*: Feature nesting accomplishes instance composition and type nesting in a single construct. UML provides composition, but type nesting is specified separately (cf. Fig. 3c). Alloy has no built-in support for composition and thus requires explicit parent-child constraints. It also has no signature nesting, so name clashes need to be avoided using prefixes or alike.
- *Group constraints*: Clafer's group constraints are expressed concisely as intervals. In UML groups can be specified in OCL, but using a lengthy encoding, explicitly listing features belonging to the group. Same applies to Alloy.
- *Constraints with default quantifiers*: Default quantifiers on relations, such as `some` in Fig. 3, allow writing constraints that look like propositional logic, even though their underlying semantics is first-order predicate logic.

(2) *Clafer provides a concise notation for meta-modeling.* Figure 8b shows the meta-model of Fig. 4 encoded in KM3 [21], a state-of-the-art textual meta-modeling language. The most visible syntactic difference between KM3 and Clafer is the use of explicit keywords introducing elements and mandatory braces establishing hierarchy. KM3 cannot express additional constraints in the model. They are specified separately, e.g. as OCL invariants.

It is instructive to 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.

Still, in the worst case, the lack of the previously listed constructs makes Alloy models necessarily larger. Other 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.

(3) *Clafer allows mixing feature and meta-models.* Quotations allow reusing feature or class types in multiple locations; references allow reusing both types and instances. Feature and class models can be related via constraints (Fig. 6).

(4) *Clafer tries to use a minimal number of concepts and has uniform semantics.* While integrating feature modeling into meta-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, besides their obvious role in feature modeling, they also play the role of attributes in meta-modeling. We also contribute a simplification to feature modeling: Clafer has no explicit feature group construct; instead, every feature can use a group cardinality to constrain the number of children. This is a significant simplification, as we no longer need to distinguish between "grouping features" (features used purely for grouping, such as menus) and feature groups. The grouping intention and grouping cardinalities are orthogonal: any feature can be annotated as a grouping feature and any feature may chose to impose grouping constraints on children. Finally, both feature and class modeling have a uniform semantics: a Clafer model instance, just like Alloy's, is a set of relations.

5.2 Experimental Evaluation

Our experiment aims to show that Clafer can express a variety of realistic feature models, meta-models and model templates and that useful analyses can be performed on these encodings in reasonable time. Then it follows that the richness of Clafer's applications, does not come at a cost of lost analysis potential with respect to models in more specialized languages.

The experiment methodology is summarized in the following steps:

1. *Identify a set of models representative for the three main use cases of Clafer: feature modeling, meta-modeling, and mixed feature and meta-modeling.*
2. *Select representative analyses.* We studied the analyses in published literature and decided to focus on a popular class of analyses, which reduce to model instance finding. These include inconsistency detection, element liveness analysis, offline and interactive configuration, guided editing, etc. Since all these have similar performance characteristics, we decided to use model instance finding, consistency and element liveness analysis as representative.
3. *Translate models into Clafer and record observations.* We created automatic translators for converting models to Clafer if it was enough to apply simple rewriting rules. In other cases, translation was done manually.

4. *Run the analyses and reporting performance results.* The analyses are implemented by using our Clafer-to-Alloy translator, and then employing Alloy Analyzer (which is an instance finder) to perform the analysis.

The Clafer-to-Alloy translator is written in Haskell and comprises several chained modules: lexer, layout resolver, parser, desugarer, semantic analyzer, and code generator. Layout resolver makes braces grouping subfeatures optional. 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 (cf. Fig. 3). Semantic analyzer resolves names and deals with inheritance. The code generator translates the core language into Alloy. The generator has benefited from the knowledge about the class of models it is working with to optimize the translation, in the same way as analyzers for specialized languages have this knowledge.

The experiment was executed on a laptop with a Core Duo 2 @2.4GHz processor and 2.5GB of RAM, running Linux. Alloy Analyzer was configured to use Minisat as a solver. All Clafer and generated Alloy models are available at gsd.uwaterloo.ca/sle2010. In the subsequent paragraphs we present and discuss the results for the three subclasses of models.

Feature Models. In order to find representative models we have consulted SPLOT [27] — a popular repository of feature models. We have succeeded in automatically translating all 58 models from SPLOT to Clafer (non-generated, human-made models; available as of July 4th, 2010). These include models with and without cross-tree constraints, ranging from a dozen to hundreds of features.

