Optimized Translation of Clafer Models to Alloy

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

Abstract

Clafer is a textual language for modeling and analysis of software product lines. The language is able to express a wide range of models: feature models, meta-models, and feature-based model templates. Reasoning about those models is required to verify their consistency, detect dead elements, or to guide the configuration process for end-users. Clafer uses the Alloy Analyzer as a back-end to perform analyses on variability models. This paper presents how optimized translation rules of Clafer models to Alloy simplified the Alloy models and improved reasoning performance. The work was evaluated on publicly available feature models, slices of meta-models, and feature-based model templates. The results showed that the improved translation shortened reasoning time at least 2 times. The improvement by factor of 4 was not uncommon.

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

General Terms Design, Languages, Modeling

Keywords feature modeling, meta-modeling, constraints, optimization

1. Introduction

Software product lines (SPLs) are families of related systems built from shared assets, such as models, or software components. The premise of SPLs is to increase productivity, decrease time to market, and improve overall software quality by systematic reuse of assets. Software product lines are often described in terms of *problem space*, *solution space*, and a *mapping* between them. Figure 1 shows this conceptual view.

Problem space specifies high-level requirements and products available in the SPL. It captures commonalities and variabilities among the products. For example, every car has an engine; the user, however, can choose between electric and gasoline engines. Solution space defines product line architecture, usually as components and connectors. Mapping links the two spaces, so that elements from problem space are related to elements from solution space. That way when user specifies concrete requirements in problem space, that configuration is reflected in solution space and the user obtains an automatically generated product.

[Copyright notice will appear here once 'preprint' option is removed.]

1.1 Problem Space

Feature modeling is a technique used for capturing commonalities and variabilities in software product lines. Feature models naturally fit into problem space as they determine the products available in product line. They were introduced by Kang et al. as a part of *Feature-Oriented Domain Analysis* (FODA) methodology [13].

FODA feature models are tree-like structures, as the one in Fig. 2 representing an automobile product line. Each box represents a *feature*, that is a user-relevant product characteristic, such as Engine or Radio. Mandatory features are marked with filled circles; optional features with hollow circles. In our example, every car must have an Engine, but not every car must have a Radio. Feature models also have *alternative groups* (marked with empty arcs) and *or-groups* (marked with filled arcs). An *alternative group* allows to select only one subfeature, e.g. a car must have either Gasoline or Electric engine. An *or-group* requires at least one subfeature to be selected, e.g. a valid car configuration has either CD Player, or Tape player, or both of them. Finally, some constraints cannot be encoded in the tree structure. In that case we use Cross-Tree Constraints (CTCs) specified as propositional formulas to restrict the feature model. For instance, the constraint below the tree says that cars with Electric engine must have some Radio player on board.

Several extensions to feature models have been proposed. The most common ones include *attributes* [2] and *feature cardinalities* [10]. An attribute, allows feature to store a value of primitive type (such as integer, string, enumeration). Feature cardinalities allow to have several copies of each feature, e.g. a car can have several displays. Each display is configured separately, though. In FODA all features have cardinality either 1 (mandatory feature), or 0..1 (optional feature).

1.2 Solution Space

Solution space is composed of different types of artifacts: software components, software models, engineering models, etc. Those assets are building blocks for software delivered by the SPL. Assets have built-in variability, which is resolved later when user configures feature model. In many cases software variabilities are expressed by conditional compilation, such as IFDEF directives in C source code. Thus, the artifacts define a family of products.

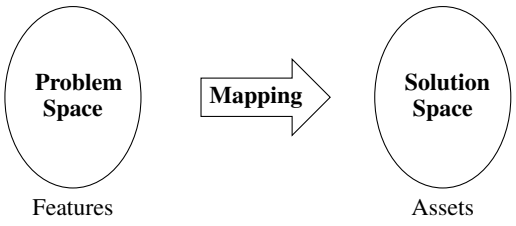


Figure 1. Software product line conceptual view

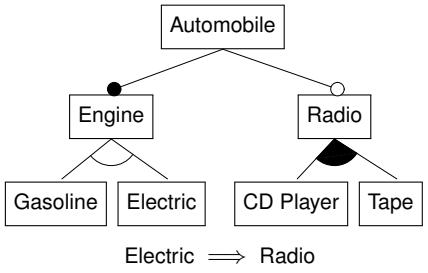


Figure 2. Feature model for automobile product line

On the high level, components and connectors from solution space can be modeled by class models, meta-models, and additional constraints that altogether define product line architecture. Mapping between problem and solution space specifies relations between features and assets. It links features to implementations. In many practical examples of product lines the mapping is defined by preprocessor directives and makefiles that orchestrate conditional compilation.

Feature modeling and class modeling notations were designed with different purposes in mind and model different types of variability. Feature models capture selections from predefined fixed structure, thus we always 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.

1.3 Clafer

Clafer [4] is a textual variability modeling language for specification and analysis of software product lines. It covers the three conceptual elements of software product lines: problem space, solution space, and mapping. The 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. A Clafer model is composed of *clafers* and constraints. A clafer is an element that unifies the notion of feature with the notion of class.

Clafer aims at providing a common infrastructure for analyses of feature and meta-models. The need for analyses was recognized, for example, in the operating systems domain by the Linux Kernel and eCos developers [3]. They both provide variability models of system kernels that are supported by configuration tools. The tools guide the configuration process, so that end-users are less likely to build incorrect kernels. Many non-trivial analyses of variability models are reducible to the NP-hard problem of finding model instances by combinatorial solvers. At present, Clafer translates input models to Alloy [12]. Alloy uses SAT solvers to do model checking, or to find model instances.

