Cardinality-Based Feature Modeling and Constraints: A Progress Report

Krzysztof Czarnecki, Chang Hwan Peter Kim
University of Waterloo
200 University Ave. West
Waterloo, ON N2L 3G1, Canada
{kczarnec,chpkim}@swen.uwaterloo.ca

ABSTRACT

Software factories have been proposed as a comprehensive and integrative approach to generative software development. Feature modeling has several applications in generative software development, including domain analysis, product-line scoping, and feature-based product specification. This paper reports on our recent progress in cardinality-based feature modeling and its support for expressing additional constraints. We show that the Object-Constraint Language (OCL) can adequately capture such constraints. Furthermore, we identify a set of facilities based on constraint satisfaction that can be provided by feature modeling and feature-based configuration tools and present a prototype implementing some of these facilities. We report on our experience with the prototype and give directions for future work.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements / Specifications—*Tools*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Computer-aided software engineering (CASE)*; D.2.13 [**Software Engineering**]: Reusable Software—*Domain engineering, Reuse models*

General Terms

Design, Documentation

Keywords

Feature modeling, model-driven development, product configuration, constraint satisfaction, software-product lines, variability management

1. INTRODUCTION

Generative software development [10, 4, 11] aims at automating product development in the context of software product line engineering. Software product line engineering [8, 26] eases the development of products within an application domain by leveraging the commonalities among these products while managing the differences among them in a systematic way. As a result, individual products can be created by reusing assets, such as models, components, platforms, generators, documentation, etc., that are created within a separate product-line development process. Generative approaches aim at automating the product development part of product-line engineering by means of a wide range of static and dynamic technologies, including metaprogramming, reflection, program and model analysis, constraint-based configuration, and aspect-oriented techniques.

Software Factories [16] can be viewed as a comprehensive and integrative approach to generative software development, as they both share the same goal of automating product development in the context of software product line engineering. A particular strength of the Software Factory approach is its support for different degrees of automation in product development. The approach recognizes that domain knowledge may exist at different maturity levels and thus a wide range of concepts, such as patterns, architectures, frameworks, components, aspects, and domain-specific languages, etc., may be required for adequately packaging the knowledge as reusable assets.

Feature modeling is a technique for managing commonalities and variabilities within a product line. The technique has numerous applications in the context of generative approaches. Among others, it can be used to

- capture the results of domain analysis;
- facilitate scoping of product lines, domain-specific language families, components, platforms, and other reusable assets; and
- provide a basis for automated configuration of concrete products, languages, components, platforms, etc.

There has been a significant amount of progress on the notation, processes and techniques, tools, and applications of feature modeling since feature modeling was first proposed by Kang et al. [17] (see [13, Section 2] for an extensive set of bibliographic references related to feature modeling).

In this paper we report on our recent progress on a particular form of feature modeling, which is referred to as *cardinality-based* [12], the means of expressing additional constraints for cardinality-based feature modeling, and constraint-satisfaction facilities for feature modeling and feature-based configuration tools.

The main contributions of this paper are the following.

1. We analyze what is required from a notation for expressing additional constraints on cardinality-based fea-

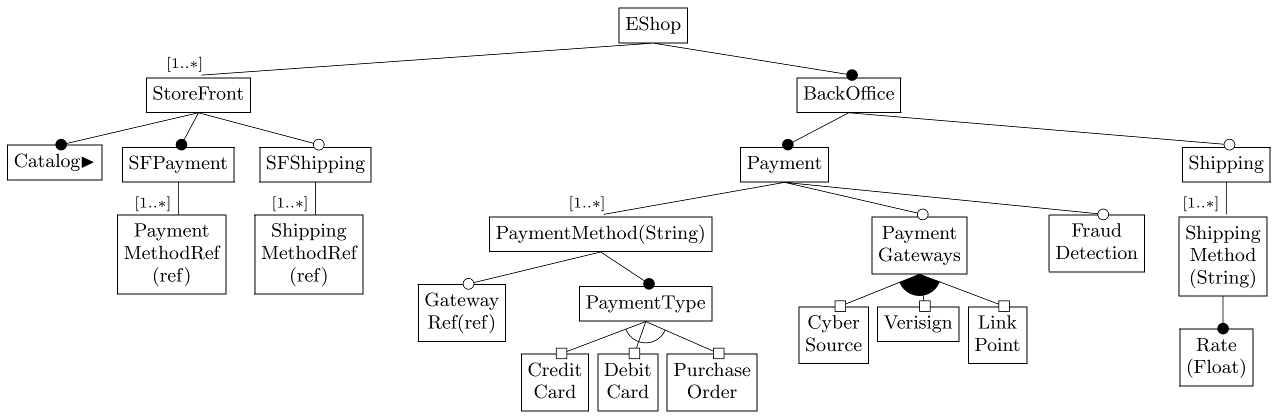


Figure 1: A sample feature model

ture models. We argue that Object Constraint Language (OCL) [20] is adequate for expressing such constraints and support our claim with a number of sample constraints.