Results for all models are available online at the above link. Here, we report the most interesting cases together with further four, which have been randomly generated; all listed in Table 1. Digital Video Systems is a small example with few cross-tree constraints. Dell Laptops models a set of laptops offered by Dell in 2009. This is one of few models that contains more constraints than features. Arcade Game describes a product line of computer games; it contains tens of features and constraints. EShop [25] is the largest realistic model that we have found on SPLOT. It is a domain model of online stores. The remaining models are randomly generated using SPLOT, with a fixed 10% constraint/variable ratio.

We checked consistency of each model by instance finding. Table 1 presents summary of results. The analysis time was less then a second for up to several hundred features and less than a minute for up to several thousand features. Interestingly, the biggest bottleneck was the Alloy Analyzer itself (which translates Alloy into a CNF formula)—reasoning about the CNF formula in a SAT-solver takes no more than hundreds of milliseconds.

Meta-Models In order to identify representative meta-models, we have turned to the Ecore Meta-model Zoo (www.emn.fr/z-info/atlanmod/index.php/Ecore), from where we have selected the following meta-models: AWK Programs, ATL, ANT, BibTex, UML2, ranging from tens to hundreds of elements. We translated all these into Clafer automatically. One interesting mapping is the translation of EReference elements with eOpposite attribute (symmetric reference), as there is no

first-class support for symmetric references in Clafer. We modeled them as constraints relating references with their symmetric counterparts. Moreover we have not handled multiple inheritance in our translation.

Since none of these meta-models contained OCL constraints, we extracted OCL constraints from the UML specification [29] and manually added them to the Clafer encoding of UML2. We did observe certain patterns during that translation and believe that this task can be automated for a large class of constraints. Table 2 presents sample OCL constraints translated into Clafer. Each constraint, but last, is written in a context of some class. Their intuitive meanings are as follows: 1) ownedReception is empty if there is no isActive; 2) endType aggregates all types of memberEnds; 3) if memberEnd's aggregation is different from none then there are two instances of memberEnd; 4) there are no two types of the same names. All Clafer encodings of the meta-models are available at the above link.

There are several reasons why Clafer constraints are more concise and uniform compared with OCL invariants. Similarly to Alloy, every Clafer definition is a relation. This approach, eliminates extra constructions such as OCL's collect, allInstances. Finally, assuming the default some quantifier before relational operations (e.g. memberEnd.aggregation - none), we can treat result of an operation as if it was a propositional formula, thus eliminating extra exists quantifiers.

We applied automated analyses to slices of the UML2 meta-model: Class Diagram from [10], State Machines, and Behaviors (Table 3). Each slice has tens of classes and our goal was to include a wide range of OCL constraints. We checked the *strong consistency* property [9] for these meta-models. To verify this property, we instantiated meta-models' elements that were at the bottom of inheritance hierarchy, by restricting their cardinality to be at least one. The same constraints were imposed on containment references within all meta-model elements. The analysis confirmed that none of the meta-models had dead elements. Our results show that liveness analysis can be done efficiently for realistic meta-models of moderate size.

Feature-Based Model Templates. The last class of models are feature-based model templates akin to our telematics example. A FBMT consists of a fea-

Table 1. Results of consistency analysis for *feature models* expressed in Clafer.

model name	nature	size [# features]	[# constraints]	running time [s]
Digital Video System	Realistic	26	3	0.012
Dell Laptops	Realistic	46	110	0.025
Arcade Game	Realistic	61	34	0.040
eShop	Realistic	287	21	0.15
FM-500-50-1	Generated	500	50	0.45
FM-1000-100-2	Generated	1000	100	1.5
FM-2000-200-3	Generated	2000	200	4.5
FM-5000-500-4	Generated	5000	500	28.0

ture model (cf. Fig. 6, left), a meta-model (cf. Fig. 4), a template (cf. Fig. 5a), and a set of mapping constraints (cf. Fig. 6, right). To the best of our knowledge, Electronic Shopping [25] is the largest example of a model template found in the literature. We used its templates, listed in Table 4, for evaluation: FindProduct and Checkout are activity diagram templates, and TaxRule is a class diagram template. Each template had substantial variability in it. All templates have between 10 and 20 features, tens of classes and from tens to hundreds constraints. For comparison, we also include our telematics example.

