Mapping SPL Feature Models to a Relational Database

Hazim Shatnawi[*]
University of Mississippi
Department of Computer and Information Science
University, Mississippi 38677
hhshatna@go.olemiss.edu

H. Conrad Cunningham
University of Mississippi
Department of Computer and Information Science
University, Mississippi 38677
hcc@cs.olemiss.edu

ABSTRACT

Building a software product line (SPL) is a systematic strategy for reusing software within a family of related systems from some application domain. To define an SPL, domain analysts must identify the common and variable aspects of systems in the family and capture this information so that it can be used effectively to construct specific products. Often analysts record this information using a feature model expressed visually as a feature diagram. The overall goal of this project is to enable wider use of SPLs by identifying relevant concepts, defining systematic methods, and developing practical tools that leverage familiar web programming technologies. This paper presents a novel approach to specification of feature models: capture the details using automatically generated user interfaces, encode the models in a relational database, and then validate the models and construct specific products using SQL.

KEYWORDS

Software product line, feature model, relational database, domain analysis, domain design, feature selection.

ACM Reference format:
Hazim Shatnawi and H. Conrad Cunningham. 2017. Mapping SPL Feature Models to a Relational Database. In *Proceedings of ACM SouthEast Conference, Kennesaw, GA, USA, April 2017 (ACMSE'17),* 8 pages.
DOI: 10.475/123_4

1 INTRODUCTION

Four decades ago Parnas observed that "software will inevitably exist in many versions" [23]. Thus, a "software designer should be aware that he is not designing a single program but a family of programs" [24]. He describes a software family as a set "of programs whose common properties are so extensive that it is advantageous to study the common properties of the programs before analyzing individual members" [23]. That is, developers should study the commonalities of the programs before studying their variabilities. These ideas underlie contemporary research on software product lines [26].

A software product line (SPL) in some application domain is a set of related systems that share common software features but may vary in how those features are realized. To identify these commonalities and variabilities, developers must analyze the domain to identify the organization's business goals, the SPL's scope, the types of products to be developed, and the features of those products. They often document the details of the feature analysis as a tree-structured feature model. A parent node represents a decision in the design of a product. The children of that node represent more detailed decisions for realizing the parent decision. The constraints among the nodes determine how valid products can be configured in the SPL.

Feature models are represented in various ways in the literature. These include the FODA feature diagram [18] and several extensions [8, 12, 13, 16, 19, 27], special purpose languages [17, 33], and formal models [4, 31]. However, these have one or more shortcomings: some do not support automated analysis of SPLs or specific product configurations; some focus on specific programming languages; and some require the mastery of language, logic, or algebraic concepts unfamiliar to many programmers.

The overall goal of this project is to enable wider use of SPLs by identifying relevant concepts, defining systematic methods, and developing practical tools that leverage familiar web programming technologies. In Section 5, we present a novel approach to specification of feature models: encoding them in a relational database (RDB). An RDB separates the concept of a feature from its actual implementation, which helps developers and clients to understand the feature model. We leverage the capabilities of the database query language (SQL) to extract information about features and their relationships to validate the model and construct specific products. The database design also enables our tool to generate a web-based user interface for product configuration.

Before we present our approach, let us review relevant aspects of software product lines, feature models, and relational databases.

2 SOFTWARE PRODUCT LINES

According to the Software Engineering Institute (SEI), an SPL is a "set of software-intensive systems sharing a common, managed set of features that satisfy the specific need of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way" [10]. A feature is a stakeholder-visible behavior of a software system [18]. Czarnecki and Eisenecke [12] suggest making a feature visible not only helps the customer to understand the software, but it also helps domain engineers, software engineers, and others in the development of the SPL.

SPLs focus on reusing domain knowledge [22]. A domain is an abstract space in which a set of related systems share common terminology, requirements, and design choices. An organization can build reusable software assets that capture the detailed knowledge specific to a certain domain area. Consider a company that develops

[*]Also with National Center for Computational Hydroscience and Engineering.

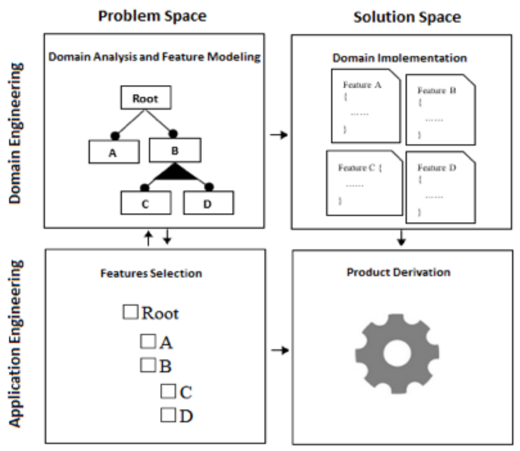


Figure 1: SPL development process

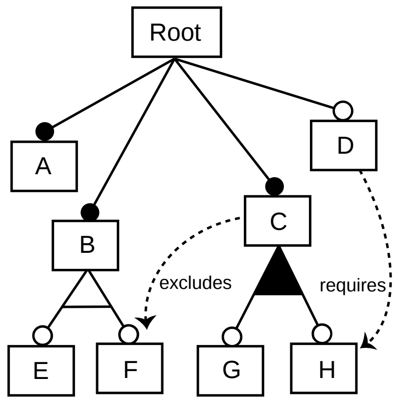


Figure 2: A feature model for a product line

software for banks. Instead of developing a unique banking system for each client, the company can use its general knowledge of banking systems—the domain—to produce a family of similar systems that share common core functionalities but that can be customized for specific clients. This family forms a software product line.