2. We identify a set of facilities based on constraint satisfaction that can be provided by feature modeling and feature-based configuration tools.

3. We describe a prototype supporting several of such facilities based on Binary Decision Diagrams (BDDs) and report on our experience with the prototype. Of particular notice is the user interface of the prototype configurator, which visually distinguishes between user and machine configuration choices.

The remainder of the paper is organized as follows. Section 2 gives the necessary background information on cardinality-based feature models, including some recent changes to the notation. The special requirements of cardinality-based feature modeling on expressing additional constraints are discussed in Section 3. Section 4 give examples of additional constraints on cardinality-based feature models and shows that they can be adequately expressed in OCL. Section 5 classifies constraint satisfaction algorithms and feature modeling and feature-based configuration facilities based on such algorithms. A prototype implementing some of these facilities and our experience with the prototype are described in Section 6. Related work is discussed Section 7, and Section 8 closes with conclusions and direction for future work.

2. BACKGROUND: CARDINALITY-BASED FEATURE MODELING

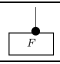
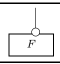
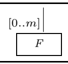
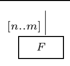
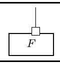
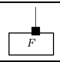
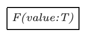
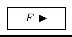
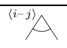
Cardinality-based feature modeling integrates a number of extensions to the original FODA notation. An example of a cardinality-based feature model describing a family of electronic shops is given in Figure 1. The notation is summarized in Table 1. A brief explanation of the notation follows.

A cardinality-based feature model is a hierarchy of features, where each feature has a *feature cardinality.* A feature cardinality is an interval of the form $[m..n]$, where $m \in \mathbb{Z} \wedge n \in \mathbb{Z} \cup \{*\} \wedge 0 \leq m \wedge (m \leq n \vee n = *)$. Feature cardinality denotes how many clones of the feature (with its entire subtree) can be included as children of the feature's parent when specifying a concrete configuration. Note that we allow a feature cardinality to have as an upper bound the Kleene star *. Such an upper bound denotes the possibility to take a feature an unbounded number of times. Features with the cardinality [1..1] are referred to as *mandatory*, whereas features with the cardinality [0..1] are called *optional*. For example, `Payment` is a mandatory feature, whereas `Shipping` is an optional feature. Features with a cardinality having an upper bound larger than one can be cloned during configuration. Features under a cloned feature can still be configured if they have variability. Cloning is useful if a configuration needs to include multiple copies of a part, where each part may be differently configured. For example, the configuration of an electronic shop may include multiple store fronts that may be configured differently, for example, by having different selections of payments and shipping methods, if any.

Additionally, features can be arranged into *feature groups*, where each feature group has a *group cardinality.* A group cardinality is an interval of the form $\langle m\text{–}n\rangle$, where $m, n \in \mathbb{Z} \wedge 0 \leq m \leq n \leq k$, where k is the number of features in the group. Group cardinality denotes how many group members can be selected. For example, at least and at most one of the features `CreditCard`, `DebitCard`, and `PurchaseOrder` must be selected as a subfeature of `PaymentType`.

A feature can have an *attribute type*, indicating that an attribute value can be specified during configuration (unless the value is already present). We allow at most one attribute per feature. If several attributes are needed, a set of subfeatures, where each subfeature has an attribute, can be introduced. The attribute type can be a basic type, such as *String* or *Integer*, or *FRef*, which denotes the set of all references to features in a given configuration. We also refer to an attribute with the type *FRef* as a *feature reference attribute*. For example, our sample feature model in Figure 1 uses string attributes to represent names of payment methods and shipping methods, and a float attribute to represent the rate of a shipping method. Furthermore, feature reference attributes are used in `PaymentMethodRef` and `ShippingMethodRef` to point to predefined payment and shipping methods in the back office, i.e., clones of `Payment-Method` and `ShippingMethod`. Although a feature reference can point to any feature in a configuration, we will later show how to restrict the attributes to point to any of the

Table 1: Symbols used in cardinality-based feature modeling

Symbol	Explanation
	Solitary feature with cardinality [1..1], i.e., *mandatory* feature
	Solitary feature with cardinality [0..1], i.e., *optional* feature
	Solitary feature with cardinality $[0..m]$, $m > 1$, i.e., *optional clonable* feature
	Solitary feature with cardinality $[n..m]$, $n > 0 \wedge m > 1$, i.e., *mandatory clonable* feature
	Grouped feature with cardinality [0..1]
	Grouped feature with cardinality [1..1]
	Feature F with attribute of type T and value of *value*
	Feature model reference F
	Feature group with cardinality $\langle 1\text{–}1\rangle$, i.e. *xor*-group
	Feature group with cardinality $\langle 1\text{–}k\rangle$, where k is the group size, i.e. *or*-group
	Feature group with cardinality $\langle i\text{–}j\rangle$

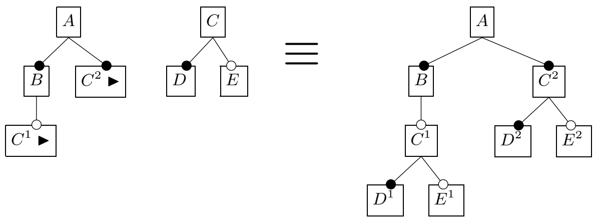
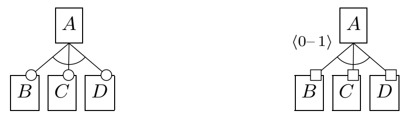


Figure 2: Unfolding of feature model references

clones of `PaymentMethod` or `ShippingMethod` using an additional constraint. Thus, feature reference attribute are particularly useful in the context of cloning.