We manually encoded the above FBMTs in Clafer. For each of the diagrams in [25], we took a slice of UML2 meta-model and created a template that conforms to the meta-model, using mandatory and optional singleton classes as described in Sect. 4.2. To create useful and simple slices of UML diagrams, we removed unused attributes and flattened inheritance hierarchy, since many superclasses were left without any attributes. Thus, the slice preserved the core semantics. Furthermore, we sliced the full feature model, so that it contains only features that appear in diagram. Finally, we added mappings to express dependencies between features and model elements, as described in Sect. 4.3.

We performed two types of analyses on FBMTs. First, we created sample feature configurations (like in Fig. 7) and instantiated templates in the Alloy Analyzer. We inspected each instance and verified that it was the expected one.

Second, we performed element liveness analysis for the templates. The analysis is similar to element liveness for meta-models [9], but now applied to template elements. We performed the analysis by repeated instance finding; in each iteration we required the presence of groups of non-exclusive model elements.

Table 4 presents summary of inspected models and times of analyses. Often the time of liveness analysis is very close to the time of instantiation multiplied by the number of element groups. For instance, for FindProduct, liveness analysis was three times longer than time of instantiation, because elements were arranged into 3 groups of non-conflicting elements. This rule holds when the Alloy Analyzer uses the same scope for element instances.

We consider our results promising, since we obtained acceptable timings for slices of realistic models, without fully exploiting the potential of Alloy. The

Table 2. Constraints in OCL and Clafer.

Context	OCL	Clafer
Class	(not self.isActive) implies self.ownedReception->isEmpty()	~isActive => no ownedReception
Association	self.endType = self.memberEnd-> collect(e | e.type)	endType = memberEnd.type
Association	self.memberEnd->exists(aggregation <> Aggregation::none) implies self.memberEnd->size() = 2	memberEnd.aggregation - none => #memberEnd = 2
–	Type.allInstances() -> forAll (t1, t2 | t1 <> t2 implies t1.name <> t2.name)	all disj t1, t2 : Type | t1.name != t2.name

results can clearly be further improved by better encoding of slices (for example, representing activity diagram edges as relations instead of sets in Alloy) and using more intelligent slicing methods; e.g. some constraints are redundant, such as setting source and target edges in ActivityNodes, so removing these constraints would speed up reasoning process. However already now we can see that Clafer is a suitable vehicle for specifying FBMTs and analyzing them automatically.

Threats to Validity

External Validity Our evaluation is based on the assumption that we chose representative models and useful and representative analyses.

All models, except the four randomly generated feature models, were created by humans to model real-word artifacts. As all, except UML2, come from academia, there is no guarantee that they share characteristics with industrial models. Majority of practical models have less than a thousand features [24], so reasoning about corresponding Clafer models is feasible and efficient. Perhaps the biggest real-world feature model up to date is the Linux Kernel model (almost 5500 features and thousands of constraints) [31]. It would presently pose a challenge for our tools. Working with models of this size requires proper engineering of analyses. Our objective here was to demonstrate feasibility of analyses. We will continue to work on robust tools for Clafer in future.

We believe that the slices of UML2 selected for the experiment are representative of the entire meta-model because we picked the parts with more complex constraints. While there are not many existing FBMTs to choose from, the e-commerce example [25] was reversed engineered from the documentation of an IBM e-commerce platform, which makes the model quite realistic.

Not all model analyses can be reduced to instance finding performed using combinatorial solvers (relational model finder in case of Alloy [34]). However combinatorial analyses belong to most widely recognized and effective [7].

Instance finding for models has similar uses to testing and debugging for programs [19]—it helps to uncover flaws in models, assists in evolution and configuration. For example it helped us discover that our original Clafer code was missing constraints (lines 9–10 and 14–15 in Fig. 5a and line 14 in Fig. 6). 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 [26].

Liveness analysis for model elements has been previously exploited, for instance in [33, 9]. Tartler et al. [33] analyze liveness of features in the Linux kernel

Table 3. Results of strong consistency analysis for UML2 *meta-model* slices in Clafer

meta-model/instance size	[#classes]	[#constraints]	running time [s]
State Machines	11	28	0.08
Class Diagram	19	17	0.15
Behaviors	20	13	0.23

code, reporting about 60 previously unreported dead features in the released kernel versions. Linux is not strictly a feature-based model template, but its build architecture, which relies on (a form of) feature models and presence conditions on code (conditional compilation) highly resembles our model templates.

