1-1-2021

# Constructing and Validating Feature Models Using Relational, Document, and Graph Databases

hazim shatnawi
*University of Mississippi*

# CONSTRUCTING AND VALIDATING FEATURE MODELS USING RELATIONAL, DOCUMENT, AND GRAPH DATABASES

A Dissertation
presented in partial fulfillment of requirements
for the degree of Doctor of Philosophy
in the School of Engineering
The University of Mississippi

by

Hazim Shatnawi

May 2021

## 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, a domain analyst must identify the common and variable aspects of a family of systems and capture them for later use in construction of specific products. To do so, Feature-Oriented Domain Analysis (FODA) introduced the feature model as an abstraction to represent the common and variable aspects, using a feature diagram to depict the model visually. However, this abstraction is often difficult for developers to use because most tools rely on specialized theories, notations, or technologies.

This dissertation takes a novel approach. It uses mainstream database and Web technologies familiar to most developers. It represents feature models as directed acyclic graphs and encodes them using the relational (MariaDB), document-oriented (MongoDB), and graph (Neo4j) database paradigms. The design integrates these storage mechanisms with a Web interface that enables users to construct syntactically and semantically correct feature models and to configure specific products from the stored model. To enable the exchange of models among databases, the design also enables the models to be encoded as JSON text files. It provides translators from the relational database encoding to the JSON encoding and vice versa and includes algorithms to manipulate the JSON encoding directly. Finally, to determine which database encodings are the "best" from various perspectives, the dissertation evaluates them experimentally against a set of performance criteria and subjectively against a set of desirable qualities.

## DEDICATION

This dissertation is dedicated to my family (my parents and two sisters) for their endless love and support. A special thanks and appreciation to my advisor for both my M.S. and Ph.D. degrees, Professor H. Conrad Cunningham, for his countless help and support while accompanying me on this journey of exploration.

ACKNOWLEDGEMENTS

I thank my family for its endless support and love. I'm also very thankful to my advisor, Professor H. Conrad Cunningham, who supported and helped with everything he could possibly do through my M.S. and Ph.D. studies. Furthermore, I thank the committee members, who in addition to my advisor, are

- Professor Dawn E. Wilkins, Chair of the Department of Computer and Information Science
- Professor Yixin Chen, Graduate Coordinator of the Department of Computer and Information Science
- Professor Gregory L. Easson, the external member, Associate Dean for Research and Graduate Programs in the School of Engineering

for agreeing to serve on my dissertation committee and for their encouragement and support. I also thank Professors Wilkins and Chen for also agreeing to serve on my comprehensive exam committee. I extend a special thanks to Professor Wilkins, who suggested and encouraged me during the prospectus exam to pursue the research reported in Chapter 9. This addition contributed to the novelty and significance of this dissertation.

TABLE OF CONTENTS

## LIST OF FIGURES

LIST OF TABLES

# CHAPTER 1

# INTRODUCTION

In the 1970s, Parnas observed that “software will inevitably exist in many versions” [101]. Thus, a “software designer should be aware that he is not designing a single program but a family of programs” [102]. 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” [101]. That is, developers should study the commonalities of the programs before studying their variabilities. These ideas underlie contemporary research on software product lines [106].

## 1.1 Research Context

A *software product line* (SPL) is a set of software systems from some application domain in which all members share some characteristics. For any pair of systems from the set, there are also some characteristics that differentiate one from the other. The shared and differing characteristics are called *commonalities* and *variabilities*, respectively. These characteristics, or software assets, are known as *features* [12, 62].

However, as an SPL grows in size (i.e., in the number of features), it can become complex and confusing because of the many dependencies among its features. To manage this complexity, Kang et al. [62] introduced the Feature-Oriented Domain Analysis (FODA) method. In this method, the analyst studies the set of related software systems to identify its features and then assembles the features and their interrelationships into a feature model, which can be depicted using a feature diagram.

To identify an SPL's 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 recognized in the literature as one of the greatest contributions to software product line engineering [10]. A feature model is a compact representation of all possible products of a software product line. They are represented in various ways in the literature. These include the Feature-Oriented Domain Analysis (FODA) feature diagram [62] and several extensions and studies [5, 27, 35, 53, 63, 82, 124, 112], special purpose languages [54, 126], refactoring and management techniques [2], and formal models [18, 121].

Chapter 2 of this dissertation elaborates on this research context and explores other related work.

## 1.2 Research Motivation

In the previous section, we identify several ways to represent feature models. We argue that each of these has one or more of the following shortcomings:

- It does not correct flawed feature models or detect incorrect product configurations.
- It focuses on specific programming languages.
- It requires the mastery of language, logic, or algebraic concepts unfamiliar to many programmers.

As feature models grow large in size (i.e., in the number of features), they need to be represented in a way that makes the variability management reliable and convenient. This includes support for creating features, deleting features, defining relationships between features, building up a feature model, and selecting a valid set of features to form a specific product configuration.