Finally, a node in a feature model can also be a *feature model reference*. A feature model M with a reference to another feature model is semantically equivalent to a copy of M in which the reference has been unfolded, i.e., substituted by the feature model to which the reference points. Feature model references allow us to split large feature models into smaller modules. Also, a feature model can contain several references to the same sub-model, which is illustrated in Figure 2. We use subscripts in order to distinguish among the different feature copies created through unfolding. Our sample feature model in Figure 1 has one feature model reference, which is `Catalog`. For brevity, the feature model that the reference points to is not shown.

The example in Figure 1 shows the full repertoire of cardinality-based feature modeling. However, practical applications may depend on a subset of the notation. In particular, feature models describing high-level characteristics of product lines usually do not need feature cloning. Feature cloning is often useful in defining configurations of runtime components, policy profiles, and platform configurations. More



(a) Original cat-book notation

(b) New notation

Figure 3: A group of alternative features in the cat-book and the new notation

extensive discussion on the applicability of cardinality-based feature modeling is given elsewhere [13, Section 6].

It is interesting to note that, in contrast to our previous work [12], we now also associate feature cardinalities with grouped features [18]. However, the only possible values are [0..0], [0..1], and [1..1]. Normally, a grouped feature has [0..1] as its default cardinality. The cardinalities [1..1] and [0..0] are used for features that were selected or eliminated, respectively, from a group during specialization. Furthermore, we use squares instead of circles for grouped feature cardinalities in order to avoid confusion with the earlier notation described in the "cat book" [11]. In the book, group cardinalities and grouped feature cardinalities were interpreted in sequence, but now, group cardinalities can be thought of as additional constraints. Figure 3 illustrates this difference by showing an example in the original cat-book notation and its equivalent in the current notation.

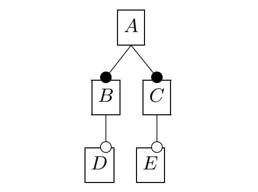
3. CARDINALITY-BASED FEATURE MODELING WITH CONSTRAINTS

Not all kinds of constraints within a feature model can be expressed using its tree structure and cardinalities. Such constraints need to be captured as *additional constraints*. Probably the best known examples of such constraints are *implies* and *excludes* constraints. Also, the addition of attributes prompts the need to define constraints over attribute values. Finally, cloning requires the notion of a context for scoping purposes and leads to constraints over sets of features.

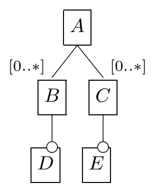
Let us first explain how cloning influences the set of facilities necessary for expressing additional constraints. For that purpose, consider the simple feature model in Figure 4(a). The constraint stating that feature D implies feature E could be simply expressed as

```
D implies E
```

Unfortunately, this constraint would not make much sense for the feature model in Figure 4(b). This is because the features B and C are now clonable, and a configuration conforming to this model may now contain multiple clones of the features D and E (since a feature is cloned with its entire subtree). An example of a constraint for the model in Figure 4(b) would be "the non-emptiness of the set of all D clones implies the non-emptiness of the set of all E clones". Furthermore, practical constraints (as we will see later) may require collecting clones of a given feature not in the entire feature model, but in the scope of a *context feature* (i.e., clones that are direct or indirect subfeatures of that feature).

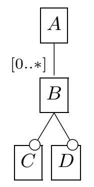


(a) No clonable features

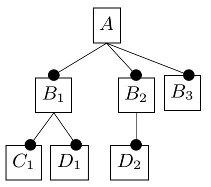


(b) Two clonable features (B and C)

Figure 4: Sample feature models without and with clonable features



(a) Feature model with a local constraint: C implies D in the context of B



(b) Sample configuration

Figure 5: Sample feature model with a local constraint and its configuration

The notion of a context feature is illustrated in Figure 5. The constraint that we would like to express for the feature model in Figure 5(a) is that for every clone of B it is guaranteed that if the clone has C as a subfeature, it also has D as a subfeature. In other words, the constraint is expressed in the context of B:

```
context B:
C implies D
```

Figure 5(b) shows a configuration conforming to the feature model in Figure 5(a) that also satisfies the above constraint. Subscripts were applied to the names of clones in Figure 5(b) in order to uniquely identify them.