Figure 1 shows the development of an SPL consisting of two processes: ***domain engineering*** and ***application engineering*** [1]. Domain engineering is the process of analyzing a domain by studying the stakeholders' requirements, reviewing existing systems within the domain, and performing *domain analysis*. The domain analysis identifies the types of products included in the SPL and identifies the design choices that are common to the entire domain and those that vary among the specific applications. These design choices are implemented as software parts that represent the common and variable features of a product. In addition, the domain analysis examines customers' requests and decides whether to integrate these requests within the product line or just to implement them as custom features for specific requests.

Application engineering is the process of delivering a product by selecting a valid set of features from the SPL and implementing new customer requirements that are within the scope of the SPL.

Domain analysis with feature modeling represents the ***problem space*** of a product line. A problem space defines a feature as a high-level abstraction by separating the feature from its implementation details. This gives the stakeholders a clear description of the product line features. The ***solution space*** represents features as source code and converts the stakeholders' selections into concrete products.

3 FEATURE MODELS

A feature model captures all the design choices in one high-level description [3, 14, 29]. Each feature corresponds to one design choice. Some features are shared across all products in a product line (i.e., a commonality). Some only appear in specific products (i.e., a variability). The success of an SPL depends upon effective management of the variabilities. A variability means that the feature can be configured and customized when included in a delivered product [9].

A feature model documents the product line architecture resulting from the domain analysis [18]. In traditional feature models, a model is depicted as a tree that represents all design choices (i.e., features) as nodes and the constraints that one choice imposes on others by various types of edges between the nodes. Based on the defined relationships among the features in the model, the product line can generate a specific product by selecting a valid set of features—a set of features that satisfies all the constraints (i.e., rules).

A feature model captures the commonalties and variabilities by representing the primary relationships among features as a tree. The root represents the entire product line. An edge between a parent and a child is a relationship between a high-level design decision and a detailed design decision needed in its realization.

Kang proposed feature models for use in the Feature-Oriented Domain Analysis (FODA) method [18]. Other researchers have extended the feature modeling notation in various ways [12, 13, 16, 19, 27].

In feature models, features are connected by two kinds of relationships:

- Relationships between parent and child features
- Cross-tree inclusion or exclusion constraints between features

We can express a feature model visually as a feature diagram as shown in Figure 2. Root nodes are called concepts. Each concept represents a domain [2] or a complete product line [6]. The model structures the other features into a multi-level parent-child hierarchy.

There are four kinds of parent-child relationships in feature models: *mandatory*, *optional*, *alternative*, and *OR* features [4, 12, 18, 27].

Mandatory features are software artifacts that must appear in all possible configurations (generated products) of a product line. A mandatory feature can be a parent or child feature. If the feature is mandatory, then it has to be selected in the generated product whenever its parent is included. A mandatory feature is indicated by a black circle on top of the node in a feature diagram. In Figure 2, features A, B, and C are mandatory features.

Optional features are features that may or may not be selected in the generated product. The optional feature may be included if its parent is included. If its parent is optional and not included, then the feature will not be included. An optional feature is indicated by a white circle on top of the node in a feature diagram. In Figure 2, features D, E, F, G, and H are optional features.

Alternative features are group features that mean the following: if the parent of a set of alternative features is included in the generated product, then exactly one feature from this set is included. An alternative feature group is graphically represented in a feature diagram by an arc or a line that joins the alternative features' edges, forming a triangular shape. In Figure 2, features E and F are alternative features.

OR features are group features that mean the following: if a parent of a set of OR features is included in the generated product, then at least one feature from this set is included. A group of three OR features can result in selecting one, two, or three features. An OR feature group is graphically represented in a feature diagram by a black-filled arc or line that joins the OR features' edges, forming a black triangular shape. In Figure 2, features G and H are OR features.

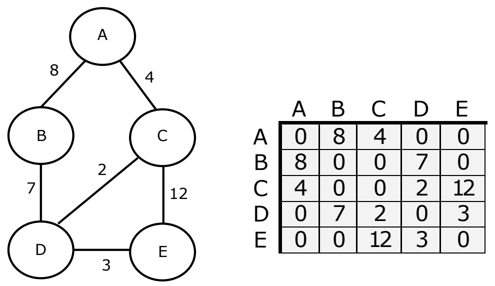
A cross-tree constraint is represented by dotted edge. In Figure 2, feature D *requires* feature H, since the edge is directed to H feature. In this case, if feature D is selected to be part of the generated product, then feature H must be selected too, but not vice versa. Feature C *excludes* feature F means that, if feature C is selected, then feature F cannot be selected, and vice versa. The *require* and *exclude* relationships are outside the hierarchical (parent-child) structure because they can relate two features that are in different branches of the tree.