We developed a Clafer to Alloy translator named clafer2alloy. The translator provides a uniform translation for feature and meta-models. Unfortunately, it negatively impacts reasoning efficiency as the reasoner is unable to exploit properties of those models. It turned out that translation rules heavily influence reasoning time in Alloy. The clafer2alloy translator generated overly large and complex Alloy models. The goal of my course project was to **improve reasoning time in Alloy** by optimizing Clafer to Alloy translation rules. We also **optimized the** clafer2alloy **translator** to enable it to handle large input models. In particular, the contribution includes:

1. Refactoring clafer2alloy code. That step was crucial to make the code more modular. The translator can use multiple code generators (thus is not limited to Alloy), and makes it easier to add new functions to the translator.
2. Intermediate language representation. The new data structure for representing Clafer models is better suited for compile-time analyses and transformations. It is easier to store analyses results together with the input Clafer model.
3. Optimization of translation rules. Depending on the input model the translator tries to apply as many optimizations as possible. The rules result in producing smaller and less complex Alloy models.
4. User controls the translation process. Users may leverage their knowledge about input models to apply certain optimizations and shorten the translation time.

```
Automobile                  <0−*> Automobile 1..1 {
    xor Engine                  <1−1> Engine 1..1 {
        Gasoline                    <0−*> Gasoline 0..1 {}
        Electric                    <0−*> Electric 0..1 {}
    or Radio ?                  }
        CDPlayer                <1−*> Radio 0..1 {
        Tape                        <0−*> CDPlayer 0..1 {}
                                    <0−*> Tape 0..1 {}
[Electric ⟹ Radio]              }
                            }
                            [some Electric ⟹ some Radio]
```

a) Concise notation b) Desugared notation

Figure 3. Clafer automobile model

The paper is organized as follows. We describe variability modeling and analysis in Clafer in Sect. 2. We discuss the challenges with old translation rules and optimized Clafer to Alloy translation in Sect. 3. We evaluate the language experimentally in Sect. 4. We conclude in Sect. 6, after comparing Clafer with related work in Sect. 5.

2. Variability Models in Clafer

Figure 3a shows a Clafer model that corresponds to the automobile feature model from Fig. 2. Hierarchy of model elements is established by code indentation. Feature group (*alternative* and *or*) specification precedes the name (e.g. xor before Engine). Optionality of elements is marked by the question mark following the name (e.g. Radio). Finally, cross tree constraints are specified in square brackets.

Such a Clafer model expresses the space of available products. Its meaning is given by *configuration semantics*, that is all possible configurations that conform to the variability model. For our example this set is shown in Fig. 4. Please note that each car with Electric engine must have a Radio. That requirement was specified as a cross-tree constraint. The number of possible configurations grows exponentially with the number of elements in the tree. FODA feature models consider only logical implications and exclusions as valid constraints. Clafer allows much more complex expressions including arbitrary first-order logic formulas.

2.1 Consistency

Clafer models are either consistent or inconsistent. A Clafer model is consistent if there is at least one instance satisfying all structural and cross-tree constraints. Structural constraints are imposed by hierarchical dependencies among parent and children elements (e.g. an Engine cannot exist without Automobile). To restrict the

{{Automobile, Engine, Electric, Radio, CDPlayer},
{Automobile, Engine, Electric, Radio, Tape},
{Automobile, Engine, Electric, Radio, CDPlayer, Tape},
{Automobile, Engine, Gasoline},
{Automobile, Engine, Gasoline, Radio, CDPlayer},
{Automobile, Engine, Gasoline, Radio, Tape},
{Automobile, Engine, Gasoline, Radio, CDPlayer, Tape}}

Figure 4. Valid configurations of the model from Fig. 3a

model from Fig. 3a to a single configuration we have to add an extra constraint, e.g. [Automobile ∧ Engine ∧ Gasoline]. The constraint corresponds to a selection of features. Such a model results in the configuration {Automobile, Engine, Gasoline} belonging to the set defined in Fig. 4. Therefore, it is consistent.

Inconsistent Clafer models have no valid configurations. If we added the constraint [¬ Radio ∧ Electric] specifying that there is no Radio, but the engine is Electric we would obtain an inconsistent Clafer model. The model is inconsistent since {Automobile, Engine, Electric} does not belong to the set of valid configurations in Fig. 4. The added constraint contradicts the cross-tree constraint from Fig. 3a.

2.2 Analyses

Model analyses are useful for software product line developers and end users who configure products. Analyses over Clafer models allow to detect problems with SPLs at the earliest possible stage. This reduces cost of development and deployment. What makes most of analyses hard in practice is the size of configuration space, presence of cross-tree constraints, and the use of non-Boolean types (such as integer or string) in constraints.

The existing Clafer infrastructure supports checking model consistency, and finding contradicting constraints by means of the Alloy Analyzer. Semi-automatically, we are able to detect dead elements and repair inconsistent configurations. A dead element is an element that never appears in any configuration. Although it exists in the model or code-base, it is never used by any product. Furthermore, when a configuration is invalid, the tools present a set of contradicting constraints. Then the user has a chance to fix the model.

Analyses over Clafer models reason about the whole space of possible configurations without actually generating all SPL products. This is what Clafer achieves. Theoretically, for a given variability model one could enumerate all possible configurations, deploy them, and then check for given property. For the automobile example, there would be 8 configurations, of which one would be invalid. Each configuration corresponds to building a new software version. In practice this approach is infeasible, since the number of products grows exponentially with the number of model elements.

2.3 The Toolchain

Clafer reuses existing tools to perform model analyses. We picked Alloy as our target language. Alloy is a structural modeling language that can do model checking and model instance generation. The second case can be thought of as generation of test cases for a given specification. Many analyses are reducible to solving either of the two problems. Alloy provides clean syntax and has good expressivity. Alloy models are as expressive as first-order logic extended with transitive closure. Alloy models are composed of sets, relations, and constraints. The Alloy Analyzer uses SAT solvers to reason on models. If a model is consistent, then the analyzer shows a valid model instance, e.g. Fig. 5 shows an instance of the car variability model. If a model is inconsistent, it highlights the conflicting constraints, e.g. Fig. 6 shows contradicting constraints for a wrong car configuration.

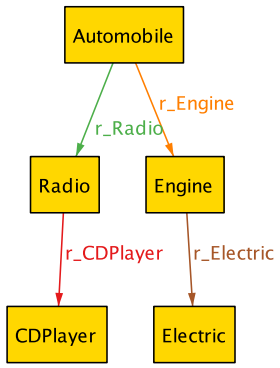


Figure 5. Alloy instance of the automobile model

```
fact { (some @r_Electric) => (some @r_Radio) }
fact { (no @r_Radio and some @r_Electric) }
```

Figure 6. Inconsistent Alloy constraints in an invalid automobile configuration

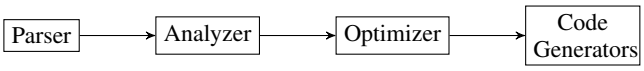


Figure 7. Architecture of the clafer translator

Figure 7 shows architecture of the Clafer translator. The translator takes Clafer model as input, desugars the model by expanding default values (compare Fig. 3a and Fig. 3b), performs semantic analysis, applies optimizations, and finally generates Alloy code. The translator is implemented in Haskell; its frontend is generated by BNFC [18].

