Modeling Variation Space of Tailored Messages

Kacper Bak and Rafael Lotufo

Generative Software Development Lab, University of Waterloo, Canada,
{kbak,rlotufo}@gsd.uwaterloo.ca

1 Introduction

Persuasive technology is a growing field of research that should be able to change both the way we interact with computers, mobiles and electronic systems in general, as well as influence changes to our lifestyle. Persuasive technology in healthcare, for example, is believed to have a great potential to radically change behavior of patients towards prevention and treatment, as well as access to information.

Systems that are able to interact with users through personalized messages are very important in this field, as personalization is believed to improve persuasiveness and effectiveness of rhetoric [17, 15]. Personalization requires the modeling of the user, and of the tailorable message. However, we have not seen a formal notation and semantics for these models. Evermore, the generation (instantiation) of personalized messages given the user configuration and the message model still has to be manually programmed for every new user and message model.

The general problem of modeling variability is well known in the field of automated software engineering, which tries to automate production and maintenance of large software systems. In practice, many of the systems provide similar functionality but can be tailored on demand. So called software product lines represent families of related, but customizable, software systems built from shared components. Is is easy to see that there is a direct correspondence between modeling software variability and tailored messages in persuasion.

Feature modeling is a standard technique used for capturing commonalities and variabilities in software product lines. Feature models are usually represented in various visual notations basing on FODA [14]. Typically they are depicted as tree structures with additional constraints specifying relationships between parent feature and its subfeatures.

In this work, we present how feature modeling can be used to formalize the user and message models of tailored messages. Furthermore, we present Clafer, a notation for representing these models, which which can be used to create an interpreter to generate personalized messages given any user and document model. We will also present an instance of the user and document models based on data that will be provided to us. Finally, we describe how to easily build a proof of concept tool that allows automatic generation of tailored messages given a user and document model, using existing infrastructure.

The paper is organized as follows. We introduce feature modeling and discuss its extensions in Sect. 2. We then formalize variation space of tailored messages in Sect. 3. We present Clafer in Sect. 4. Sample variation spaces are modeled in Clafer in Sect. 5 and Sect. 6. In Sect. 7 we give overview of existing feature modeling infrastructure. We discuss related work in Sect. 8 and conclude in Sect. 9.

2 Feature Models

Feature models were introduced by Kang [14] to represent and model software products available in a family of software products — a software product line — where products are represented by a set of features. A valid set of features in a software product line compose a valid product, also known as a *configuration*.

Feature is an abstract and overloaded term. For software product lines, we think the most appropriate definition is "an increment in program functionality" [13]. This definition can easily be used to describe features for other domains other than software. As an example, lets consider a product line of automobiles with the following features: manual transmission, automatic transmission, leather seats, heated seats, cd player, tape and radio. Each one of these features add some sort of "functionality" to the automobile product.

Given the above features, we shall now describe the available automobile products that a hypothetical manufacturer produces: automobiles may have either manual transmission or automatic transmission; leather seats and heated seats are optional; cd player and tape are optional; if cd player or tape is selected then the automobile also has a radio; if an automobile has heated seats then the transmission must be automatic.

The above textual description of the automobile product line is a set of constraints on the features, where only valid products — configurations — adhere to the constraints. Feature models are therefore a visual notation for formalizing these constraints. Figure 1 shows a feature model that represents the automobile product line.

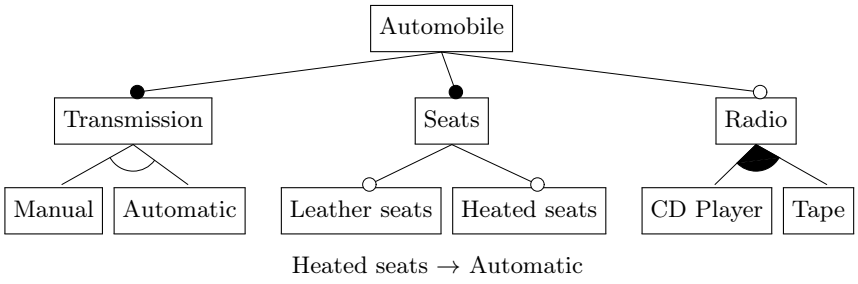


Fig. 1. Feature model for automobile product line

1 $\text{Transmission} \leftrightarrow \text{Automobile}$
2 $\text{Seats} \leftrightarrow \text{Automobile}$
3 $\text{Radio} \rightarrow \text{Automobile}$
4 $(\text{Manual} \vee \text{Automatic}) \wedge (\text{Manual} \rightarrow \neg\text{Automatic}) \wedge (\text{Automatic} \rightarrow \neg\text{Manual})$
5 $\text{Leather seats} \rightarrow \text{Seats}$
6 $\text{Heated seats} \rightarrow \text{Seats}$
7 $\text{CD Player} \vee \text{Tape}$
8 $\text{CD Player} \rightarrow \text{Radio}$
9 $\text{Tape} \rightarrow \text{Radio}$
10 $\text{Heated seats} \rightarrow \text{Automatic}$

Fig. 2. Propositional formulas for the automotive feature model

As can be seen in Fig. 1, feature models are represented by a *feature diagram*: a hierarchy of features and special links between features with different symbols. Both hierarchy and symbols add constraints to the feature model. Hierarchy specifies that for every feature that is selected (except the root feature), its parent must also be selected. For example, it is not possible to have leather seats without seats.

As for symbols, a white ball specifies that the feature is optional, as are leather seats and heated seats. Black balls specify that the feature is mandatory, i.e., a car must have transmission and seats. Arcs specify constraints on groups of features: white arcs are exclusive-or constraints, i.e., a transmission *must* be manual or automatic; black arcs are or groups, where *at least one* feature must be selected, i.e., the radio must have at least a cd player or tape, but can have both. Constraints that can not be represented in the feature diagram are specified as *extra-constraints*, as are given as propositional formulas together with the feature diagram. The constraint on heated seats is given as $\text{heated seats} \rightarrow \text{automatic}$, where the $\rightarrow$ symbol means *implies*.