The *optional*, *alternative*, and *OR* features are variation points that represent hierarchical arrangement of features. Feature models can give domain and software engineers a precise count of the possible products that can be generated from the product line based on the requirements imposed by the feature model. These require the management of variation points based on relationships and types of features. According to Figure 2:

- A is mandatory and thus it will be included in all products. The possibility of having it is always 1.
- Feature B is also mandatory and thus it will be included in all products. Feature B has two children E and F grouped in an alternative set. In this case we have the option of selecting parent B with child E or selecting parent B with child F. Therefore, feature B has two possibilities when deciding to include it in the product.
- C is mandatory and has an OR group child with two features, G and H. There are three possibilities of including feature C in a product: selecting C with G, selecting C with H, or selecting C with both G and H.
- Feature D is optional and thus it has two possibilities: either to select it in the final product or just ignore it.

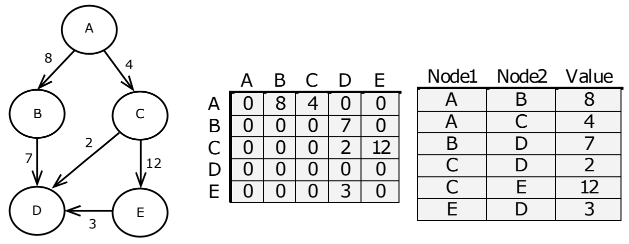
The result is achieved by multiplying the possibilities as follows:

1 possibility for A × 2 possibilities for B ×
3 possibilities for C × 2 possibilities for D
= 12 possible product configurations.



$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 8 & 4 & 0 & 0 \\ B & 8 & 0 & 0 & 7 & 0 \\ C & 4 & 0 & 0 & 2 & 12 \\ D & 0 & 7 & 2 & 0 & 3 \\ E & 0 & 0 & 12 & 3 & 0 \end{array}$$

Figure 3: Adjacency matrix for a graph



$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 8 & 4 & 0 & 0 \\ B & 0 & 0 & 0 & 7 & 0 \\ C & 0 & 0 & 0 & 2 & 12 \\ D & 0 & 0 & 0 & 0 & 0 \\ E & 0 & 0 & 0 & 3 & 0 \end{array}$$

Node1	Node2	Value
A	B	8
A	C	4
B	D	7
C	D	2
C	E	12
E	D	3

Figure 4: Labeled Digraph, adjacency matrix, RDB table

4 RELATIONAL DATABASES

A database organizes a collection of data. A relational database organizes a collection of data into tables with rows (called records) and columns (called fields or attributes). Tables link to one another using key fields. A primary key is a table column or a group of columns that uniquely identify each row. A foreign key is a table column or a group of columns that references a primary key in another table, thus creating a link between the two tables.

Tables in a relational database can be manipulated using SQL, a language for editing, querying, and updating data in a database. Database normalization reduces data redundancy and increases the performance of SQL queries.

To manipulate the feature models in a relational database, we must store both the parent-child and the cross-tree relationships. For this purpose, we consider the feature model as a graph and represent it using an adjacency matrix.

A graph is a set of nodes with edges that link pairs of nodes. If two nodes are connected by an edge, then the nodes are considered adjacent. Figure 3 shows the representation of a graph with $\boldsymbol{n}$ nodes as an $\boldsymbol{n} \times \boldsymbol{n}$ adjacency matrix. The nonzero values denote the presence of an edge between the two nodes.

We represent the parent-child and cross-tree relationships as directed edges. The matrix is sparse, so all we need to record are the pairs of adjacent nodes. Thus, we need a table with columns for the two adjacent features in the feature model and a third column to describe the relationship between the features. Figure 4 shows how we can use an adjacency matrix to represent a graph in a relational database table. Because the graph is directed, only one row is needed for each pair of features.