Hierarchical structure is one of the most important characteristics of Clafer models, while in Alloy the hierarchy must be established manually. The Clafer to Alloy translator maps each clafer to a set and relation that relates the set to its owner. In the Alloy model, each signature corresponds to single clafer and contains a list of relations to its children. The signature is followed by an optional list of invariants that must hold for each signature instance. For example, the Engine clafer from Fig. 3a is translated to Engine signature in Fig. 8. The Engine signature has two relations to its children: Gasoline and Electric. Engine is related with its parent by the r_Engine relation . The r_Engine relation is placed under the parent (i.e. Automobile). In that way we establish hierarchy among Alloy signatures.

Alloy performs bounded model analysis on the model. Before performing the analysis we need to set the scope parameter. The scope determines the maximum number of elements belonging to each set (signature). The scope of analysis is set in the second line of Fig. 8 to 1. So for our example there is at most one element of each signature, i.e. at most one Automobile, at most one Engine, etc. The larger the scope, the longer it takes to reason about the model. For each model one has to determine the scope separately. If the scope is too small, then the Alloy Analyzer will not be able to do the analysis. An Alloy model is a mixture of relational, first-order, and navigational logic. The Alloy Analyzer uses KodKod to translate the input model to a 3CNF formula that can be reasoned upon by a SAT solver.

```
pred show {}
run show for 1

one sig Automobile
{ r_Engine : one Engine
, r_Radio : lone Radio }

one sig Engine
{ r_Gasoline : lone Gasoline
, r_Electric : lone Electric }
{ let children = (r_Gasoline + r_Electric) | one children }

lone sig Gasoline {}
{ one r_Gasoline }

lone sig Electric {}
{ one r_Electric }

lone sig Radio
{ r_CDPlayer : lone CDPlayer
, r_Tape : lone Tape }
{ one r_Radio
  let children = (r_CDPlayer + r_Tape) | some children }

lone sig CDPlayer {}
{ one r_CDPlayer }

lone sig Tape {}
{ one r_Tape }

fact { (some r_Electric) ⟹ (some r_Radio) }
```

Figure 8. Alloy automobile model

3. Optimizations

We created the old clafer2alloy translator as a research prototype. It was good at handling small models, but made reasoning on models with more than 20 elements virtually impossible. Below we describe various optimizations that allowed to scale the translator to automatically handle models with thousands of elements.

3.1 Improved Name Resolution

The hierarchical structure of Clafer models allows arbitrary nesting of clafers. Each clafer implicitly introduces a new namespace. Names do not have to be globally unique; the only restriction is that a parent cannot have two children of the same name. Such a naming policy requires a name resolution algorithm to resolve ambiguous names. Those ambiguities appear especially in constraints since they may refer to any names used in the model.

A name is resolved in context of a clafer in up to six steps. First, it is checked to be a special name like this or parent. Secondly, the name is looked up in subclafers in breadth-first search manner. If it is still not found, the algorithm searches in the top-level definition that contains the clafer 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.

A significant difference between Clafer and Alloy is lack of nesting in the latter. Alloy namespace within a file is mostly flat. The Clafer translator must keep track of clafer names. The old translator provided limited support for that, which resulted in acceptance of incorrect models. For example, if we added the constraint [Engine.Hybrid] under Engine in Fig. 3 (requiring that there is a hybrid engine), then we would reference a non-existing clafer: Hybrid. The new translator uniquely identifies each name by combining its original name with an integer. It allows to statically detect ill-specified constraints.

Furthermore, when a name is resolved in Clafer, the translator generates a fully qualified name as a path expression. Paths expressions are lists of identifiers separated by dots, e.g. Engine.Electric. They are used for navigation in programming languages, OCL, or Alloy. Path expressions in programming languages have always unique source, e.g. a concrete instance of Engine. In databases and Alloy there may be multiple sources, i.e. multiple instances of Engine. There dot is not a simple mechanism of referencing elements, but more of a database join operator.

Path expressions in Alloy use the dot operator (relational join) to navigate among relations. Alloy's dot operator removes the column on which relations are joined. For example, the Electric identifier from constraint in Fig. 3a is resolved as Automobile.Engine.Electric. The result of joining the three relations in Alloy is a set of electric engines of all automobiles. In the running example there may be only one such engine as there is only one automobile.

In programming languages path expressions are just pointers to memory addresses. They are computed within constant time. Path expressions in Alloy are the actual operations of joining relations. They require reasoning to navigate from Automobile to the Electric engine. In practice the use of path expressions slows down the reasoning process, as SAT solver has more work to do. The Clafer translator is aware of this problem and uses a simple heuristic to eliminate reasoning over path expressions.

The translator checks if all names in the model are unique. If this is the case, then no path resolution is required (no use of the dot operator). So Electric in the constraint would be kept as is (see last line in Fig. 8), since it is the only clafer of this name, and does not require disambiguation. If names are not unique, then the translator uses path expressions to resolve names. The improved name resolution algorithm shortens reasoning time and decreases the size of Alloy models.

3.2 Inheritance Flattening

There are three types of clafers: concrete, abstract, and reference. Concrete clafers define a new type and have cardinalities that specify the number of possible instances. All elements in Fig. 3 are of this type. Concrete clafers are always part of the configuration semantics. If an element is concrete, then all its subelements are concrete. Abstract clafers define only a type and cannot be instantiated unless extended by a concrete clafer. Thus, they are part of configuration semantics if at least one concrete clafer extends them. This distinction is somehow similar to abstract classes in object-oriented programming: one cannot create an object of abstract class. Finally, reference clafers are references that point to instances of clafers existing in the configuration semantics. Thanks to references Clafer models are not restricted to trees, but may be arbitrary graphs.

Clafer provides inheritance as a mechanism of reuse. Figure 9a shows Clafer model with an abstract Component. Each component has the isDependent flag indicating whether the component depends on other components or not. There are also two concrete components: display and ecu. The constraint in the last line requires that there is at least one dependent component. In total, there are three possible configurations of the model, i.e. display and ecu are always present, but at least one of them must have the isIndependent flag set.