As can be seen in the example, feature models represent constraints on a configuration in a simple and intuitive manner, instead of presenting this variability purely as a set of propositional formulas, or by a list of valid configurations. It is also evident that feature models are a general representation of valid configurations, and can be used to solve and model a wide variety of problems.

2.1 Feature Model and Propositional Formulas

As shown in [2] and [18], feature models can be represented as a propositional formula, where every feature is a variable in the formula such that only valid configurations will yield 'true'. Figure 2 shows the list of propositional formulas for the automotive feature model show in Fig. 1. The propositional formula for the feature model is the conjunction of all formulas.

Mandatory features are expressed by $\leftrightarrow$ between the parent and mandatory child feature as shown in lines 1 and 2 in Fig. 2. Hierarchy constraints on optional features are expressed using implication from child to parent, as shown by the formula on lines 3, 5, 6, 8 and 9. Line 4 shows how the propositional formula

for mutual exclusion (xor), and can be adapted to groups of any size. Or groups are formalized as disjunction of each feature in the group, as shown in line 7. Finally, the extra-constraints are appended to the list of formulas, as shown in line 10.

By transforming feature models to propositional formulas, it is possible to automate reasoning and formalize its semantics. BDDs and SAT solvers have been shown to provide good support for reasoning about feature models [18], and both require as input propositional formulas.

This kind of automation is useful both for feature model creation, as well as for configuration. When creating feature models, we want to make sure that there are no dead features (features that can never be selected), and also that the feature model allows all desired configurations and no more. This becomes a hard task as soon as feature models grow in size and number of constraints. As for configuration, we need tools that both assist users in configuration, such as with choice propagation, as well as makes sure constraints are followed.

2.2 Feature Diagrams and Grammars

Feature diagrams can also be represented as grammars [2]. Figure 3 shows an *iterative tree grammar* for the automotive feature model. In iterative grammars, repetition is expressed by `+` and `*`, for one or more, and zero or more constructs, respectively. Optional features are expressed as `[Radio]`, as in line 1 of Fig. 3. Mutual exclusion groups are expressed by using `|`, as shown in line 2. Or groups are written as `[CD_Player | Tape]+` as shown in line 4.

```
1  Automotive: Transmission Seats [Radio];
2  Transmission: Manual | Automatic;
3  Seats: [Leather_seats] [Heated_seats];
4  Radio: [CD_Player | Tape]+;
```

Fig. 3. Grammar for automotive feature model

By transforming feature diagrams to grammars it is possible to represent feature diagrams using textual notation. In Sect. 4 we will present *Clafer*, a grammar for feature models that is concise, expressive and intuitive, a great advance over grammars as shown in Fig. 3.

2.3 Extended Feature Models

Several researchers proposed extensions to feature models, so as to add more expressiveness to the feature diagrams. The two most prominent extensions are *cardinality-based* feature models and the addition of *attributes* to features.

A notation for cardinality-based feature models were proposed by Czarnecki in [9] and allow cardinalities to groups and individual features. Feature models

presented by Kang already have some cardinality: optional features have cardinality 0..1, whereas mandatory features have 1..1 cardinality. Cardinality-based feature models allow any cardinality to be specified for features and groups of features. For example, in the automobile product line a cardinality of 4..6 could be added to seats, specifying that automobiles should have from 4 to 6 seats. Figure 4 shows the extended version of the feature model for automobiles with cardinality constraint for the seat feature. To represent such constraints in traditional features models it would be necessary to clone the seat feature to 6 features, 4 of which would be mandatory and 2 would be optional. If the manufacturer would not like to impose a constraint on the maximum number of seats, the cardinality constraint can be written as 4..*, allowing 4 to an infinite number of seats. The semantics of * in cardinality constraints can not be expressed in traditional feature models. Clearly, adding cardinality to feature models adds expressiveness with a simple notation.

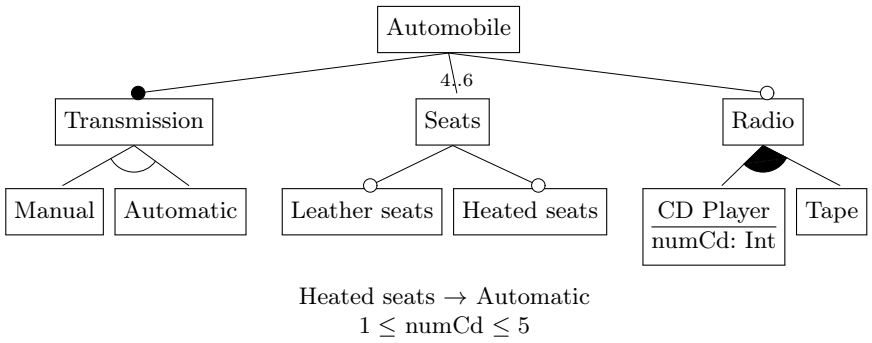


Fig. 4. Extended feature model for automobile product line

Attributes were proposed as an addition to feature models in [8, 3], also with the objective to add simple notation to feature models. Attributes in feature models are very similar to attributes in class diagrams in UML: features may have a number of attributes and attributes may have types, such as integers, strings, or even references to other features. In our example, we have added an attribute to CD Player that specifies the number of CDs that can be inserted into the device at the same time. This attribute is of type integer, and has a constraint specifying that it must be a number from 1 to 5. Figure 4 shows this extension to the automobile feature model. In a similar fashion to cardinality constraints, attributes can be modeled in traditional feature models by regular features. However, the feature model would explode in the number of features and would loose its desired visual conciseness.

3 Variation Space

Original feature models are unable to convey the full variation space of tailored messages. They are appropriate for describing variabilities but cannot express concrete data structures, such as text messages. In the context of variation space, there are conceptual differences between abstract variabilities and concrete data structures. We propose to split the variation space into *problem space* and *solution space*, and establish a *mapping* between them (see Fig. 5). Such a construction is well known in the SPL community, where separation of specification and implementation is the first step towards constructing a product line. It helped us to identify key domains and separate user features from the textual content of messages.