Interestingly, additional constraints for cardinality-based feature models can be adequately expressed using Object Constraint Language (OCL) [20] or XPath [27]. Examples of additional constraints expressed using XPath are given elsewhere [2]. In this paper, we explore using OCL for this purpose. An important advantage of OCL over XPath is that modelers are more likely to be familiar with OCL than XPath. Other advantages include the fact that OCL has formally defined semantics [20, Appendix A] and that the template interpretation of OCL described in a previous paper [14] can be used to give formal semantics to OCL in the context of cardinality-based feature modeling.

4. OCL AS A CONSTRAINT LANGUAGE FOR CARDINALITY-BASED FEATURE MODELS

OCL can be adopted for feature modeling by interpreting feature models as class models with a containment hierarchy. For example, Figure 6 shows a UML class model corresponding to the feature model in Figure 1. The multiplicity at the aggregate end of every composition in that model is assumed to be 1. The multiplicities at the other ends are the same as the corresponding cardinalities in the feature model. Furthermore, `FRef` is an interface that every class representing a feature realizes. The group cardinalities from the feature model can be expressed as additional OCL constraints on the class model. Semantically, the class model represents a set of object structures which obey the multiplicities and the additional OCL constraints. Each of these object structures corresponds to a configuration of the feature model.[1]

Let us take a look at a number of sample additional constraints on the feature model in Figure 1 expressed in OCL.

The first example is a simple *implies* constraint that does not involve any clonable features.

```
context Payment inv:
  FraudDetection.isSelected() implies PaymentGateways.isSelected()
```

The constraint states that selecting `FraudDetection` implies that `PaymentGateways` is also selected. The constraint is stated as an OCL invariant in the context of `Payment`. The occurrences of `FraudDetection` and `PaymentGateways` denote navigation. According to an OCL convention, the names of the classes being the targets of the navigation can be used instead of role names if the latter are not available (which is the case in Figure 6). Furthermore, we assume that `isSelected()` is provided as a synonym for the standard OCL operation `isDefined()`. Please note that the same constraint can also be expressed using `EShop` as the context:

```
context EShop inv:
  BackOffice.Payment.FraudDetection.isSelected() implies
  BackOffice.Payment.PaymentGateways.isSelected()
```

As a second example, we would like to express that the reference attribute of `GatewayRef` should refer only to a payment gateway, i.e., any of the subfeatures of `PaymentGateways`:

```
context GatewayRef inv:
  EShop.BackOffice.Payment.PaymentGateways.sub()
    ->includes(att)
```

The above constraint is stated in the context of `GatewayRef`, meanig that the body of the constraint must hold for any instance of `GatewayRef`. In our case, the body must hold only if `GatewayRef` has been selected (i.e., an instance of it exists in the object structure). We assume that the operation `sub()` returns all subfeatures of a feature (i.e., all objects

[1]At this point, the reader may wonder about the differences between feature models and class models. There are at least three main differences. Firstly, feature models can be seen as a restricted form of class models. In particular, features do not have operations, composition has always the multiplicity of 1 at the aggregate end, and relationships such as inheritance and associations are not available. Secondly, feature modeling offers syntactic sugar. In particular, feature modeling has convenience mechanisms such as feature groups, which would have to be expressed as OCL constraints in a class model, and a more concise concrete syntax. The tree structure can also be exploited by tools to provide automatic layout and outlining. Thirdly, feature modeling comes with its own modeling philosophy. Feature models represent hierarchies of properties, which is rather unusual for class modeling.

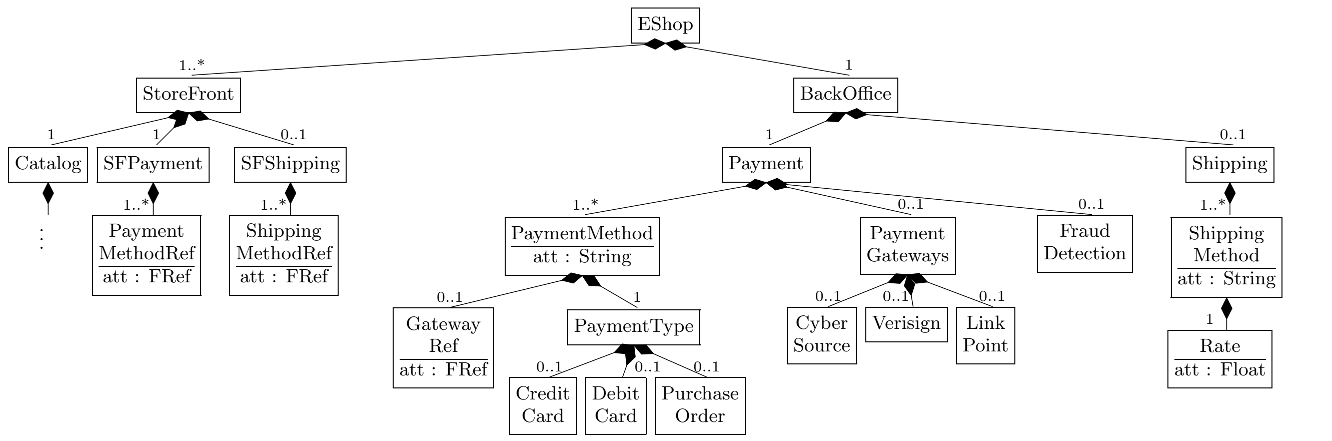


Figure 6: UML class model corresponding to the feature model in Figure 1

contained in the target of the operation). The right arrow is used in OCL to invoke operations on collections. In our case, we check whether the value of `GatewayRef`'s attribute `att` is included in the subfeatures of *PaymentGateways*.