Inheritance allows to write models more concisely and in a way that better matches user's mental model. Unfortunately, it slows down the analysis in the Alloy Analyzer, because requires increasing the analysis scope. In Fig. 9a both concrete elements are of type component, therefore the smallest scope must be set to 2. The Com-

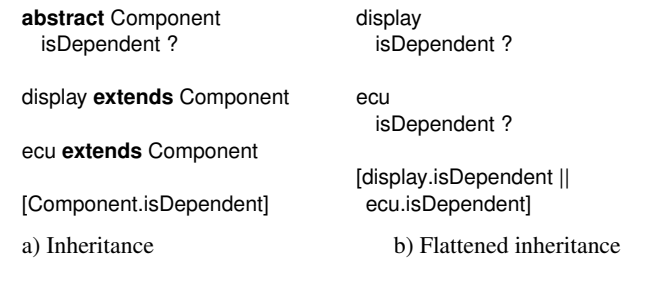


Figure 9. Inheritance flattening in Clafer

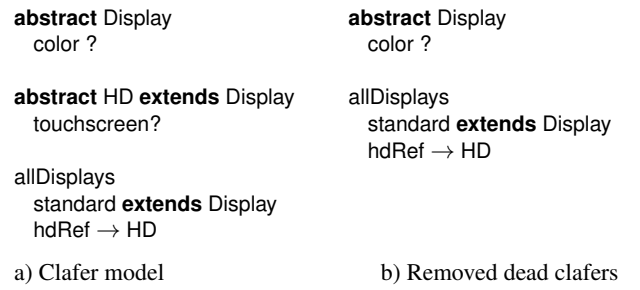


Figure 10. Removal of dead clafers

ponent signature in Alloy will contain two elements: one display component, and one ecu component. When using inheritance, the scope can grow very rapidly. One solution to this problem is to use inheritance flattening.

Inheritance flattening rewrites subfeatures of an abstract clafer as subfeatures of a concrete clafer that extends the abstract one (see Fig. 9b). The Clafer translator must also rewrite constraints that refer to abstract elements. For example, the constraint from Fig. 9a concerns all components. Therefore, the constraint must be split so that the name of abstract clafer is substituted by names of concrete clafers that extend it.

The translator must carefully analyze the model to be sure whether it can apply the flattening. In cases when there are circular dependencies, flattening leads to an infinite loop. The translator automatically detects when it can flatten inheritance. After applying inheritance flattening the resulting Alloy models typically grow in size (in terms of lines of code), but they are much easier to analyze for SAT solver. Elimination of inheritance allows to reduce the scope in Alloy. In our example from Fig. 9b there is one instance of display and ecu, and they have no common superclafer. The scope can be set to 1. Alloy reasons on such signatures very efficiently.

3.3 Dead Abstract Clafers Removal

Some Clafer models (especially meta-models), contain a significant number of abstract clafers. Figure 10a shows a model with two abstract clafers (Display and HD), and a concrete one: allDisplays. The allDisplays clafer aggregates all displays available in a car. Here the car has only one standard display. We use inheritance to say that standard is of type Display, i.e. it extends the abstract element. Furthermore allDisplays also has the hdRef reference pointing to an HD display. References point to clafer instances of given type. Please note that in the model there are no instances of HD. We will discuss this issue later.

An abstract clafer is considered dead if it is not extended by any concrete clafer (directly or indirectly). In the example HD is a dead element, but Display is not. The latter is extended by standard that belongs to a concrete clafer. To reduce the size of Alloy models, the Clafer translator automatically finds and removes dead abstract clafers. Dead elements are safe to remove, because they have no impact on configuration semantics of Clafer model. Figure 10b shows the same model, but with removed dead element. Configuration semantics of both models is exactly the same.

The algorithm for finding dead abstract clafers is only concerned with inheritance among clafers. It does not, however, take references into account. Although, hdRef points to HD, it does not prevent the algorithm from removing the clafer. This is an interesting case because in Fig. 10 hdRef points to the type that is no longer available in the model. The resulting model is incorrect. At present the Clafer translator does not issue any warning when that happens, but the problem is reported later by the Alloy Analyzer.

Referencing a dead abstract clafer that has no valid instances indicates a problem with the model. Such a model is invalid, because pointing to something that cannot exist cannot be satisfied by the reasoning engine. The Alloy Analyzer reports this problem during the type-checking phase, so user has a chance to learn about the error even before the actual reasoning. To fix the issue user should either add a concrete element that extends the abstract clafer, or should remove the problematic reference. Besides decreasing the size of Alloy models, removal of dead abstract elements helped us to discover errors in some parts of the eCos variability model translated to Clafer (the original eCos model was correct, though).

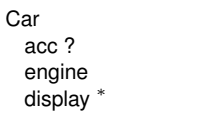
3.4 Hierarchical Constraints

Clafer models are mostly hierarchical structures, i.e. parents own their subclafers and a child cannot exist if its parent is absent. Those two constraints must hold between each pair of clafers in parent-child relation. Figure 11a presents Clafer model with features of a Car. Each car must have an engine, may optionally have Adaptive Cruise Control system (acc), and may have any number of displays. The syntactic sugar for clafer cardinalities (marked with the question mark, asterisk, or nothing) is expanded in Fig. 11b.

The parent-child constraints are among the most often appearing constraints in Clafer models. Therefore, it is a good idea to have them as few as possible to simplify reasoning. For our car example the old clafer2alloy translator generated the Alloy model in Fig. 12a. Each signature extends an abstract set named clafer. The clafer signature declared the parent relation among two clafers. That way, the translator did not have to define this relation for each signature but had to constrain the parent relation to point to the actual parent. It simplified translation, because the translator did not have to keep track of all types and names.

The acc, engine, and display signatures have the constraint specified below them. The <: operator corresponds to mathematical domain restriction. It was needed to overcome the limited name resolution to avoid name clashes. Once the domain of relations is restricted to those belonging to Car, it is joined by the dot operator with the current instance of a signature (this). Relational join is similar to join in databases but it removes the column on which join occurs. Join with this uniquely identifies parent, since only parent can own the this instance. This uniform constraint must be solved by SAT solver. It negatively impacts the overall reasoning performance and also clutters Alloy models.