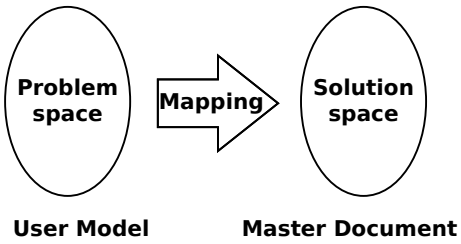
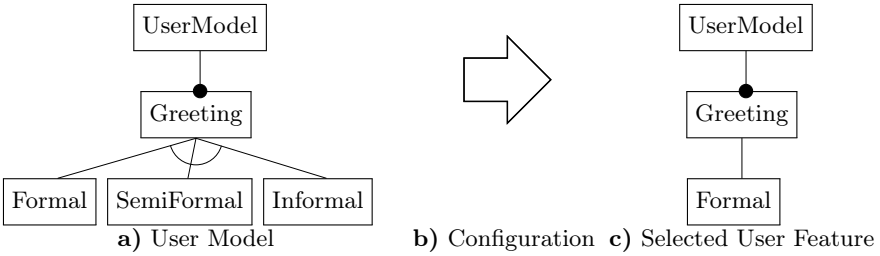


Fig. 5. Problem and solution spaces

The problem space contains *user model* to describe variations among people. We decided to apply feature modeling, as it naturally captures the required type of variability. Our model is very generic and represents as much user variability as is needed. To represent a single person, we configure the model, i.e. add constraints to either set or unset each feature. For example, Fig. 6a specifies sample user model, where user prefers only one greeting style. After picking a particular style, the model has no variability (see Fig. 6c) and becomes an instance of the feature model.



a) User Model **b)** Configuration **c)** Selected User Feature

Fig. 6. Configured user model

The solution space contains *master document* to represent structure of a generic message. The structure is mostly defined by feature model, but there are also solution-specific elements, such as the actual message content. For example, Fig. 7 shows a document with three possible greetings. Each greeting is a concrete text, e.g. "Dear". In the master document there are no group restrictions that make greetings mutually exclusive. These restrictions will be enforced by mapping between spaces.

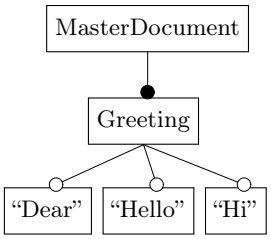


Fig. 7. Master document

The two spaces, although separate, cannot stay in complete isolation. We *couple* them by creating a mapping that expresses dependencies between user model and master document. It is defined by attaching cross-tree constraints to the feature model in Fig. 7. Cross-tree constraints are important part of feature modeling, but there is no standard notation for combining them with diagrams. We simply add a constraint enclosed in square brackets following the feature name/text in the box (see Fig. 8). In our example we refer to features from the user model, therefore they are preceded by a path (as the feature Formal is preceded in UserModel.Greeting.Formal). The *requires* constraint in "Dear" [*requires* UserModel.Greeting.Formal] says that presence of "Dear" implies presence of the UserModel.Greeting.Formal feature.

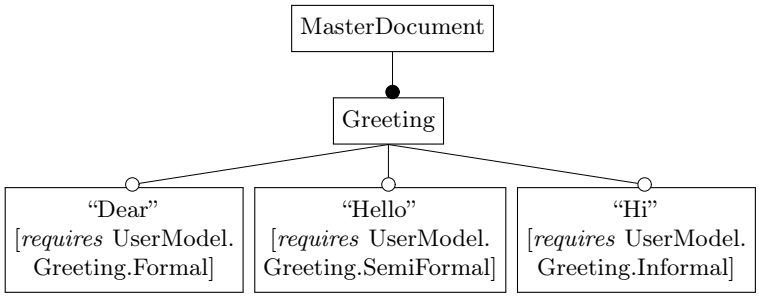


Fig. 8. Master document with mapping constraints

To sum up, we described the variation space by decomposing it into smaller subspaces and providing a mapping. Our approach elegantly couples the two spaces by enforcing dependencies among features and the actual texts. The mapping is precise enough to automatically configure the solution space by configuring the problem space. For example, if user prefers formal greeting, then the tailored message will start with "Dear". All these conceptual constructions can be concisely expressed in Clafer.

4 Clafer

Clafer is a general-purpose modeling language. Initially, we designed it as a convenient notation for expressing various aspects of software product lines, such as commonalities, variabilities, constraints and architectural models. Clafer captures abstractions from class and feature modeling, and provides a lot of syntactic sugar for common constructions. The language is not limited to software-related problems; technically, it can express any statement from the first-order predicate logic.

Generality of the language makes it suitable for modeling variation space of tailored messages. The problem we solve is not very different from our work on software product lines. The only difference lies in *what* we model, but many abstractions are the same. In this work, we show a subset of Clafer that is powerful enough to automatically tailor messages to specific users.

Clafer is a textual language with formal semantics defined by translation to Alloy [12]. It allows us to use Alloy infrastructure, such as the Alloy Analyzer and SAT-solvers, to generate instances of Clafer models. Moreover, the textual representation of feature models, in contrast with graphical notations, requires only a text editor to create models.

4.1 Feature Modeling

Feature models in Clafer are very similar to cardinality-based feature models augmented with cross-tree constraints. We present the language by showing short code snippets and explaining their meaning.

Clafer's representation of the feature model from Fig. 6a is visible in Fig. 9a; it consists of hierarchically nested features. The top-most feature is named UserModel and has one child (Greeting). The language uses code indentation to indicate a block of subfeatures. Figure 9b contains the same model, but with expanded group and feature cardinalities, and hierarchy indicated by curly braces.

FODA or cardinality-based feature models traditionally distinguished grouping features from other features. That is, a feature has either a group cardinality or a feature cardinality attached to it. The two concepts are orthogonal in Clafer, thus each feature has both types of cardinalities.