Analyzers based on instance finding solve an NP-hard problem. Thus no hard guarantees can be given for their running times. Although progress in solver technologies has placed these problems in the range of practically tractable, there do exist instances of models and meta-models, which will effectively break the performance of our tools. Our experiments aim at showing that this does not happen for realistic models.

There exist more sophisticated analyzes (and classes of models) that cannot be addressed with Clafer infrastructure, and are not reflected in our experiment. For example instance finding is limited to instances of bounded size. It is possible to build sophisticated meta-models that only have very large instances. This problem is irrelevant for feature models and model templates as they allow no no classes that can be instantiated without bounds.

Moreover special purpose languages may require more sophisticated analyses techniques such as behavioral refinement checking, model checking, model equivalence checking, etc. These properties typically go beyond static semantics expressed in meta-models and thus are out of scope for generic Clafer tools.

Internal Validity Translating models from one language to another can introduce errors and change semantics of the resulting model.

We used our own tools to convert SPLOT and Ecore models to Clafer and then to translate Clafer to Alloy. We translated FBMTs and OCL constraints manually. The former is rather straightforward; the latter is more involved. We publish all the models so that their correctness can be reviewed independently.

Another threat to correctness is the slice extraction for UML2 and e-commerce models. Meta-model slicing is a common technique used to speed-up model analyses, where reasoner processes only relevant parts of the meta-model. We performed it manually, while making sure that all parts relevant to the selected constraints were included; however, the technique can be automated [30].

The correctness of the analyses relies on the correctness of the Clafer-to-Alloy translator and the Alloy analyzer. The Alloy analyzer is a mature piece of software. We tested Clafer-to-Alloy translator by translating sample models to Alloy and inspecting the results.

Table 4. Analyses for *Feature-Based Model Templates* expressed in Clafer. Parentheses by the model names indicate the number of optional elements in each template.

FBMT	#features/#classes/#constraints	instantiation [s]	element liveness [s]
Telematics (8)	8/7/17	0.04	0.26
FindProduct (16)	13/29/10	0.07	0.18
TaxRules (7)	16/24/62	0.11	0.12
Checkout (41)	18/78/314	1.6	5.8

6 Related Work

We have already mentioned related work on model analysis; here we focus on work related to our main contribution, Clafer's novel language design.

Asikainen and Männistö present Forfamel, a unified conceptual foundation for feature modeling [4]. The basic concepts of Forfamel and Clafer are similar; both include 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 by stating an indented feature name. Also, the conceptual foundations of Forfamel and Clafer differ; e.g., features in Forfamel correspond to Clafer's instances, but features in Clafer are relations. Also, a feature instance in Forfamel can have several parents; in Clafer, an instance has 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; Clafer limits feature modeling to its original FODA scope [22], but integrates it into class modeling. Finally, Forfamel considers a constraint language as out of scope, hinting at OCL. Clafer comes with a concise constraint notation.

TVL is a textual feature modeling language [8]. 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. 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 only partially possible.

As mentioned earlier, class-based meta-modeling languages, such as KM3 [21] and MOF [28] cannot express feature models as concisely as Clafer.

Nivel is a meta-modeling language, which was applied to define feature and class modeling languages [3]. 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 [16]; a Clafer-like graphical profile for Ecore, having a bidirectional translation between an annotated Ecore model and its rendering in the graphical syntax [32]; and the Clafer-like notation used to specify framework-specific modeling languages [2]. None of these works provided a proper language definition and implementation like Clafer; also, they lacked Clafer's concise constraint notation.

Gheyi et al. [17] pioneered translating Boolean feature models into Alloy. Anastasakis et al. [1] automatically translated UML class diagrams with OCL constraints to Alloy. Clafer covers both types of models.

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 [13, 18]. 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. This design allows us to elegantly reuse a single constraint language at all these levels. As another example, Janota and Botterweck show how to relate feature and architectural models using constraints [20]. 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 [5], 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.