The new Clafer translator is much smarter about establishing parent-children relationships. Figure 12b shows the car model with the same configuration semantics, but different Alloy representation. The translation does not introduce an explicit parent relation anymore. Also, it avoids the domain restriction operation thanks to better name resolution algorithm. Furthermore, the parent-child relationship may be unnecessary in cases when both parent and



Element	Cardinality
Car	1..1
acc	0..1
engine	1..1
display	0..*

a) Car Clafer model b) Elements' cardinalities

Figure 11. Clafer model with various cardinalities

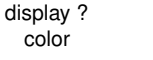
```
abstract sig clafer
{parent : one clafer}

sig Car extends clafer
{ r_acc : lone acc
, r_engine : one engine
, r_display : set display }

sig acc extends clafer {}
{ parent = (Car <:r_acc).this }

sig engine extends clafer {}
{ parent = (Car <:r_engine).this }

sig display extends clafer {}
{ parent = (Car <:r_display).this }
```

a) Extra integer signatures

```
one sig Car
{ r_acc : lone acc
, r_engine : one engine
, r_display : set display }

lone sig acc {}
{ one r_acc }

one sig engine {}

sig display {}
{ one r_display.this }
```

b) Integer attribute

Figure 12. Integer optimization



Element	Cardinality	Global cardinality
display	0..1	0..1
color	1..1	0..1

a) Clafer model b) Elements' cardinalities

Figure 13. Global cardinalities

child are mandatory, such as Car and engine. When the element is optional, such as acc, then only the information about one parent is necessary to enforce parent-child relationship. In most general cases, such as display, where there can be any number of displays, the join is unavoidable. It must be there to uniquely identify parent of current instance (this). Lack of this constraint would allow to have an instance of child when the parent is absent. The new translation eliminates a fair number of constraints and leverages domain knowledge to optimize Alloy models.

3.5 Global Cardinalities

Each definition of a clafer consists of several properties. As we could see each clafer has a name. It also has a cardinality that specifies the number of clafer instances. In Clafer optional features are followed by the question mark. Features whose cardinality is not explicitly specified are mandatory in most of the cases. For example, Fig. 13a shows model of a display that must be color. The color feature is mandatory, while display itself is optional. Those cardinalities are specified explicitly in the second column of Fig. 13b.

A straightforward translation to Alloy yields the code visible in Fig. 14a. There are two sets (signatures): one for display and

```
sig display
{ r_color : one color }

sig color {}
{ one r_color }

fact { lone display }
```

a) No global cardinalities

```
lone sig display
{ r_color : one color }

lone sig color {}
{ one r_color }
```

b) Global cardinalities

Figure 14. Added signature cardinalities

one for color. The display signature contains the r_color relation that indicates that each display instance is related with one color instance. On the other hand, color cannot exist when its parent does not exist. For this reason we add one constraint under color to indicate that there is exactly one parent: the r_color relation has one element. Those two constraints guarantee that the child exists if and only if parent is part of the model. The parent (display) is itself optional, which is specified by the fact in last line. In Alloy lone stands for one or less than one.

This model expresses the configuration semantics that we have in mind. The practical problem is that the Alloy Analyzer does not know in advance how many instances of display and color there will be (so 0, 1, 6 are equally likely numbers). It cannot make any assumptions, and will use SAT solver to determine this number. The SAT solver will have to find a solution that satisfies the constraint specified as a fact. This obviously has negative impact on the reasoning time since SAT solver has more constraints to satisfy.

A modeler, on the other hand, already knows the maximum number of clafer instances. If display is present then color is also present (hence the upper limit of color is equal to 1). If there is no display, then there is no color (hence the lower limit of color is equal to 0). We define clafer's *global cardinality* as a range $mm'..nn'$, where $m..n$ is parent's global cardinality and $m'..n'$ is clafer's cardinality. For elements that have no parent global cardinality is equal to element's cardinality. Global cardinality tells us how many instances of given element there might be on the whole (depending on presence of its parent). The third column of Fig. 13b shows global cardinalities for the example from Fig. 13a.

Alloy provides special syntax for those global cardinalities. They may precede a signature definition as in Fig. 14b. Both signatures have 0..1 global cardinalities and are specified by the lone keyword. That way the Alloy Analyzer knows in advance that there will be up to one instance of each signature. The SAT solver no longer has to find that number. It enables the Alloy Analyzer to do more optimizations and generate formulas that are easier to solve in SAT solver. From our experience we can tell that those global cardinalities are of huge help to the Alloy Analyzer, especially when dealing with big feature models.

3.6 Primitive Types

FODA feature models involve only propositional constraints. Many practical variability models are much richer, i.e. they involve constraints over integers and strings. [3, 9]. Reasoning over feature models with attributes of primitive types is even harder than reasoning over FODA feature models. Currently Clafer uses Alloy to provide limited reasoning over integers and strings. Certain string operations (such as equality) are easily reducible to reasoning over integers.

Figure 15 shows a small example of model with primitive types. The model has a Component with a subclafer that specifies component's major version. Version is specified as integer, a primitive

```
Component
  version : integer
    subversion : integer
```

Figure 15. Component Clafer model

```
abstract sig integer {val : Int}

one sig Component
{ r_version : one version }

one sig version extends integer
{ r_subversion : one subversion }

one sig subversion extends integer {}
```

a) Extra integer signatures

```
one sig Component
{ r_version : one version }

one sig version
{ ref : one Int
, r_subversion : one Int }
```

b) Integer attribute

Figure 16. Integer optimization

type in Clafer. The version clafer contains subversion that specifies component's minor version.

The old Clafer to Alloy translation treated all model elements in a uniform way. For that reason version, subversion and integer were distinct signatures in Alloy (see Fig. 16a). Whenever integers were used in the model, there was always an extra clafer extending the integer signature. Those clafers had the val relation that pointed to Int, which is Alloy's primitive type for integer. Such a solution obviously introduced overhead, both to model size and reasoning time. The main problem was extension of the integer signature. Each extension required to increase the global scope for model analysis in Alloy. Even in small models the number quickly reached value that made it impossible to perform analysis in a reasonable time.

The new Clafer translator simplified handling of primitive types in several ways. First of all, it no longer needs the global signature for integers. Lack of global signature decreases the scope parameter in Alloy. Secondly, the translator recognizes clafers of integer type and treats them in a special way. Whenever there is an integer clafer and it has no subclafers, it is translated directly to a relation with Int, e.g. r_subversion in Fig. 16b. In cases when there are subclafers, the translator must still create a new signature, e.g. for version in Fig. 16b to relate it with subversion. The extra ref relation specifies component's version. In either case the Alloy model can be analyzed with small scope.