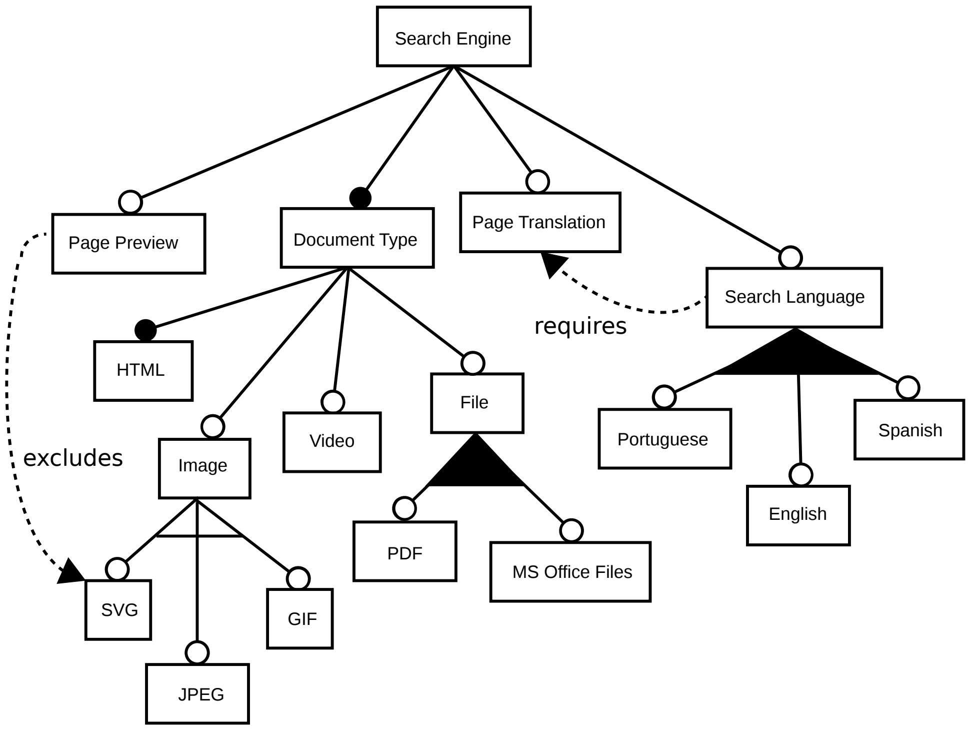


Figure 5: Feature model for a search engine SPL

5 FEATURE MODEL IN A DATABASE

We propose a novel approach that encodes feature models of SPLs in a relational database. This approach enables automated support for the creation of feature models and the generation of products from the models.

We adopt the adjacency matrix approach described in the previous section to encode feature models in a relational database system such as MySQL or PostgreSQL.

We use a simplified *Search Engine* product line [21] to illustrate our approach. Figure 5 shows a feature diagram that represents all design choices—expressing the commonalities as *mandatory* features and the variabilities as *optional*, *alternative*, and *OR* features. The diagram also shows cross-tree constraints between features represented by *require* and *exclude* relationships.

The *Search Engine* product line includes search engines that support a variety of stakeholder requirements. Each product generated from the product line will consist of common and variable features that are composed based on the features selected. The search engine system provides *Page Preview* functionality that adds an outline of the search results as thumbnails next to the webpages links. The search engine can search for and display *HTML*, *Video*, *File*, and *Image* documents. Supported formats for the *Image* document type are *SVG*, *JPEG*, and *GIF*. Supported formats for the *File* document type are *PDF* and *Microsoft Office* files (i.e., Word, Excel, PowerPoint, and Access). The search engine can *translate pages* from one language to another. A *search* can be done in any of three *languages*: *Portuguese*, *English*, and *Spanish*.

One possible product from the *Search Engine* product line is:

{ *Search Engine, Page Preview, Document Type, HTML, Image, JPEG, Video* }.

This product does not include optional features *File*, *Search Language*, and *Page Translation*. It also does not include *GIF* and *SVG* since only one feature can be selected from the alternative group. The selection of *Page Preview* feature excludes the *SVG* feature from the product. If *Page Preview* is selected, then the search engine cannot support documents of type *SVG*.

Another possible product from the *Search Engine* product line is:

{ *Search Engine, Document Type, HTML, Image, SVG, File, PDF, Search Language, English, Spanish, Page Translation* }.

The *Search Language* feature has a *requires* relationship with the *Page Translation* feature. If the stakeholder decides to have a *Search Engine* with *Search Language* functionality, then the *Page Translation* feature must be added to the generated product. *Spanish* and *English* features are selected from a three-features OR group and *PDF* is selected from a two-features OR group.

We encode feature diagrams in a relational database using the adjacency matrix approach by creating three tables: **Feature**, **featuresRelations**, and **Relationships**. We design the tables to be in third normal form (3NF) [11]. The following subsections explain the structure and use of these tables in detail.

5.1 Feature Table

The first table in our design is the **Feature** table. Each row represents a feature in a feature model. The table consists of three columns as shown in Figure 6. The first column is the **id** field,

id	name	description
SE	Search Engine	Concept node representing the product line
PP	Page Preview	Adds thumbnail preview for search results
DT	Document Type	Explains types of documents that SE support
HTM	HTML	Pages in HTML format
IMG	Image	Document type as an image
SVG	SVG	Image format for vector graphics
JPG	JPEG	Image format for pixel-graphics
GIF	GIF	Image format supports animations
VID	Video	Document type as a video
FIL	File	Document type as a file
PDF	PDF	A file in portable document formant
MSO	MS Office Files	Files with MS Office extensions
PT	Page Translation	Translates pages into a number of languages
SL	Search Language	Allows the system to support three search languages
POR	Portuguese	Portuguese language option
ENG	English	English language option
SPN	Spanish	Spanish language option

Figure 6: Feature table

which is the primary key of the feature table. It uniquely identifies each row in the table. We abbreviate the feature's **name** to get the **id** for the feature.

The second column is the **name** field of the feature. As mentioned in Section 2, a feature is a stakeholder-visible behavior of a system. Therefore, the feature's name should clearly identify its functionality for both developers and stakeholders.