Feature cardinalities restrict the number of instances of a feature. The cardinality is specified by an interval $m..n$, where $m \in \mathbb{N}, n \in \mathbb{N} \cup \{*\}, m \leq n$. Feature cardinality specification is optional and if present, follows the feature

name. Clafer provides syntactic sugar similar to syntax of regular expressions: `?` denotes 0..1; `*` denotes 0..∗; and `+` denotes 1..∗. No explicit feature cardinality denotes 1..1 (mandatory) by default, modulo four exceptions explained shortly. For example, Greeting (line 2) in Fig. 9a is mandatory.

```
1  UserModel
2    xor Greeting
3      Formal
4      SemiFormal
5      Informal
```

a) Concise notation

```
1  <0-*> UserModel 1..1 {
2    <1-1> Greeting 1..1 {
3      <0-*> Formal 0..1 {}
4      <0-*> SemiFormal 0..1 {}
5      <0-*> Informal 0..1 {}
6    }
7  }
```

b) Desugared feature model

Fig. 9. Feature model in Clafer

Group cardinalities constrain the number of child instances i.e., the instances contained by subfeatures. Group cardinality is specified by an interval $\langle m–n\rangle$, with the same restrictions on m and n as for feature cardinalities, or by a keyword: `xor` denotes $\langle 1–1\rangle$; `or` denotes $\langle 1–*\rangle$; `mux` denotes $\langle 0–1\rangle$; and `opt` denotes $\langle 0–*\rangle$; further, each of the keywords makes subfeatures optional by default. If any, a group cardinality specification precedes a feature name. For example, `xor` on Greeting (line 2) in Fig. 9a states that only one child instance of either Formal or SemiFormal or Informal is allowed. Because the three subfeatures have no explicit cardinality attached to them, they are all optional (cf. Fig. 9b). No explicit group cardinality stands for $\langle 0–*\rangle$, except when it is inherited as illustrated next.

4.2 Feature Reuse

Code reuse is one of the key concepts in software engineering as it improves programmer's productivity, code quality and flexibility. There are several ways in which the language can support code reuse, i.e. modularization, inheritance, conditional file inclusion, and code clones. Inheritance is at the heart of class modeling, and therefore is available in Clafer.

There is no notion of feature reuse in the original feature models. The only method was to copy a tree structure and paste into another model. Clafer overcomes this limitation by distinguishing *abstract* and *concrete* features. All the features presented so far were concrete features, because they contained the actual feature instances. Abstract features, on the other hand, define abstract types that can be reused (extended) by other features. Abstract features are never instantiated unless extended by a concrete feature. Furthermore, only abstract features can be extended by concrete features. Abstract features are always top-level definitions, thus it is impossible to nest them.

The syntax of abstract features is very similar to the example in Fig. 9, but such a feature is preceded by the abstract keyword, and has no feature cardinality. Figure 10a shows an abstract exclusive-or group of three subfeatures. The type definition is extended and instantiated in Fig. 10b. The ProgressDay feature inherits Goal's group cardinality and subfeatures. It also adds a new feature StrategyDays, which becomes the fourth alternative. Figure 10c depicts the structure of ProgressDay.

```
1 abstract xor Goal
2   Exceeded
3   Met
4   Failed
```

a) Abstract feature

```
1 ProgressDay extends Goal
2   xor StrategyDays
3     IncreaseDays
4     KeepDays
5     DecreaseDays
```

b) Feature inheritance

```
1 xor ProgressDay
2   Exceeded
3   Met
4   Failed
5   xor StrategyDays
6     IncreaseDays
7     KeepDays
8     DecreaseDays
```

c) All subfeatures

Fig. 10. Inheritance in Clafer

Clafer offers syntactic sugar for inheritance if the extending feature has the same name as its superfeature. It is known as *quotation*, because syntactically it is a name of abstract feature preceded by the left quote symbol (‘). The extending feature inherits all assets of an abstract feature. In this way, we create new features that, in fact, instantiate abstract features. The mechanism is useful to define a feature once and reuse it later. Figure 11a shows an example. There is an abstract feature color that is either green or blue or brown. The model contains two unrelated features: car and eye, and each of them has some color. The meaning of quotation is shown in Fig. 11b, where ‘color is expanded as color **extends** color. Although, it might seem confusing, the construction has a well-defined meaning: the left color is name of the new feature, the right color refers to the abstract feature.

4.3 Textual Content

Textual strings go beyond the original feature modeling notation. Combination of feature models with specific values is known as feature models with attributes. While feature models represent general concepts, strings store specific texts. Here we present two ways of including strings in Clafer models: 1) inline definition, 2) by quotation.

The inline definition is a string placed within double quotes (e.g. line 2 in Fig. 12a). In the example, the text is nested under parent feature Greeting and acts as a subfeature. Inline definition creates an implicit feature that stores the content. Lack of explicit name prohibits the programmer from referring to the

```
1  abstract xor color
2    green
3    blue
4    brown
5
6  car
7    'color
8
9  eye
10   'color
```
a) Quotation

```
1  abstract xor color
2    green
3    blue
4    brown
5
6  car
7    color extends color
8
9  eyes
10   color extends color
```
b) Desugared quotation

Fig. 11. Feature model in Clafer

```
1  Greeting
2    ''Dear''
```
a) Inline definition

```
1  abstract dear = "Dear"
2
3  Greeting
4    'dear
```
b) Quotation

Fig. 12. Textual content in Clafer

feature in constraints. The inline method is preferable if the text is short and is not meant to be reused or constrained.

The second method, quotation, makes textual content more explicit, allows for more expressiveness and is better suited for long texts. As with inheritance, first we need to define the abstract feature, and then instantiate it. In the example from Fig. 12b, line 1 names the textual string by creating an abstract feature. Then the feature is instantiated in line 4 by referring to the quoted dear name. The requirement that abstract features have to be defined at top-level makes the code more readable and better structured.

4.4 Mapping

Mappings between solution and problem spaces are given by constraints. Technically, cross-tree constraints can be placed in either of the spaces or in a separate file. Clafer's constraints are based on Alloy's relational logic. They concisely express statements from the first-order predicate logic. In this project, however, we used only implications, due to simplicity of the constraints. Constraints are specified in square brackets, e.g. line 3 in Fig. 13. The statement enforces that presence of exceedStepsText implies presence of ProgressSteps.Exceeded. The constraint is evaluated only if exceedStepsText exists in the model.