3.7 References

In class/meta-modeling it is common to use references to point to elements in other parts of the model. References may simplify constraints, or might be necessary when setting some property according to existing element. References add expressiveness to the language, since the models are no longer trees, but arbitrary graphs. An example of model with references is shown in Fig. 17. The model is composed of two concrete clafers: Controller and Engine. The former has a flag indicating whether it is the main engine controller. The Engine contains a reference to the Controller that controls it. The ctrl clafer has also a reference subclafer that points to the isMain flag of its controller.

The old clafer2alloy translator relied very much on Alloy constraints when generating code for reference clafers. Figure 18a shows translation of the engine controller model to Alloy. One

```
Controller
  isMain ?

Engine
  ctrl → Controller
    mainCtrl → isMain
```

Figure 17. Engine Clafer model

```
one sig Controller extends clafer
{ isMain : lone isMain }

lone sig isMain extends clafer {}
{ one r_isMain }

one sig Engine extends clafer
{ ctrl : one ctrl }

one sig ctrl extends clafer
{ ref : one clafer
, mainCtrl : lone clafer }
{ ref in Controller
  mainCtrl in Controller.r_isMain }
```

a) Extra reference signatures

```
one sig Controller
{ r_isMain : lone isMain }

lone sig isMain {}
{ one r_isMain }

one sig Engine
{ r_ctrl : one ctrl }

one sig ctrl
{ ref : one Controller
, r_mainCtrl : lone isMain }
```

b) Simplified referencing

Figure 18. Reference optimization

can notice that all signatures extend clafer, which is an abstract signature of all clafers. This solution enabled the translator to apply very simple translation rules. Whenever it encountered a reference clafer, it just pointed to a generic clafer type (e.g. ref and mainCtrl). Later it restricted the type by attaching constraints as the two last lines of Fig. 18a. There are two significant drawbacks of this translation. First, all signatures have to be subsets of clafer. In practice, it required increasing the scope analysis in Alloy and allowed reasoning only over models with several elements. Second, constraints are used for restricting types. Effectively, the Alloy Analyzer was using SAT solver to do type-checking, which is redundant and slows down model analysis.

The optimized translator solves both problems. In the first place, proper name resolution algorithm eliminated the need for the abstract clafer signature. A quick look at Fig. 18b shows that none of the signatures extends clafer. Thanks to it, the Alloy Analyzer can efficiently handle models with thousands of features. Additionally, the translator makes better use of Alloy's type system to specify references. Both ref and r_mainCtrl are specified as relations with well defined types. It results in fewer constraints passed to the SAT solver and better reasoning efficiency.

3.8 Parameters and Model Statistics

The old release of clafer2alloy translator took a Clafer model and generated Alloy model. It had no configuration options and printed only error messages. Users of the application had no control over behavior of the tool. It prevented them from leveraging their knowledge to speed up the translation process. Then the translation could take quite a time for huge models.

In the new translator end-users may manually turn on and off certain features. All of the previously described optimizations are applied automatically and they usually do no harm. Even in the worst case the time of analysis is shorter. There are two exceptions, though. Removal of dead abstract clafers and inheritance flattening might result in models that are hard or impossible to analyze.

Although, by default the translator applies both optimizations when possible, user can turn them off.

In Clafer it is impossible to have an instance of abstract signature without extending it by a concrete clafer. The translator maps abstract clafers to abstract signatures in Alloy. In Alloy, instances of abstract signatures are legal if the signature is not extended by any other signature. For this reason, users might want to depend on this "hackish" behavior of Alloy when generating model instances. It frees them from defining concrete instances of abstract clafers when those clafers are referenced (that point was discussed in Sect. 3.3).

Inheritance flattening is a very useful option that helps to keep the Alloy scope low. It is especially useful for meta-models because reduces the reasoning time. The side effect of inheritance flattening is potentially significant grow of Alloy models (in terms of lines of code). Those models are easy to analyze for the SAT solver, but it takes more time for the Alloy Analyzer to generate 3CNF formulas for them. In extreme cases size of the model may explode, and might be too large to translate it to a 3CNF formula in a reasonable time.

The tool also contains other parameters that allow user to set options of its internal modules, such as layout resolver and name resolver. Clafer models use code indentation to indicate parent-child dependency among clafers. Layout resolver inserts braces where necessary to make the model parsable. One can turn off the resolver to skip the phase of inserting missing braces. Furthermore, the name resolver applies several strategies to disambiguate names. When user knows that all the names are unique, that can save a huge amount of time when translating models with thousands of elements. The problem is rather technical and is a limitation of current implementation.

Finally, after performing the translation, the translator shows a short summary of the model. The summary shows statistics of the model, such as the number of abstract, reference clafer, the number of constraints, information about ambiguous names, and also minimal scope of analysis. The last parameter might be very useful to set the least possible scope in Alloy for analyzing the model. The number is imprecise, since the scope may be influenced by constraints. In practice the estimation returned by the translator is fairly good and saves time on finding the optimal scope (which must be done manually).

4. Evaluation

The new Clafer translator was evaluated on a variety of models: feature models, meta-models, and feature-based model templates (FBMTs). We believe that end-users may perform complex analyses on rich Clafer models in a reasonable time. We reevaluated the translator following the same methodology as in [4]. In fact, we take our clafer2alloy implementation that was evaluated in [4] as a baseline. In that implementation we optimized Alloy models manually. Therefore, the current evaluation compares the new Clafer translator with manually optimized models generated by the old translator. Comparison of the new translator with the old one (without all automatic optimizations) would yield much better improvements but would not be totally fair. The old translator had several versions that had hard-coded optimizations for specified types of models.

The methodology for our experiment is the following. First, we selected representative models, and selected representative analyses. Next, all models were translated to Clafer from other formats. Finally, we executed the Clafer translator, run the analyses in the Alloy Analyzer, and reported performance results. We did the experiment on a laptop with Core Duo 2 @2.4GHz processor and 2.5GB of RAM, running Linux. The Alloy Analyzer was configured to use Minisat as a solver. The subsequent sections present and discuss the results for the three subclasses of models.

4.1 Feature Models

SPLOT [15] is a popular repository of Boolean feature models. We automatically translated to Clafer 58 models created by humans (as of July 4th, 2010). The human-made models were rather small in size; they contained up to hundreds of features. To test the translator and efficiency of reasoning we also included machine-generated models with thousands of features and hundreds of constraints. For each model we checked its consistency, by finding a valid instance. All of the models were consistent.

Table 1 summarizes the results. Human-made models were analyzed within the range of tens milliseconds. For those models the new version of Clafer translator shortened the reasoning time 2 to 5 times. For larger, machine-generated, models the reasoning speedup was around the factor of 2. Boolean feature models benefited mainly from optimization of hierarchical constraints, and from finding global cardinality for each feature.