Similarly, we want to enforce that payment methods available in a given store front are a subset of the payment methods provided by the back office:

```
context PaymentMethodRef inv:
  EShop.BackOffice.Payment.PaymentMethod->includes(att)
```

The above constraint assumes that the root feature `EShop` can be used as a constant denoting the sole instance of `EShop`, i.e., it is a syntactic sugar for `EShop.allInstances()`. Since the upper bound of the cardinality of `PaymentMethod` is higher than one, the result of the navigation `EShop.BackOffice.Payment.PaymentMethod` is a collection.

An analogous constraint is needed for shipping methods:

```
context ShippingMethodRef inv:
  EShop.BackOffice.Shipping.ShippingMethod->includes(att)
```

The next constraint is an example of an implies constraint within the context of a clonable feature (similar to the example in Figure 5). It states that every payment method of the type debit or credit card must have a specified gateway:

```
context PaymentMethod inv:
  (PaymentType.CreditCard.isSelected() or
  PaymentType.DebitCard.isSelected()) implies
  GatewayRef.isSelected()
```

As an example of a constraint on the size of a set, let us restrict the number of payment methods that are available in a store front to 3:

```
context StoreFront inv:
  SFPayment.PaymentMethodRef->size() <= 3
```

Finally, we give two examples of constraints involving numeric attribute values. The first constraint states that a shipping rate cannot be a negative number:

```
context Rate inv:
  att>=0
```

The second constraint states that if an electronic shop has shipping, at least one shipping method has a rate which is a positive number (i.e., at least one shipping method is not free):

```
context Shipping inv:
  ShippingMethod.Rate.att->exists(x| x>0)
```

Now we are ready to look at how the group cardinalities from the feature model can be expressed as additional OCL constraints on the class model. For example, the or-group under `PaymentGateways` can be represented by the following constraint:

```
context PaymentGateways inv:
  let
    numOfSelectedFeatures =
      CyberSource->union(Verisign->union(LinkPoint))->size()
  in
    numOfSelectedFeatures >= 1 and
    numOfSelectedFeatures <= 3
```

5. TOOL SUPPORT FOR FEATURE MODELING WITH CONSTRAINTS

A feature model with additional constraints can be translated into a set of variables and set of constraints over these variables. The variables represent the possible individual choices, such as feature selections and attribute value assignments, and the set of constraints includes both the constraints implied by the feature hierarchy and the additional constraints. A value assignment that satisfies the constraints corresponds to a correct configuration. A value assignment that assigns a value to every variable in the variable set corresponds to a full configuration. A partial assignment corresponds to a partial specialization [12], which is sometimes also referred to as a partial configuration.

The area of Constraint Satisfaction offers numerous algorithms that are of interest to feature modeling and feature-based configuration tools. The main algorithm classes of interest include:

- *Constraint checking*: Checking whether a particular (complete or partial) variable assignment satisfies the constraints.
- *Constraint propagation*: Infering the values of undecided variables from the values of decided variables.
- *Constraint satisfiability*: Checking whether a set of constraints has at least one solution.
- *Constraint solving*: Computing solutions for a set of constraints.

- *Computing the number of solutions*: Computing the number of complete solutions for a set of constraints.

A feature modeling tool can offer facilities based on constraint satisfaction algorithms to help the developer create correct feature models. Examples of such facilities are

- *Model consistency checking*: Constraint satisfiability allows determining that the feature model has at least one correct configuration.

- *Detecting anomalies*: Anomalies such as "dead features", i.e., features that are not part of any correct configuration and thus cannot be selected [24], can also be detected through constraint satisfiability. A feature is dead if the conjunction of the variable corresponding to the feature and the constraints implied by the feature model is not satisfiable.

- *Computing metrics*: The number of correct configurations is a useful metric indicating the *degree of variability* embodied in a feature model [15, 25]. It can be computed by calculating the number of solutions to the set of constraints implied by the feature model (including the additional constraints). Another example is *degree of orthogonality*, which is the ratio between the number of solutions to the set of constraints implied by the feature model (including all additional constraints) and the number of solutions to the set of constraints implied by the feature hierarchy and additional constraints that are local (i.e., involve just one feature). A high degree of orthogonality means that decisions about feature selections can be done locally, i.e., without worrying about their influence on choices in other parts of the feature hierarchy.