The third column is the **description** field, which gives a general description of the feature.

5.2 Relationships Table

The second table in our design is the **Relationships** table. This table defines the types of relationships between features. The table consists of two columns as shown in Figure 7.

As described in Section 4, *mandatory* features represent the commonalities across all products in the product line. *OR*, *alternative*, and *optional* features are variation points—features that might or might not be selected in the generated product. *Require* and *exclude* assertions are cross-tree constraints between features. The first column of the **Relationships** table (**id**) assigns an identifier to the relationship name in the second column (**relation**).

id	relation
0	optional
1	mandatory
2	or
3	alternative
4	require
5	exclude

Figure 7: Relationships table

fromFeature	toFeature	relationType
root	SE	1
SE	PP	0
SE	DT	1
DT	HTM	1
DT	IMG	0
IMG	SVG	3
IMG	JPG	3
IMG	GIF	3
DT	VID	0
DT	FIL	0
FIL	PDF	2
FIL	MSO	2
SE	PT	0
SE	SL	0
SL	POR	2
SL	ENG	2
SL	SPN	2
PP	SVG	5
SL	PT	4

Figure 8: featuresRelations table

5.3 featuresRelations Table

The third table in our design is the **featuresRelations** table. This table defines the relationships between features. As shown in Figure 8, the table consists of three columns: **fromFeature**, **toFeature**, and **relationType**. This table represents the feature model as a directed graph as described in Section 4.

In the **featuresRelations** table, a row represents an "edge". The edge relates the feature listed in the first column (**fromFeature**) to another feature listed in the second column (**toFeature**). The type of the relationship is listed in the third column (**relationType**). The **fromFeature** and **toFeature** columns hold feature **id** keys from the **Feature** table. The **relationType** column holds relation keys from the **Relationships** table. Thus, it is not difficult to construct the feature diagram for an SPL by examining the **featuresRelations** table.

In the **featuresRelations** table, the primary key is a composite of the **fromFeature** and **toFeature** columns. In Figure 8, the

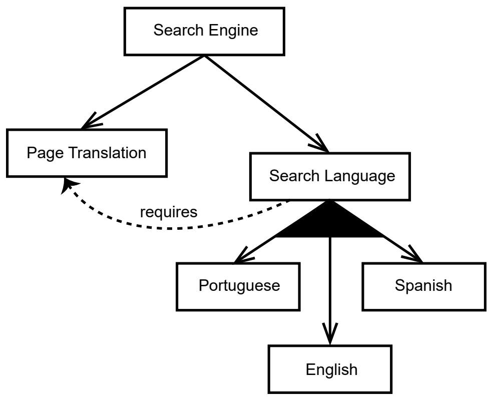


Figure 9: Feature model as DAG with labeled edges

fromFeature and **toFeature** columns together form a primary key that uniquely identifies each row in the table.

5.4 Feature Model Syntax and Semantics

Syntactically, a feature model, as represented by a feature diagram, forms a directed acyclic graph (DAG) with labelled edges. The node names are features defined in the **Feature** table. The edges are defined in the **featuresRelations** table. The possible labels on the edges are defined in the **Relationships** table. As described in previous sections, most of the edges denote relationships directed from a parent feature to a child feature in a feature tree. Other edges represent cross-tree constraints; these cannot constrain ancestor or descendent features. Our feature model specification process enforces the DAG syntax and builds valid relational database tables.

The DAG's edge labels give additional semantics of feature models encoded in the relational database. Parent-child relationships include *mandatory*, *optional*, *alternative*, and *OR* relationships described in Section 3. To simplify a model, we restrict a parent to having one group of alternative and OR children. (If more than one group is needed, we can introduce an "abstract feature" for each group and add another level to the model.) Cross-tree constraints include the *require* and *exclude* relationships as described in Section 3. The feature model specification process also encodes the intended semantics as it builds the database tables.

Figure 9 shows part of the feature model from Figure 5. It illustrates the feature model as a DAG with labeled edges.

Syntactically, our design treats the *exclude* relationship as unidirectional. However, semantically, we treat it as bidirectional. As shown in Figure 5, if feature *Page Preview* excludes feature *SVG*, then the directed edge points to *SVG* and one row in the **features-Relations** table is enough to represent this relationship. During product configuration, if a stakeholder selects *SVG* first, then the *Page Preview* feature cannot be selected later, and vice versa.

Currently, our SPL feature models have a single root. As shown in Figure 8, the **featuresRelations** table records this root with a row having the keyword *root* in the **fromFeature** column and the concept node (i.e., top-level feature) in the **toFeature** column. This enables an SQL query to identify the root easily.

Our design does not currently support feature models with more than one root or more than one parent for a child feature. But the DAG-based approach and the database encoding can be readily extended to support both.

- Multiple roots could represent a set of product lines from overlapping domains that share some features.
- Multiple parents could represent a product line with complex interactions among features at the code level.

We leave these extensions for future work.