When doing the experiments we noticed that the Alloy Analyzer performs reasoning in several stages. First, it translates the model into a 3CNF logical formula, and then the formula is passed to a SAT solver. The actual reasoning takes up to hundreds of milliseconds for SAT solver for the largest models. Rest of the time is spent on translation to 3CNF formula. Probably the Alloy Analyzer was optimized for handling other kinds of models: those that are small, but still complex. Ours were rather large but used only propositional constraints.

4.2 Meta-Models

The Ecore Meta-model Zoo (www.emn.fr/z-info/atlanmod/index.php/Ecore) is a public repository of meta-models. UML2 is the largest and the most complex meta-model in the repository. Unfortunately, none of the meta-models contained OCL constraints. For UML2 we decided to extract OCL constraints from the UML specification [16] and to manually add them to the Clafer encoding of UML2. The use of constraints is what makes reasoning harder.

We performed automated analyses on slices of the UML2 meta-model: Class Diagram from [7], State Machines, and Behaviors (Tab. 2). Each slice has between 10 and 20 classes. When selecting slices we were looking for elements with a variety of OCL constraints. We checked the *strong consistency* property [6] 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 analysis confirmed that none of the meta-models had dead elements.

Meta-models are fairly rich models, i.e. they use abstract classes, inheritance, references, primitive types, constraints. A wide range of Clafer constructions allowed to apply all Clafer optimizations. After applying those optimizations reasoning time in the Alloy Analyzer was 3 to 4 times shorter. For the evaluated slices it dropped from hundreds to tens of milliseconds.

4.3 Feature-Based Model Templates

A FBMT combines feature with meta-models and contains a mapping to couple the two kinds of models. Configuration of a FBMT is automatically reflected in configuration of class/meta-models. For analysis the mapping was established by means of logical constraints. To the best of our knowledge, Electronic Shopping [14] is the largest example of a model template found in the literature. We used two its templates of activity diagrams (listed in Tab. 3) for evaluation: FindProduct and Checkout. Furthermore, Telematics is a model template of a car information system. Although it is relatively small, it uses a wide range of Clafer constructions. Each template had substantial variability in it. All templates have be-

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

				running time		
model name	**nature**	**[# features]**	**[# constraints]**	**old [s]**	**new [s]**	**improvement [×]**
Digital Video System	Realistic	26	3	0.012	0.003	4.0
Dell Laptops	Realistic	46	110	0.025	0.007	3.6
Arcade Game	Realistic	61	34	0.040	0.008	5.0
eShop	Realistic	287	21	0.15	0.077	1.9
FM-500-50-1	Generated	500	50	0.45	0.212	2.1
FM-1000-100-2	Generated	1000	100	1.5	0.731	2.1
FM-2000-200-3	Generated	2000	200	4.5	2.300	2.0
FM-5000-500-4	Generated	5000	500	28.0	15.0	1.9

Table 2. Results of strong consistency analysis for UML2 *meta-model* slices in Clafer

				running time		
meta-model/instance	**size**	**[#classes]**	**[#constraints]**	**old [s]**	**new [s]**	**improvement [×]**
State Machines		11	28	0.08	0.029	2.8
Class Diagram		19	17	0.15	0.037	4.0
Behaviors		20	13	0.23	0.057	4.0

tween 10 and 20 features, tens of classes and from tens to hundreds constraints.

We performed two types of analyses on FBMTs. First, we created sample feature configurations and instantiated templates in the Alloy Analyzer. Next, we did element liveness analysis for each template. The analysis is similar to element liveness for meta-models [6], but now applied to template elements.

Table 3 summarizes the inspected models, times of analyses, and improvement factors. For the Telematics example the new Clafer translator allowed for analyses 5 times shorter. In the other two cases, the analyses took about half of the time required for the model generated by the old translator. Improvement factors for element liveness analyses were similar. It was anticipated since element liveness analysis is equivalent to performing several model instantiations.

4.4 Summary of the Experiment

Overall, the results look very encouraging. Although the Clafer model optimizations were relatively simple, they provided good results. Majority of the analyses times were a way below one second. That makes those analyses fast enough to apply them in end-user tools . There is still much room for improvement. Model creators have extra knowledge that is not fully used by the Clafer translator or the Alloy Analyzer. Further improvements should reduce the costs of translating Alloy models to 3CNF formulas.

Our current work was vulnerable to the same threats to validity as the previous experiment [4]. In addition, the new Clafer translator might have changed configuration semantics of input models. Change of semantics might have influenced reasoning times.

5. Related Work

There are several languages for variability modeling. There are also tools for automated analysis of various models. Most of the work has been done in the area Boolean feature models (without attributes and cardinalities). This section presents some of the results and relates them with Clafer.

SPLOT [15] is the biggest public repository of feature models. It supports the original FODA feature models; they have no attributes and features cannot be multiply instantiated. SPLOT uses BDDs to reason about the models and is able to find inconsistencies, detect dead features, count core features, and count the number of valid configurations. In contrast with Clafer, SPLOT does not need to recognize input models, since all of them are Boolean feature models. In terms of expressivity, SPLOT models are a small subset of what Clafer can represent. Real world variability models, such as the Linux kernel Kconfig model are much more complex than the original feature models [3]. They involve operations on integers, strings, which are unsupported by SPLOT.

TVL [8] is a textual variability modeling language that covers feature models with attributes. TVL is aimed at feature modeling, while Clafer models also capture structural variabilities. Besides attributes, Clafer models cover classes, and references to features and classes. At present, the tool support for TVL and Clafer is similar. TVL uses the Sat4J SAT solver to find a model instance. It can also list products derivable from the variability model. Clafer uses Alloy to perform the same operations. Additionally, the Alloy Analyzer can use unsat core to indicate contradicting constraints.

Gheyi et al. worked on translation of FODA feature models to Alloy. They provided two theories [11], one for checking properties of feature models, and another one for verification whether given model is a refactoring of the other one. The two theories in Alloy significantly differ from Clafer's translation. Gheyi's translation of FODA feature models utilizes only propositional formulas and Boolean variables, while Clafer introduces a new set (signature) for each feature. It is necessary to use sets to model features if the language supports feature cardinalities. Furthermore, their Alloy models generate a set of feature model instances; Clafer models generate only one instance. In practice, it is infeasible to generate all instances of variability models at once, since that number grows exponentially with the number of features.