Constraint satisfaction algorithms are particularly of interest for feature-based configuration tools. Examples of facilities based on constraint satisfaction in feature-based configuration are

- *Configuration checking*: Checking that a complete configuration satisfies a set of constraints is a relatively simple and inexpensive operation amounting to expression evaluation. Checking a partial configuration is more expensive: it means checking to see whether the partial configuration is implied by the feature model, i.e., it requires checking constraint satisfiability.

- *Showing the number of remaining configurations*: Computing the number of configurations after every configuration step gives the user a better idea about the progress of the configuration process. If the number is one, the user arrived at a complete configuration. If the number is zero, the user has just created an incorrect configuration and will need to revise some of the configuration steps.

- *Propagating configuration choices*: Selecting a particular feature may require selecting some other feature(s), in which case the latter selection(s) can be inferred automatically from the first one using constraint propagation. In general, a configuration choice that assigns a particular value to a variable may restrict the domains of available values for other variables.

- *Auto-completion of configurations*: Constraint solving can be used for computing solutions (including completing partial solutions) that can be presented to the user.

- *Debugging incorrect configurations*: In some cases, the user may wish to quickly make several choices without verifying the number of remaining configurations along the way. This is particularly the case for an expert user. The resulting configuration may not be correct, in which case it needs to be fixed. While some of the user choices will have to be revised, the user may care about keeping some choices more than keeping some other choices. Thus, we need efficient ways of configuration fixing, which takes user preferences into account.

Auto-completion and debugging are probably the most challenging facilities to provide. Adequate solutions may require weak constraints, heuristics, and custom solution procedures.

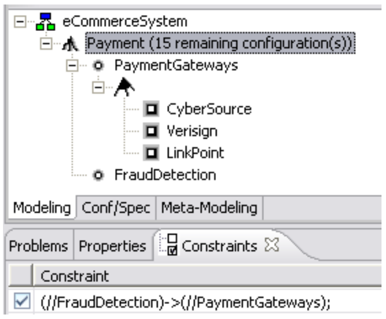
6. TOOL PROTOTYPE

We have created a prototype extending *fmp* [2] with some of the facilities based on constraint satisfaction discussed in the previous section. *fmp* is an Eclipse plug-in for feature modeling and feature-based configuration. In particular, we have extended *fmp* with the following facilities.

- *Computing the number of configurations represented by a feature model*: This facility allows the user to verify that a model has at least one configuration, and it also gives the user an idea about the degree of variability embodied in the feature model. Figure 7 shows a sample feature model with the number of concrete configurations it represents.

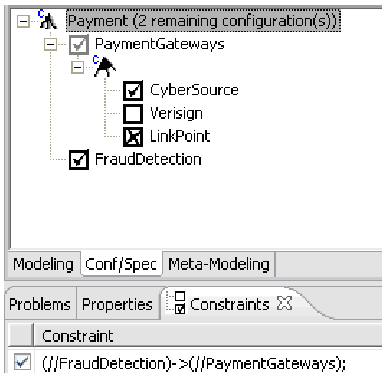
- *Propagating configuration choices*: Figure 8 shows a screen shot of the configuration interface, which renders the feature hierarchy with a check box for every variable feature. The check box can have one of five states: *undecided*, *user-selected*, *machine-selected*, *user-eliminated*, and *machine-eliminated*. Undecided is rendered as an empty check box and means that the user has not decided whether this feature should be selected or eliminated. The user can explicitly select or eliminate a feature, which is rendered by a dark check or cross, respectively. A user choice may trigger machine choices by means of propagation. Features selected or eliminated by the machine are rendered as light checks or crosses, respectively. For example, Figure 8 shows the result of the user selecting feature `FraudDetection` in a configuration where all check boxes were initially empty. Since `FraudDetection` implies `PaymentGateways`, choice propagation has the effect of placing a light check on `PaymentGateways`. The user then decided to select `CyberSource` and eliminate `LinkPoint`, while leaving `Verisign` still undecided. Using a different rendering for machine choices allows the user to better understand the effect of choice propagation.

- *Computing the number of remaining configurations during the configuration process*: The number of remaining configurations is the number of concrete configurations represented by the current (potentially partial)



(//FraudDetection)->(//PaymentGateways);

Figure 7: Sample feature model in fmp with constraint support



(//FraudDetection)->(//PaymentGateways);

Figure 8: Sample configuration in fmp with constraint support

configuration. This number may give the user an idea about the progress of the configuration process. For example, the partial configuration in Figure 8 represents two concrete configurations, meaning that further configuration choices are needed if the user wishes to arrive at a single complete configuration. The number also informs the user when the partial or complete configuration being constructed becomes incorrect, in which case the number of remaining configurations becomes zero.

- *Auto-completion of configurations*: After making zero or more choices, the user may request the tool to complete the configuration such that the configuration conforms to the feature model with its additional constraints. The user may ask the tool to show up to a user-specified number of correct solutions. The list of alternative solutions can be inspected in a solution browser and the desired solution can be selected.