Real-world feature models [16] usually contain fairly simple implication constraints, such as one presented in Fig. 13 . To make our notation more informative, we included the *requires* keyword to specify implied feature inline, e.g. line

```
1  ReviewProgressMade
2    'exceedStepsText ?
3      [ProgressSteps.Exceeded]
```

Fig. 13. Expanded constraint

2 in Fig. 14. The keyword goes after cardinality constraint and is followed by the implied feature.

```
1  ReviewProgressMade
2    'exceedStepsText ? requires ProgressSteps.Exceeded
```

Fig. 14. Concise notation

Meaning of the two models in Fig. 13 and Fig. 14 is exactly the same. Figure 15 shows the underlying semantics, by expanding quotation and inserting the some keyword in the constraint. Mathematically, everything in Clafer is a relation; features are binary relations. Therefore, we require one of the quantifiers: no (zero), one, lone (zero or one), some (at least one) to precede each relation and specify the number of its elements. In this way, we can treat relational formulas as valid Boolean formulas.

```
1  ReviewProgressMade
2    exceedStepsText extends exceedStepsText ?
3      [some ProgressSteps.Exceeded]
```

Fig. 15. Desugared notation

5 Personal Health Coach

The objective of Personal Health Coach project [10] is to assist physicians in providing personalized information to patients about their health and treatments. It is believed that automatically tailored messages improve patient's satisfaction, use of physician time, and hospital resources. In contrast with generic messages, the project takes into account patient's *context* to deliver customized content.

Initially, the project used Open Document Text Format (OpenOffice.org) to represent variation space split into user model and master document. Although ODT is an open format, there are disadvantages of using it as underlying data structure:

1. No logical structure. The user model was represented as a table where each row corresponded to a feature with subfeatures. Unfortunately, it was unable to clearly express feature hierarchies and other than xor feature groups. Master document contained mappings to the user model, but all of them were written as textual comments.
2. Complex to analyze. ODT is an open format, but was not created for modeling variability. Analysis of a file requires familiarity with the internal file format. After parsing the file, the syntax tree must be filtered to remove irrelevant information.
3. Lack of tool support. There are no external tools for checking consistency of mappings between user model and master document. Furthermore, code generation for the user interface and backend was hard to automate and required help of a programmer.
4. No formal semantics. It is related to the points mentioned earlier, as lack of formal semantics leads to ambiguous documents, that are hard to analyze.
5. Project specific solution. The approach is not reusable, since there are no generic and well-defined structures.
6. Not scalable. The ODT documents were not partitioned into smaller pieces, therefore users could not work concurrently on the project.

We argue that our solution addresses most of the above issues, since Clafer has formally defined syntax and semantics, is very generic and scales by modularization and inheritance. Clafer is still a very young language, so the tool support is limited; there is a translator from Clafer to Alloy, but analysis of certain models in the Alloy Analyzer takes some time. On the other hand, thanks to uniform semantics many of the existing class and feature modeling tools could be used with Clafer, and we expect to provide this kind of support in the nearest future.

5.1 User Model

The structure of user model is presented in Fig. 16. It begins with two abstract exclusive-or groups that indicate whether a particular goal was exceeded, met or failed. We made the PositiveGoal group abstract since there are several goals in the feature model that have similar structure (e.g. UserModel.ReviewProgressOverall). They inherit the Exceeded and Met features, and add Failed with additional subfeatures. The meaning of Failed is that in case of a failure, some action (modeled as subfeatures) should be taken.

Next is the feature model of user profile. It is composed of six direct subfeatures, where user specifies his favorite type of greeting, type of relationship (friendly, encouraging, etc.), and other characteristics. Most of them are aggregated either under exclusive-or or mutually-exclusive groups. Feature model makes variability explicit by placing group cardinalities before each feature. This variability information was not available in the ODT representation of user model. Some variability knowledge existed only in people's minds.

```
1  abstract xor PositiveGoal
2    Exceeded
3    Met
4
5  abstract Goal extends PositiveGoal
6    Failed
7
8  UserModel
9    xor Greeting
10     Formal
11     SemiFormal
12     Informal
13
14   xor Relationship
15     --content
16
17   xor ReviewProgressOverall extends PositivetGoal
18     Failed
19       ProgressWeightLoss extends Goal
20       xor ProgressIntakeGoal extends PositivetGoal
21         --content
22
23       ProgressSteps extends PositiveGoal
24         Failed
25           --content
26         MeetGymExceed
27
28       ProgressDay extends PositiveGoal
29         --content
30
31       mux CurrentBarriers
32         --content
33
34       xor StrategicStepsOverall
35         --content
36
37   xor AppraiseBehaviour
38     --content
39
40   xor ReinforceResults
41     --content
42
43   mux RestageGoal
44     --content
```

Fig. 16. User model

5.2 Master Document

Master document represents generic structure of tailored messages. Figure 17 shows excerpt of the code. The document starts with a list of abstract features that define textual strings. Message content is represented either as abstract features (line 1) or inline strings (line 9). As suggested earlier, we placed short messages inline, and long messages as abstract features.

There are no group cardinalities attached to features. Although texts are mutually exclusive, the constraint is not expressed explicitly in the master document. Instead, it is inferred from the user model by attaching constraints to features in the master document. The model contains only *requires* constraints, thus they express how the final output depends on user's characteristics.

5.3 Summary

In our opinion, Clafer concisely and naturally expressed variation space of the Personalized Health Coach project. In contrast with the ODT representation, models are complete and have well-defined semantics. We modeled all the variability either as group cardinalities or *requires* constraints. Thus, the variation space is not complex by itself, but the complexity comes from potentially large feature models with hundreds or thousands possible textual messages. Testing all the configurations is practically infeasible, but there exist tools that can greatly support programmer.

6 Breast Reconstruction

This section briefly shows how we have modeled the user variation space in Clafer, for the Breast Reconstruction project, from a flowchart used by doctors to classify and determine the best response for patients. This flowchart is shown in Fig. 18.