5.5 Product Configuration

So far we have primarily considered the upper-left quadrant of Figure 1—how to represent the problem space during the domain engineering process. The result is the encoding of feature models in a relational database using the syntax and semantics described in the previous section. The feature model also provides the structure for the domain implementation (upper-right quadrant), which we plan to address in future work.

Using a feature model, how can we address the lower-left quadrant of Figure 1—selecting the product features during the application engineering process? That is, how can we build a valid software product from the specification of an SPL?

A valid product is one that conforms syntactically and semantically to the SPL's feature model. The feature model specifies the valid combinations of features. It is conveniently encoded in a relational database. Thus the product configuration process requires the system to provide the application engineer with the possible configurations and then to validate the selections made.

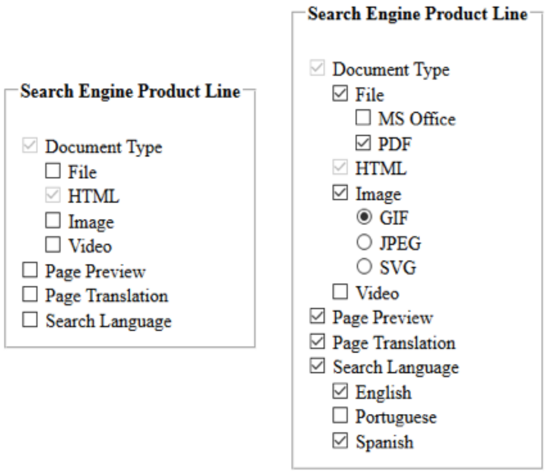
Our representation of a feature model is generic. It can represent a feature model of arbitrary depth. It is not dependent on a particular language or complex mathematics. It can be used with any programming or domain-specific language. It depends only on pervasive relational database concepts.

Our research targets development of SPLs where several programming languages can be used to build a specific product. For example, consider web applications. The client-side (front-end) languages execute in a browser on the user's machine. When the user loads a web page from a server, the browser executes the page's associated HTML, CSS, and JavaScript code locally. The server-side (back-end) languages run on the server; server-side programs may generate the client-side page and interact with it during execution. Popular server-side languages include PHP, JavaScript, ASP.NET, Perl, and Java.

As a proof of concept, we developed a general, web-based tool that recursively visits each feature in a feature model, starting from the root node. It interprets the syntax and semantics of the feature model and generates a browser-based user interface that enables the selection of any valid combination of features. The tool's current implementation uses PHP, MySQL, SQL, HTML, CSS, and JavaScript.

The left side of Figure 10 shows an automatically generated web form listing all choices available in the *Search Engine* product line given in Figure 5. This form shows *mandatory* features as preselected checkboxes because they must exist in every configured product. The form shows *optional* features as checkboxes that enable the stakeholder to include or exclude that feature.

The right side of Figure 10 shows the corresponding specification for a product that has the chosen features. It shows *File*, *Image*, and



Search Engine Product Line

- ☑ Document Type
  - ☐ File
  - ☑ HTML
  - ☐ Image
  - ☐ Video
- ☐ Page Preview
- ☐ Page Translation
- ☐ Search Language

Search Engine Product Line

- ☑ Document Type
  - ☑ File
    - ☐ MS Office
    - ☑ PDF
  - ☑ HTML
  - ☑ Image
    - ● GIF
    - ○ JPEG
    - ○ SVG
  - ☐ Video
- ☑ Page Preview
- ☑ Page Translation
- ☑ Search Language
  - ☑ English
  - ☐ Portuguese
  - ☑ Spanish

Figure 10: Product configuration

Search Language as expanded features. When a stakeholder selects a feature that has children, the system expands the feature's node to display the children indented appropriately. It uses checkboxes for *OR* features where the stakeholder can select one or more features from a group. *Alternative* features are represented by radio buttons, where the stakeholder can select only one from a group.

The cross-tree relationships *require* and *exclude* are not shown in the structure. In our example, the *Search Language* feature requires the *Page Translation* feature. If a stakeholder selects *Search Language* first, the system displays a message explaining that this feature requires the selection of *Page Translation*. If *Page Translation* is selected first, then the system does not show a warning because the relationship is in one direction. The *Page Preview* feature excludes the *SVG* feature. In this case, if the stakeholder selects either one, the system displays a message specifying this rule and disables the selection of the other. The system validates the selected choices when the stakeholder submits the configuration. The validation is based on the feature model's syntax and semantics as encoded in the database.

We plan a similar browser-based user interface to enable stakeholders to define feature models and populate the database. To enable this work and further development of the product configuration interface, we are exploring ways to formalize the syntax and semantics of the feature model and the associated user interfaces.

6 RELATED WORK

Feature models were initially introduced for use in the FODA method [18] to document the results of domain analysis. FODA produces a feature model that identifies the common and variable features of a set of related systems in the domain. The original FODA proposes the *mandatory*, *optional*, and *alternative* relationships. FORM (Feature-Oriented Reuse Method) [19] adds domain design and implementation phases to FODA. Others extend the FODA concepts and apply them in various domains [8, 13, 25, 34].