The current prototype supports constraints only in the form of propositional formulas over features, and it assumes feature models without clonable features. The feature hierarchy is translated into a propositional formula according to the approach given by Batory [3].

The required constraint satisfaction machinery, namely computing the number of solutions, constraint propagation, and constraint solving, is based on Binary Decision Diagrams (BDD) [1]. The particular BDD package used in the current prototype is from Configit Software [9].

The performance afforded by the underlying BDD algorithms is excellent: the time required for counting solutions, propagating choices, and computing solutions is hardly noticeable by the user. We have tested our prototype with models containing up to several hundred features. However, since the number of features equals the number of variables of the corresponding constraint satisfiability problem and current BDD packages are known to handle problems with tens of thousands of features efficiently, we do not expect any scalability problems in terms of performance.

Although the visual distinction between user and machine choices has shown to be useful, our initial experiments have shown that better facilities for reviewing the effect of choice propagation are needed. We are currently working on a user interface that allows the user to browse through the effects of propagating a user choice and to revoke the choice if desired. Also, the auto-completion facility needs further development. In particular, we want to allow the feature model designer to define weak constraints which can compute appropriate default completions based on the user choices at a given point during the configuration process.

7. RELATED WORK

Related work for cardinality-based feature modeling can be found in an earlier paper [13]. Here we only focus on expressing constraints for cardinality-based feature modeling and constraint-based tool support for feature modeling and feature-based configuration.

The use of XPath for expressing constraints for cardinality-based feature modeling has been proposed in the context of two feature modeling tools, namely XFeature [7] and an earlier version of fmp [2]. In terms of constraint satisfaction, both of these tools can check whether a concrete configuration is correct with respect to the XPath constraints, which is done by simply invoking an XPath evaluator on an XML representation of the configuration. However, more sophisticated facilities such as choice propagation or auto-completion are not supported by these tools. While the paper by Cechticky et al. [7] gave one example of a constraint involving clonable features, we are not aware of any previous work analyzing the special needs of cardinality-based feature modeling in terms of expressing additional constraints as given in Section 3. The use of OCL as a language for expressing additional constraints for feature models was previously explored in a Diploma thesis by Schilling [21] under the supervision of the first author and by Streitferdt et al. [22].

An example of work on feature-based configuration supporting constraint satisfaction facilities beyond simple configuration checking is that of Batory [3]. Batory discusses the use of SAT solvers for checking consistency of a feature model and the use of constraint propagation algorithms to propagate configuration choices. Benavides et al. present similar ideas [5]. Certain versions of the feature modeling and configuration tool *pure::variants* [6] also support choice propagation. Similar to our prototype described in Section 6, the constraint satisfaction facilities in the above works do not consider clonable features.

Von der Maßen and Lichter [25] proposed a way to calculate a rough approximation of the number of correct complete configurations. We use *SAT count*, an existing BDD-based algorithm [1], which can compute the exact number of such configurations for large feature models within a fraction of a second. Also, we are not aware of any other feature-based configuration tools showing the number of remaining configurations.

The use of a BDD package as constraint satisfaction engine for feature modeling and configuration is not new. An example of such approach is that by van der Storm [23]. A related approach using BDDs, although not directly based on feature modeling, is DESERT [19].

A large body of work dealing with constraint-based configuration exists in the area of Artificial Intelligence. We already discussed the related approaches from that are in a previous paper [13, Section 6]. Although these approaches are usually concerned with the configuration of physical products, many ideas can certainly be transported into the realm of software product lines. We expect more of such cross-disciplinary efforts in the future.

We are not aware of any other classification of constraint-satisfaction-based facilities for feature modeling comparable to the one given in Section 5.

8. CONCLUSIONS AND FUTURE WORK

In this paper, we analyzed the needs of cardinality-based feature modeling with respect to expressing additional constraints and demonstrated how these needs are satisfied by OCL. Furthermore, we classified the facilities that a feature modeling tool and a feature-based configurator can offer based on constraint satisfaction algorithms. Finally, we described a prototype implementation of some of these facilities and shared our experience with the prototype.

For future work, we plan to extend the constraint satisfaction facilities in our tool prototype to handle OCL constraints over the full cardinality-based feature modeling notation. This will be done by adapting our OCL template interpretation [14] for the purpose of feature modeling. Furthermore, we plan to improve the user interaction model and the user interface for feature-based configuration in order to allow the user to deal with large feature models more effectively.

9. REFERENCES

[1] H. R. Andersen. *Binary Decision Diagrams.* Department of Information Technology, Technical University of Denmark, Lyngby, Denmark, 1997. Lecture notes for 49285 Advanced Algorithms E97, `http://www.itu.dk/people/hra/notes-index.html`.

[2] M. Antkiewicz and K. Czarnecki. FeaturePlugin: Feature modeling plug-in for Eclipse. In *OOPSLA'04 Eclipse Technology eXchange (ETX) Workshop*, 2004. Paper available from `http://www.swen.uwaterloo.ca/~kczarnec/etx04.pdf`. Software available from `gp.uwaterloo.ca/fmp`.