Figure 19 shows the feature model representation for the breast reconstruction user model interpreted from the flowchart shown in Fig. 18. Decisions, such as 'Timing' or 'Lack of skin' are modeled with xor. Input and output, such as 'Lymph node dissection?' and 'Prophilactic mastectomy?' questions, are modeled as optional features. As for flow reuse, such as 'Implant variables' that is a target of 'Expander exchange timing' and 'Implant alone', we model using inheritance and abstract features.

The flow chart just shown is very similar to a sequence of questions that can be asked to a patient, in order to gather information about his situation and classify his current condition. Several projects using tailored messages have used questionnaires to model the user variability space. From this example, it is easy to see that feature models can easily be created from such questionnaires, essentially formalizing the variability of possible answers that make sense for the domain.

```
1  abstract activeAmbText = "Since you are already very active, I
2  know you are very ambitious about your exercise goals"
3
4  --content
5
6  MasterDocument
7    Introduction
8      Greeting
9        "Dear"  ? requires Formal
10       "Hello" ? requires SemiFormal
11       "Hi"    ? requires Informal
12
13       Connection
14         --content
15
16   InterventionCoaching
17     Monitoring
18       --content
19
20     Reinforcement
21       AppraiseBehaviour
22         'activeAmbText     ? requires Ambitious
23         'backOnTractText  ? requires MostlyOnTrack
24
25       ReinforceResultsCongratulate
26         --content
27
28       DiscourageNegativeThinking
29
30     Action
31       --content
32
33     GoalSetting
34       --content
35
36   ConclusionFrame
37     --content
```

Fig. 17. Master document

7 Feature Model Infrastructure

Feature models are a standard solution for variability modeling in software product-lines and feature-driven development. There exists a large variety of publications on the topic, and also ready available infrastructure for feature model editing and configuration.

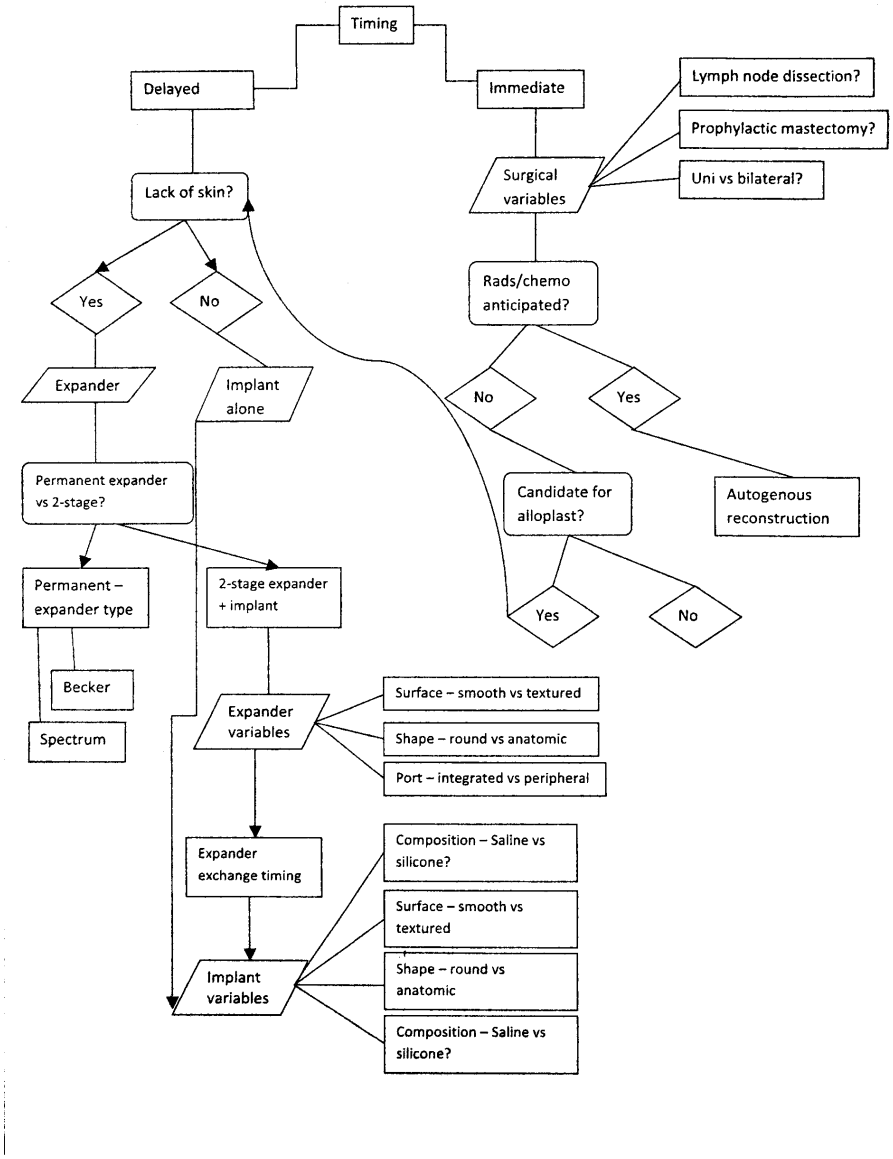


Fig. 18. Flow chart for breast reconstruction interview

Figure 20 shows features of possible tools for feature model manipulation. These tools can provide editing capabilities, such as a visual feature model editor, similar to UML model CASE tools, or simple textual editing with syntax highlighting and auto-completion. Feature model editors can also have model checking capabilities, using SAT solvers and BDDs to reason upon the propositional formula, as explained in Sect. 2.1. Model checking can assist the user in assuring that the feature model is sound (there are no dead features and that there exist at least one non-empty configuration that satisfies the model constraints), or even that the model adheres to tests specified by the user. Further editing assistance can be provided by automatically changing explicit constraints, or even providing support for batch editing, such as moving a group of features as children of a common feature.

Feature model tools can also provide simple viewing support, such as rendering feature models in text files in traditional feature diagram notation, such as used in this document. If a feature model is large, with hundreds or thousands of features [16], displaying projections or slices of the feature model might be very useful.