Our approach builds on the FODA concepts and methods. Our overall goal is to enable wider use of SPLs by developing a set of concepts, methods, and tools that leverage familiar web programming technologies. As much as possible, our approach seeks to be general, not tied to a specific domain, programming language, or application framework. We focus on the requirements of web application development but the tools should be applicable to other types of software development.

In the literature, there is no standard way to express feature models and integrate them into SPL development. Different representations of feature models have been proposed. Van Deursen and Klint [33] propose the Feature Description Language (FDL), a textual language to describe features that can be mapped to UML diagrams. Benavides et al [5] transform an extended feature model into a Constraint Satisfaction Problem (CSP) to analyze the model's properties. Ge [15] defines the Feature Modeling Language (FeatureML) to describe features using XML Schema. A feature model is an XML page that conforms to the language schema. Rohlik and Pasetti [28] describe other XML-related work. Thüm et al [32] introduce FeatureIDE, a framework for feature-oriented software development that supports several SPL implementation techniques. Seidl et al. [30] propose Hyper-Feature Models (HFMs) to capture variabilities in both configuration (space) and, an often neglected, aspect, evolution (time) by "explicitly providing feature versions as configurable units for product definition." Several researchers encode feature models in formal modeling languages such as Alloy to enable formal analysis and error checking [31]. Günther and Sunkle [17] embed the feature model as an object structure in Ruby, elevating features to a first-class entity in the language. Ruby's reflexive metaprogramming facilities enable the feature model to be modified and products configured at runtime. In a recent paper, Brisaboa et al [7] describe GISBuilder, a system that focuses on web-based geographic information systems (GIS) and uses specific web technologies such as AngularJS, Spring, and Hibernate.

We view the previous work through the lens of our goal of enabling wider use of SPLs by leveraging familiar web programming concepts, methods, and technologies. We thus adopted a relational database to encode feature models, SQL to manipulate the database and validate the feature models, and automatically generated web forms to gather information from stakeholders and populate the database. We will continue to adapt and extend the results of previous research on feature models and SPLs to meet our goal.

7 CONCLUSION AND FUTURE WORK

This paper proposes a novel approach for specification of an SPL by encoding its feature model in a relational database. The approach uses database tables to represent the feature model as a directed acyclic graph. Our design consists of three tables: the **Feature** table represents a feature with a unique identifier, a name, and a description; the **featuresRelations** table manages the hierarchical (*mandatory*, *optional*, *OR*, *alternative*) and cross-tree (*require* and *exclude*) relationships between features; and the **Relationships** table defines the possible relationships between features.

As a proof of concept, we developed a web application for product configuration. It recursively visits each feature in the feature model's graph and generates a browser-based user interface. The generated web form represents *mandatory* features as preselected checkboxes, *optional* and *OR* features as checkboxes, and *alternative* features as radio buttons. When a feature involved in a *require* or *exclude* relationship is selected, the web form displays a message explaining the effect of the constraint and automatically selects or deselects the other involved feature. The web application uses SQL to manipulate the database and validate the product configuration. The web application thus guides the selection of the features that conform to the syntax and semantics of the feature model.

In the domain engineering process, we plan to explore how best to formalize the syntax and semantics of feature models. This work will guide how we structure the user interface to specify SPLs and populate the database incrementally. Given our goal of using familiar web technologies, we plan to investigate use of technologies such as JavaScript, JSON (JavaScript Object Notation) Schema [20], and related approaches to specifying and validating feature models.

In the application engineering process, we plan to leverage the formalized syntax and semantics of feature models to enhance the product configuration interface. We also plan to extend the database design to include information about the implementations of features. This will support future work on the generation of products from the code and other artifacts stored in the database.

ACKNOWLEDGMENTS

We thank the anonymous referees for their helpful comments. We improved the paper in several ways as a result. We also thank our colleague David Troendle for his valuable assistance in reformatting the figures for use in the final paper. The first author thanks his current employer, the National Center for Computational Hydroscience and Engineering, for encouraging him to continue this research.