Evolution of software product lines requires automated reasoning to relate an evolved SPL model with the original one. If establishing the relationship can be done efficiently, then we can do incremental analysis on the new SPL. Incremental analysis shall be much shorter than analyzing the new SPL from scratch. Thüm et al. presented an efficient algorithm that can determine whether the new feature model is a refactoring, specialization (has fewer features), generalization (has more features), or an arbitrary edit. Recent work by Borba et al. [5] describes a theory of software product line refinements. It gives theoretical basis for checking whether the new

Table 3. Analyses for *Feature-Based Model Templates* expressed in Clafer. Parentheses by the model names indicate the number of optional elements in each template.

		instantiation			element liveness		
FBMT	**#features/#classes/#constraints**	**old [s]**	**new [s]**	**improvement [×]**	**old [s]**	**new [s]**	**improvement [×]**
Telematics (8)	8/7/17	0.04	0.007	5.7	0.26	0.049	5.3
FindProduct (16)	13/29/10	0.07	0.041	1.7	0.18	0.080	2.3
Checkout (41)	18/78/314	1.6	0.734	2.2	5.8	2.52	2.3

SPL is a refactoring of the original SPL. Besides feature models, it considers also assets and mapping from feature models to assets. Clafer does not offer any reasoning on SPL evolution, but we leave it as a future work.

FAMA [2] is a framework for automated analyses of feature models. It covers feature models with attributes and uses various reasoners which are chosen based on type of analysis. FAMA allows to add new reasoners, new file formats of variability models, and new analyses. There is certain overlap between FAMA's functionality and the functionality we envision for Clafer. Both tools have very similar goals, therefore we might use FAMA to reason on a subset of Clafer models. Besides attributed feature models Clafer also supports classes. Consequently, constraints over Clafer models are more complex than FAMA's constraints. Clafer constraints operate on sets, while FAMA'c constraints include Boolean formulas and attribute expressions.

Common Variability Language (CVL) [17] is the upcoming OMG standard for variability modeling. In contrast with other work, CVL models are not self-contained. They introduce variability into existing models (such as UML) without modifying them. CVL models are slightly less expressive than Clafer models (they do not support references), but they have *choices*, *classifiers*, and *parameters* that correspond to clafers. The fundamental difference between Clafer and CVL is that Clafer unifies different kinds of variability, while CVL treats them as distinct concepts. The aim of CVL is to introduce and resolve variability in existing models; Clafer focuses on SPL analyses.

An important aspect of Clafer is its human readable and textual notation. It evolved from framework-specific modeling languages (FSMLs) proposed by Antkiewicz [1]. FSMLs define abstractions and rules of framework's programming interfaces. They are formalized in the form of feature models with *mapping definitions* that describe the correspondence between features and source code. Textual notation of FSMLs introduced code indentation to form tree hierarchy, it also specified feature and group cardinalities, and feature attributes. Clafer uses a subset of that notation, but it also extends the notation to capture some notions from class modeling (e.g. inheritance, references).

6. Conclusion

In this work we presented optimized translation of Clafer models to Alloy. We also described other improvements done to the Clafer translator. Most of the optimizations are relatively simple, but they make use of domain knowledge about variability models. It was equally important to learn about specifics of our back-end: the Alloy Analyzer. Our evaluation showed that we can perform complex analyses on a wide range of models in a reasonable time.

We acknowledge that there are more opportunities to improve the reasoning time over Clafer models. First, the translator itself is still a research prototype. Proper engineering may fix some technical issues here and there. Second, the Alloy Analyzer overhead may be reduced by translating Clafer models directly to KodKod (Alloy's engine) and thus skipping Alloy's front-end. Finally, one can translate models to SAT or SMT solvers to provide the best encoding and reduce all the overhead introduced by the tools in between.

In the future, we would like to provide translations to different back-ends, including KodKod, SMT solvers, and BDDs to perform the analyses more efficiently. We would also like to implement more analyses to support configuration completion to minimize the number of required configuration steps, to count the number of distinct products available in SPL, and to support reasoning on SPL evolution.

References

[1] M. Antkiewicz. *Framework-specific modeling languages*. PhD thesis, University of Waterloo, 2008.

[2] D. Benavides, P. Trinidad, and A. Ruiz-Cortés. Automated reasoning on feature models. In *CAiSE'05*, 2005.

[3] T. Berger, S. She, R. Lotufo, A. Wąsowski, and K. Czarnecki. Variability modeling in the real: a perspective from the operating systems domain. In *ASE'10*, 2010.

[4] K. Bąk, K. Czarnecki, and A. Wąsowski. Feature and meta-models in Clafer: mixed, specialized, and coupled. SLE'10, 2010.

[5] P. Borba, L. Teixeira, and R. Gheyi. A theory of software product line refinement. In *Theoretical Aspects of Computing - ICTAC 2010*, 2010.

[6] J. Cabot, R. Clarisó, and D. Riera. Verification of UML/OCL Class Diagrams Using Constraint Programming. In *MoDeVVA'08*.

[7] E. Cariou, N. Belloir, F. Barbier, and N. Djemam. Ocl contracts for the verification of model transformations. In *OCL workshop of MoDELS'09*.

[8] A. Classen, Q. Boucher, and P. Heymans. A text-based approach to feature modelling: Syntax and semantics of TVL. *Science of Computer Programming*, In Press, Corrected Proof, 2010.

[9] K. Czarnecki, T. Bednasch, P. Unger, and U. Eisenecker. Generative programming for embedded software: An industrial experience report. In *GPCE'02*.

[10] K. Czarnecki, S. Helsen, and U. Eisenecker. Formalizing cardinality-based feature models and their specialization. *SPIP*, 10(1), 2005.

[11] R. Gheyi, T. Massoni, and P. Borba. A theory for feature models in Alloy. In *First Alloy Workshop*, 2006.

[12] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[13] 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, CMU, 1990.

[14] S. Q. Lau. Domain analysis of e-commerce systems using feature-based model templates. Master's thesis, University of Waterloo, 2006.

[15] M. Mendonça, M. Branco, and D. Cowan. S.P.L.O.T. - Software Product Lines Online Tools. In *OOPSLA'09*, 2009.

[16] OMG. *OMG Unified Modeling Language*, 2009.

[17] OMG. *Common Variability Language (CVL) Request for Proposal*, 2009.

[18] 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.