Feature model viewers may also provide features to assist users in configuration. Configuring large feature models with complex constraints is not an easy task without appropriate tool support. Choice propagation can help the user in resolving constraints so that a particular feature can be selected. Without choice propagation, users has to manually resolve all transitive constraints of the features they wish to select. Another valuable feature for configuration is to be able to backtrack to previous decisions when the user has changed his mind. This can often be the case in highly constrained feature models, where selecting a feature might restrict selection of other features, and users only become aware of such restrictions when they have already made the first selection.

Tools with product generation are capable of automatically generating complete products given a configuration. In the software product line perspective, this is equivalent to generating the resulting software products packaged and ready for deployment. For example, the Linux kernel has over 6000 features, and uses meta-programming via pre-compiler directives to direct code generation given a configuration of features. Product generation in tailored message perspective would be the automatic message generation given a user-configuration.

7.1 Proof of Concept Using Existing Infrastructure

In fact, the infrastructure used by the Linux kernel to model, configure and automatically generate products can easily be leveraged for message tailoring. Here we describe a proof of concept that can be built with existing tools, specifically, the Linux kernel Kconfig infrastructure.

The Linux kernel provides *qconf*, a configuration tool with simple model checking features. The input to qconf are *Kconfig* scripts, the variability modeling language created by the Linux kernel community [4]. When saving configurations using qconf, it exports the configuration as a special header C file. By

creating master documents using pre-compiler directives such as `#ifdef`, a C pre-compiler such as CPP can be used to generate tailored messages.

Figure 21 shows a slice of the Health Coach feature model in the qconf configuration tool. It presents features as choices, and only allows choices that satisfy the model's constraints. The Kconfig script that represents this model is shown in Fig. 22. We will not cover in details the syntax of Kconfig, as this information is described in the Linux kernel source tree[1].

The master document for Health Coach representing using pre-compiler directives is shown in Fig. 23. By executing qconf and saving a configuration for the user model, this should generate a `conf.h` header file defining the user features present in the configuration. Running master document file through cpp will produce the the tailored document.

Although this approach creates a functional configurator and product generator, Kconfig is not a feature model modeling language, but a script to create the qconf interface. Evermore, the master document using pre-compiler directive is not a model either, and is hard to reason upon.

Therefore, we recommend modeling both the user model and master document using Clafer, as previously explained, and using Kconfig and ifdefs as an intermediate representation, by transforming Clafer models to Kconfig and to a master document with ifdefs. This transformation is not hard to implement, and the result produces an executable configurator and product generator.

8 Related Work

Piglit (Patient Information Generated by Loosely Intelligent Techniques) [5] is a text-generation system that provides personalized hypertext explanations of patients' records. The project enabled diabetes patients to learn more about their conditions and topics mentioned in the records. Focus of our project is different, we presented a generic method for modeling variability that is not related to specific disease. Furthermore, we did not have access to patient's record. Instead, we assumed that there is a formal user model with all the relevant information.

The Migraine project [6] generates interactive materials for migraine patients who can ask follow-up questions via a mixture of hypertext and menu selection. Similarly to out project, Migraine has no access to patent's record. User's profile is built from a computer-based interview. In our project, we did not describe any method of building user's profile, but it can be done manually (by the doctor), or interactively (computer interview), or automatically (mining patient's record).

OPADE [7] generates personalized leaflets about drugs from existing sources of information. These independent sources included a drug database and the prescription. OPADE also tries to resolve potential conflicts between what the doctor wants to communicate and what the patient wants to know. The purpose of OPADE and our project is very similar. We believe that our formalism could be applied as underlying structure in OPADE.

[1] The documentation can be found in `Documentation/kbuild/kconfig-language.txt`

HealthDoc [11] is another patient information generation system that uses a master text document to create personalized information material. Text generation first selects the relevant fragments of text from the master document and then repairs the text to be made grammatically correct and coherent. Our method is based on the idea of tailoring the master document, but we only select parts of the document to create a message. The message is not repaired afterwards.

9 Conclusion

We have shown how feature modeling can be used to model variation space of tailored messages. We presented basics of feature modeling and summarized state-of-the art knowledge about feature models. Compared to previous text generation methods, our approach is formal and based on well-established ideas from Software Product Lines. These ideas were captured in Clafer, i.e. general purpose modeling language. The notation offers concise syntax for feature and class models augmented with cross-tree constraints.

Furthermore, the work gave a brief overview of existing feature model infrastructure. We used the Linux kernel tools to represent and automatically configure the master document by selecting features from the user model. It showed that feature modeling approach is not only formal, but can be easily applied to automatically tailor messages.

References

1. Antkiewicz, M., Czarnecki, K., Stephan, M.: Engineering of framework-specific modeling languages. IEEE TSE 35(6), 795–824 (2009)
2. Batory, D.: Feature models, grammars, and propositional formulas. In: SPLC'05. pp. 7–20 (2005)
3. Benavides, D., Trinidad, P., Ruiz-Cortés, A.: Automated reasoning on feature models. In: CAISE 2005. pp. 491–503 (2005)
4. Berger, T., She, S., Lotufo, R., Wąsowski, A., Czarnecki, K.: Variability modelling in the real: A perspective from the operating systems domain. In: FSE'10 (2010), under review
5. Binsted K, Cawsey A, J.R.: Generating personalised information using the medical record. In: AIME'95. pp. 29–41 (1995)
6. Carenini G, Mittal VO, M.J.: Generating patient-specific interactive natural language explanations. In: The Annual Symposium on Computer Applications in Medical Care'94 (1994)
7. de Carolis B, de Rosis F, G.F.: Generating recipient-centered explanations about drug prescription. In: AIM'96. pp. 123–145 (1996)
8. Czarnecki, K., Bednasch, T., Unger, P., Eisenecker, U.: Generative programming for embedded software: An industrial experience report. In: GPCE'02. pp. 156–172 (2002)
9. Czarnecki, K., Helsen, S., Eisenecker, U.: Staged configuration through specialization and multi-level configuration of feature models. Software Process Improvement and Practice 10(2), 143–169 (2005)