7 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 set off to integrate feature modeling into class modeling, rather than trying to extend feature modeling as previously done [15]. 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. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: On challenges of model transformation from UML to Alloy. Software and Systems Modeling 9(1) (2008)
2. Antkiewicz, M., Czarnecki, K., Stephan, M.: Engineering of framework-specific modeling languages. IEEE TSE 35(6) (2009)
3. Asikainen, T., Männistö, T.: Nivel: a metamodelling language with a formal semantics. Software and Systems Modeling 8(4) (2009)
4. Asikainen, T., Männistö, T., Soininen, T.: A unified conceptual foundation for feature modelling. In: SPLC'06
5. Asikainen, T., Männistö, T., Soininen, T.: Kumbang: A domain ontology for modelling variability in software product families. Adv. Eng. Inform. 21(1) (2007)

6. Bart Veer, J.D.: The eCos Component Writer's Guide (2000)
7. Benavides, D., Segura, S., Ruiz-Cortés, A.: Automated analysis of feature models 20 years later: a literature review. Information Systems 35(6) (2010)
8. Boucher, Q., Classen, A., Faber, P., Heymans, P.: Introducing TVL, a text-based feature modelling language. In: VaMoS'10
9. Cabot, J., Clarisó, R., Riera, D.: Verification of UML/OCL Class Diagrams Using Constraint Programming. In: MoDeVVA'08
10. Cariou, E., Belloir, N., Barbier, F., Djemam, N.: Ocl contracts for the verification of model transformations. In: OCL workshop of MoDELS'09
11. Clauß, M., Jena, I.: Modeling variability with UML. In: YRW at GCSE'01
12. Czarnecki, K., Pietroszek, K.: Verifying feature-based model templates against well-formedness ocl constraints. In: GPCE'06
13. Czarnecki, K., Antkiewicz, M.: Mapping features to models: A template approach based on superimposed variants. In: GPCE'05
14. Czarnecki, K., Bednasch, T., Unger, P., Eisenecker, U.: Generative programming for embedded software: An industrial experience report. In: GPCE'02
15. Czarnecki, K., Helsen, S., Eisenecker, U.: Formalizing cardinality-based feature models and their specialization. SPIP 10(1) (2005)
16. Czarnecki, K., Kim, C.H.: Cardinality-based feature modeling and constraints: A progress report. In: OOPSLA'05 Workshop on Software Factories
17. Gheyi, R., Massoni, T., Borba, P.: A theory for feature models in Alloy. In: First Alloy Workshop (2006)
18. Heidenreich, F., Kopcsek, J., , Wende, C.: FeatureMapper: Mapping Features to Models. In: ICSE'08
19. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press (2006)
20. Janota, M., Botterweck, G.: Formal approach to integrating feature and architecture models. In: FASE'08
21. Jouault, F., Bézivin, J.: KM3: a DSL for Metamodel Specification. In: IFIP'06
22. Kang, K., Cohen, S., Hess, J., Nowak, W., Peterson, S.: Feature-oriented domain analysis (FODA) feasibility study. Tech. Rep. CMU/SEI-90-TR-21, CMU (1990)
23. Kang, K.C.: FODA: Twenty years of perspective on feature modeling. In: VaMoS'10
24. Kästner, C.: Virtual Separation of Concerns: Toward Preprocessors 2.0. Ph.D. thesis, University of Magdeburg (2010)
25. Lau, S.Q.: Domain Analysis of E-Commerce Systems Using Feature-Based Model Templates. Master's thesis, University of Waterloo (2006)
26. Le Berre, D., Rapicault, P.: Dependency management for the Eclipse ecosystem: Eclipse p2, metadata and resolution. In: IWOCE'09
27. Mendonça, M., Branco, M., Cowan, D.: S.P.L.O.T. - Software Product Lines Online Tools. In: OOPSLA'09
28. OMG: Meta Object Facility (MOF) Core Specification (2006)
29. OMG: OMG Unified Modeling Language (2009)
30. Shaikh, A., Clarisó, R., Wiil, U.K., Memon, N.: Verification-Driven Slicing of UML/OCL Models. In: ASE'10
31. She, S., Lotufo, R., Berger, T., Wasowski, A., Czarnecki, K.: Variability model of the linux kernel. In: VaMoS'10
32. 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)
33. Tartler, R., Sincero, J., Lohmann, D.: Dead or Alive: Finding Zombie Features in the Linux Kernel. In: FOSD'09
34. Torlak, E., Jackson, D.: Kodkod: A relational model finder. In: TACAS'07