REFERENCES

[1] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines.* Springer.
[2] Ebrahim Bagheri, Faezeh Ensan, and Dragan Gasevic. 2012. Decision Support for the Software Product Line Domain Engineering Lifecycl. *Automated Software Engineering* 19, 3 (2012), 335–377.
[3] Don Batory. 2004. The Road to Utopia: A Future for Generative Programming. In *Domain-Specific Program Generation.* Springer, 1–18.
[4] Don Batory. 2005. Feature Models, Grammars, and Propositional Formulas. In *International Conference on Software Product Lines.* Springer, 7–20.
[5] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. 2005. Automated Reasoning on Feature Models. In *International Conference on Advanced Information Systems Engineering.* Springer, 491–503.
[6] Yves Bontemps, Patrick Heymans, Pierre-Yves Schobbens, and Jean-Christophe Trigaux. 2004. Semantics of FODA Feature Diagrams. In *Proceedings SPLC 2004 Workshop on Software Variability Management for Product Derivation–Towards Tool Support.* Springer, 48–58.
[7] Nieves R. Brisaboa, Alejandro Cortiñas, Miguel R. Luaces, and Oscar Pedreira. 2016. GISBuilder: A Framework for the Semi-automatic Generation of Web-based Geographic Information Systems. In *Proceedings of the 20th Pacific Asia Conference on Information Systems (PACIS 2016).*
[8] Johannes Bürdek, Sascha Lity, Malte Lochau, Markus Berens, Ursula Goltz, and Andy Schürr. 2014. Staged Configuration of Dynamic Software Product Lines with Complex Binding Time Constraints. In *Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems.* ACM, 16.
[9] Lianping Chen, Muhammad Ali Babar, and Nour Ali. 2009. Variability Management in Software Product Lines: A Systematic Review. In *Proceedings of the 13th International Software Product Line Conference.* Springer, 81–90.
[10] Paul Clements and Linda Northrop. 2002. *Software Product Lines.* Addison-Wesley.
[11] Edgar F. Codd. 1981. Data Models in Database Management. *ACM SIGMOD Record* 11, 2 (1981), 112–114.
[12] Krzysztof Czarnecki and Ulrich Eisenecker. 1999. *Generative Programming: Methods, Techniques, and Applications.* Addison-Wesley, Chapter 8.
[13] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. 2005. Formalizing Cardinality-based Feature Models and Their Specialization. *Software Process: Improvement and Practice* 10, 1 (2005), 7–29.
[14] Stefan Ferber, Jürgen Haag, and Juha Savolainen. 2002. Feature Interaction and Dependencies: Modeling Features for Reengineering a Legacy Product Line. In *International Conference on Software Product Lines.* Springer, 235–256.
[15] Guozheng Ge and E. James Whitehead. 2008. Rhizome: A Feature Modeling and Generation Platform. In *Proceedings of the International Conference on Automated Software Engineering.* IEEE, 375–378.
[16] Martin L Griss, John Favaro, and Massimo d'Alessandro. 1998. Integrating Feature Modeling with the RSEB. In *Proceedings of the Fifth International Conference on Software Reuse.* IEEE, 76–85.
[17] Sebastian Günther and Sagar Sunkle. 2012. rbFeatures: Feature-Oriented Programming with Ruby. *Science of Computer Programming* 77, 3 (2012), 152–173.
[18] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study.* Technical Report CMU/SEI-90-TR-21. Software Engineering Institute, Carnegie Mellon University.
[19] Kyo C. Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin, and Moonhang Huh. 1998. FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures. *Annals of Software Engineering* 5, 1 (1998), 143–168.
[20] Tom Marrs. 2017. *JSON at Work.* O'Reilly Media.
[21] Marcılio Mendonça. 2009. *Efficient Reasoning Techniques for Large Scale Feature Models.* Ph.D. Dissertation. University of Waterloo.
[22] Linda M. Northrop. 2002. SEI's Software Product Line Tenets. *IEEE Software* 19, 4 (2002), 32–40.
[23] David Lorge Parnas. 1976. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering* 2, 1 (1976), 1–9.
[24] David Lorge Parnas. 1979. Designing Software for Ease of Extension and Contraction. *IEEE Transactions on Software Engineering* 5, 1 (1979), 128–138.
[25] Leonardo Passos, Marko Novakovic, Yingfei Xiong, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wąsowski. 2011. A Study of Non-Boolean Constraints in Variability Models of an Embedded Operating System. In *Proceedings of the 15th International Software Product Line Conference, Volume 2.* ACM, 2.
[26] Klaus Pohl, Günter Böckle, and Frank J. van Der Linden. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer.
[27] Matthias Riebisch, Kai Böllert, Detlef Streitferdt, and Ilka Philippow. 2002. Extending Feature Diagrams with UML Multiplicities. In *Proceedings of the 6th World Conference on Integrated Design and Process Technology (IDPT).* SDPS.
[28] O. Rohlik and A. Pasetti. 2005. XFeature Modeling Tool. (2005). http://www.pnp-software.com/XFeature/
[29] Jean-Claude Royer and Hugo Arboleda. 2013. *Model-Driven and Software Product Line Engineering.* Wiley-ISTE.
[30] Christoph Seidl, Ina Schaefer, and Uwe Aßmann. 2014. Capturing Variability in Space and Time with Hyper Feature Models. In *Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems.* ACM, 6.
[31] Anjali Sree-Kumar, Elena Planas, and Robert Clarisó. 2016. Analysis of Feature Models Using Alloy: A Survey. In *Proceedings of the 7th International Workshop on Formal Methods and Analysis in Software Product Line Engineering (FMSPLE'16).* EPTCS, 45–60.
[32] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. 2014. FeatureIDE: An Extensible Framework for Feature-Oriented Software Development. *Science of Computer Programming* 79 (2014), 70–85.
[33] Arie Van Deursen and Paul Klint. 2002. Domain-Specific Language Design Requires Feature Descriptions. *CIT. Journal of Computing and Information Technology* 10, 1 (2002), 1–17.
[34] Jilles van Gurp, Jan Bosch, and Mikael Svahnberg. 2001. On the Notion of Variability in Software Product Lines. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA '01).* IEEE, 45–54.