[3] D. Batory. Feature Models, Grammars, and Propositional Formulas. Technical Report TR-05-14, University of Texas at Austin, Texas, Mar. 2005.

[4] D. Batory, R. Lopez-Herrejon, and J.-P. Martin. Generating product-lines of product-families. In *Automated Software Engineering, 2002. Proceedings. ASE 2002. 17th IEEE International Conference*, pages 81–92, 2002.

[5] D. Benavides, P. Trinidad, and A. Ruiz-Cortés. Automated reasoning on feature models. In *Proceedings of the 17th Conference on Advanced Information Systems Engineering (CAiSE'05), Porto, Portugal, 2005*, LNCS. Springer, 2005.

[6] D. Beuche. pure::variants Eclipse Plugin. User Guide. pure-systems GmbH. Available from `http://web.pure-systems.com/fileadmin/downloads/pv_userguide.pdf`, 2004.

[7] V. Cechticky, A. Pasetti, O. Rohlik, and W. Schaufelberger. XML-based feature modelling. In J. Bosch and C. Krueger, editors, *Software Reuse: Methods, Techniques and Tools: 8th International Conference, ICSR 2004, Madrid, Spain, July 5-9, 2009. Proceedings*, volume 3107 of *Lecture Notes in Computer Science*, pages 101–114, Heidelberg, Germany, 2004. Springer-Verlag.

[8] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns.* Addison-Wesley, Boston, MA, 2001.

[9] Configit Software. *Configit—Product Configuration Engine*, 2005. `http://www.configit-software.com/`.

[10] K. Czarnecki. Overview of Generative Software Development. In *Proceedings of Unconventional Programming Paradigms (UPP) 2004, 15-17 September, Mont Saint-Michel, France, Revised Papers*, volume 3566 of *Lecture Notes in Computer Science*, pages 313–328. Springer-Verlag, 2004. `http://www.swen.uwaterloo.ca/~kczarnec/gsdoverview.pdf`.

[11] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications.* Addison-Wesley, Boston, MA, 2000.

[12] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged configuration using feature models. In R. L. Nord, editor, *Software Product Lines: Third International Conference, SPLC 2004, Boston, MA, USA, August 30-September 2, 2004. Proceedings*, volume 3154 of *Lecture Notes in Computer Science*, pages 266–283, Heidelberg, Germany, 2004. Springer-Verlag.

[13] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged configuration through specialization and multi-level configuration of feature models. *Software Process Improvement and Practice*, 10(2):143–169, 2005. `http://swen.uwaterloo.ca/~kczarnec/spip05b.pdf`.

[14] K. Czarnecki and K. Pietroszek. Verifying feature-based model templates against well-formedness OCL constriants. Submitted for publication, 2005.

[15] U. Eisenecker, M. Selbig, F. Blinn, and K. Czarnecki. Feature modeling for software system families. *OBJEKTspektrum*, (5):23–30, 2001. special issue on product families (in German).

[16] J. Greenfield and K. Short. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools.* Wiley, Indianapolis, IN, 2004.

[17] 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, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, Nov. 1990.

[18] C. H. P. Kim and K. Czarnecki. Synchronizing cardinality-based feature models and their specializations. In *Proceedings of ECMDA'05*, 2005. `swen.uwaterloo.ca/~kczarnec/ecmda05.pdf`.

[19] S. Neema, J. Sztipanovits, and G. Karsai. Constraint-based design-space exploration and model synthesis. In *Proceedings of EMSOFT'03*, volume 2855 of *LNCS*, pages 290–305. Springer, 2003.

[20] OMG. *UML 2.0 OCL Specification*, 2003. `http://www.omg.org/docs/ptc/03-10-14.pdf`.

[21] D. Schilling. Konfiguration von komponenten in produktlinien. Diplomarbeit, DaimlerChrysler Research and Technology, Ulm, and Universität Paderborn, Paderborn, Germany, Sept. 2002. (in German).

[22] D. Streitferdt, M. Riebisch, and I. Philippow. Details of formalized relations in feature models using OCL. In *ECBS*, pages 297–304, 2003.

[23] T. van der Storm. Variability and component composition. In *Proceedings of ICSR8*, LNCS. Springer, 2004.

[24] T. von der Maßen and H. Lichter. Deficiencies in feature models. In *Proceedings of SPLC'04 Workshop on Software Variability Management for Product Derivation - Towards Tool Support*, 2004.

[25] T. von der Maßen and H. Lichter. Instantiating valid products from feature models. In *Proceedings of 9th International Software Product Line Conference (SPLC)*, LNCS. Springer, 2005.

[26] D. M. Weiss and C. T. R. Lai. *Software Product-Line Engineering: A Family-Based Software Development Process*. Addison-Wesley, Boston, MA, 1999.

[27] World Wide Web Consortium. *XML Path Language (XPath) 2.0*, 2005. `http://www.w3.org/TR/xpath20/`.