10. Dimarco, C., Hirst, G., Wanner, L., Wilkinson, J.: Healthdoc: Customizing patient information and health education by medical condition and personal characteristics. In: 1st International Workshop on Artificial Intelligence in Patient Education (1995)
11. Hirst G, DiMarco C, H.E.: Authoring and generating health-education documents that are tailored to the needs of the individual patient. In: International Conference on User Modeling'97 (1997)
12. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press (2006)
13. Jackson, M., Zave, P.: Distributed feature composition: A virtual architecture for telecommunications services. IEEE TSE 24(10), 831–847 (1998)
14. Kang, K., Cohen, S., Hess, J., Nowak, W., Peterson, S.: Feature-oriented domain analysis (FODA) feasibility study. Tech. Rep. CMU/SEI-90-TR-21, Carnegie Mellon University (1990)
15. Kramish Campbell M, DeVellis B, e.a.: Improving dietary behaviour: the efficacy tailored messages in primary care settings. AJPH 84(5), 783–787 (1994)
16. She, S., Lotufo, R., Berger, T., Wasowski, A., Czarnecki, K.: Variability model of the linux kernel. In: VaMoS'10. pp. 45–51 (2010)
17. Strecher V, Kreuter M, e.a.: The effects of computer-tailored smoking cessation letters in family practice settings. JFP 39(3), 262–270 (1994)
18. Thüm, T., Batory, D., Kästner, C.: Reasoning about edits to feature models. In: ICSE'09. pp. 254–264 (2009)

```
1
2  abstract ImplantVariables
3   Composition ?
4   Surface ?
5   Shape ?
6
7  abstract xor LackOfSkin
8   Yes
9    xor Expander
10     Permanent
11      Becker ?
12      Spectrum ?
13     2StageExpander
14      Variables
15       Surface ?
16       Shape ?
17       Port ?
18      ExchangeTiming
19       ‘ ImplantVariables
20   No
21    ImplantAlone
22     ‘ ImplantVariables
23
24 xor Timing
25  Delayed
26   ‘ LackOfSkin
27  Immediate
28   SurgicalVariables
29    LymphNodeDissection ?
30    ProphylacticMastectomy ?
31    xor Type
32     Unilateral
33     Bilateral
34   xor ChemoAnticipated
35    No
36     CandidateForAlloplast ?
37      ‘ LackOfSkin
38    Yes
39     AutogenousReconstruction
40
```

Fig. 19. Clafer model for breast reconstruction decision graph

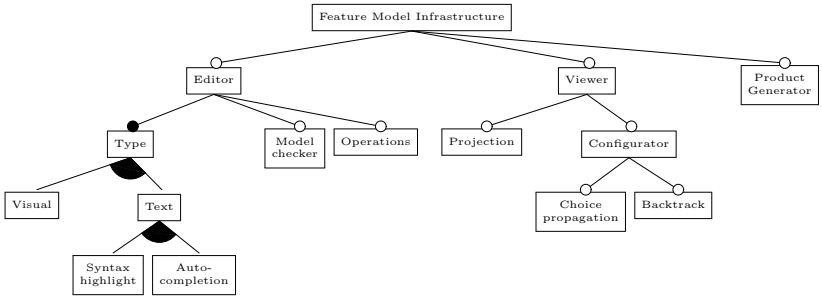


Fig. 20. Features of feature model tools

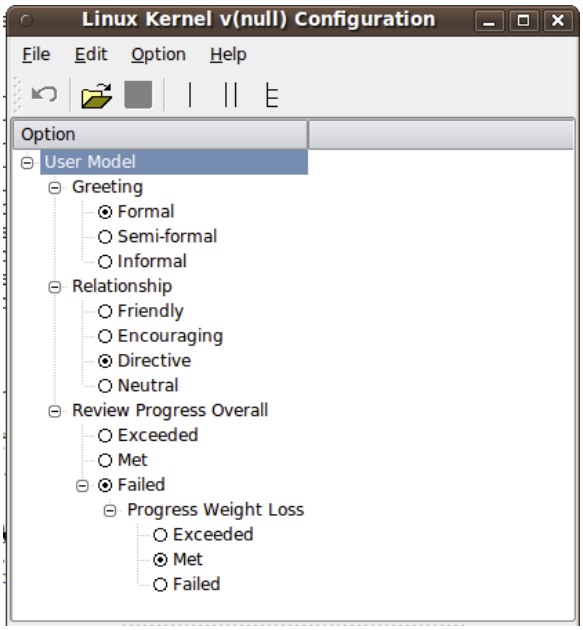


Fig. 21. Linux configuration tool for Health Coach user model

```
menu "User Model"

choice
    prompt "Greeting"

    config FORMAL
       bool "Formal"
    config SEMIFORMAL
       bool "Semi-formal"
    config INFORMAL
       bool "Informal"

endchoice # Greeting

choice
    prompt "Review Progress Overall"

    config RPO_EXCEEDED
       bool "Exceeded"
    config RPO_MET
       bool "Met"
    config RPO_FAILED
       bool "Failed"

    if RPO_FAILED
    choice
        prompt "Progress Weight Loss"

        config PWL_EXCEEDED
           bool "Exceeded"
        config PWL_MET
           bool "Met"
        config PWL_FAILED
           bool "Failed"
    endchoice # Progress Weight Loss
endif # RPO_FAILED

endchoice # Review Progress Overall

endmenu # User Model
```

Fig. 22. Kconfig representation of Personal Health Coach user model

```
1  #include "conf.h"
2
3  #ifdef CONFIG_FORMAL
4  Dear
5  #endif
6  #ifdef CONFIG_SEMIFORMAL
7  Hello
8  #endif
9  #ifdef CONFIG_INFORMAL
10 Hi
11 #endif
12
13 #ifdef CONFIG_FRIENDLY
14 How are you feeling?
15 #endif
16 You had a busy week!
17
18 I had a change to review your food and excercise logs for last week.
```

Fig. 23. Health Coach master document using pre-compiler directives