There have been a few efforts to tackle these issues [21, 22, 36, 67, 109], but more research is needed on the process of constructing a valid feature model from scratch, building up the relations between features, and presenting the evolving feature model in an understandable and convenient manner.

In the literature, we know of little work with similar goals. The first feature model editor supporting abstract features is FeatureIDE [3]. To use FeatureIDE effectively, developers and users must familiarize themselves with the feature-oriented programming [10] and aspect-oriented programming [65] paradigms.

Van Deursen and Klint [126] propose the Feature Description Language (FDL), a textual language to describe features that can be mapped to UML diagrams. Cechticky et al. [29] and Ge and Whitehead [44] propose XML-based approaches to feature modeling and configuration. White et al. [131] propose the transformation of feature models into constraint satisfaction problems to automatically diagnose errors. However, none of these provides an automated way for creating or configuring feature models for nondevelopers. Users also need to be familiar with complex topics such as propositional formulas and constraint satisfaction or may need to learn a new programming language in order to work with feature models. The system we propose in this dissertation does not require specialized programming knowledge to create, modify, and delete features in feature models or to configure a product. In addition, our system guides the user into making correct decisions.

Günther and Sunkle [54] encode the feature model as an object in Ruby, which allows the feature model to be modified and products configured dynamically. Our approach allows feature models to be created and modified dynamically without being tied to a specific programming language.

Researchers have introduced tools such as `pure::variants` [109], staged configuration [21, 36], and FeatureIDE [3] to guide developers in configuring a feature model. However, they also require considerable software expertise to use effectively. In addition, they do not help users discover created feature models with incorrect configurations or flawed feature models [69]. Our system checks the validity of the feature model at every step of its creation, modification, and use.

Chapter 2 of this dissertation gives more details on the approaches discussed in this section.

## 1.3 Research Questions

This research seeks to answer the following general research question: ***Can mainstream Web and database technologies (relational, document-oriented, and graph) be used effectively to construct syntactically and semantically correct feature models and to configure products from these models? And, if so, which database system is the best for encoding feature models?***

To achieve this, our research aims to answer the following specific Research Questions:

1. ***Can relational database tables be used to accurately encode feature models?***
2. ***Can mainstream Web and relational database technologies be used to construct correct feature models interactively and incrementally?***
3. ***Can mainstream Web and relational database technologies be used to configure correct products corresponding to a feature model?***
4. ***Can JSON technologies be used to represent feature models correctly and enable them to be exchanged in textual form?***
5. ***Can a document-oriented NoSQL database be used to accurately encode feature models?***

6. ***Can a graph-oriented NoSQL database be used to accurately encode feature models?***

7. ***Which database system is the best for encoding feature models?***

## 1.4 Research Contributions

In this proposed research, we seek to demonstrate that the answer is "Yes" to the first six research questions and that the seventh can be answered usefully. We do so by showing how to achieve each of the following:

1. ***Encoding feature models in relational databases***

   In Chapter 3, we encode a feature model in the tables of a relational database (RDB) [116]. Use of an RDB separates the concept of a feature from its actual implementation, which helps developers and clients to understand the feature model. This chapter also specifies how to encode a feature model in an comma-separated values (CSV) text file that is equivalent to the RDB encoding.

   We published a preliminary version of this work in 2017 [116].

2. ***Designing a Web interface that automates the creation, modification, and deletion of features in a feature model.***

   Chapter 4 builds on the relational database encoding of feature models defined in Chapter 3. We design a Web interface that enables mainstream developers to create new features, define their relationships with other features, and store them in the database. The interface also enables the developer to modify or delete existing features. We base our novel design on mainstream Web technologies, using a dynamic Web form. This Web interface and the relational database encoding can form a part of a comprehensive, interactive environment.

   We use SQL to interact with the RDB tables and verify the correctness of the relevant feature's attributes upon creation, modification, or deletion. For example, we need to make sure that feature names are unique and that the relationships among features are properly structured.

We published a preliminary version of this research in 2020 [117].

3. ***Designing a Web-based approach to configuring valid products from feature models.***

   In Chapter 5, we extend the design of the Web interface from Chapter 4 with a new Web form that enables the user to configure a product from a feature model by selecting any valid combination of features. The interface interactively guides users to configure valid members of the product family represented by a feature model stored in the database. This Web form is generated dynamically by interpreting the syntax and semantics of the stored feature model.

   We published preliminary versions of this research in 2017 [116] and 2020 [117].

4. ***Defining a JSON representation for feature models that enables them to be exchanged as text and checked for correctness.***

   JavaScript Object Notation (JSON) [73] is a simple, pervasive, machine-independent, text-based language that is commonly used for transmitting and storing structured data. In the literature, feature models are described through formal methods or special purpose programming languages, which requires the mastery of a certain language, logic, or algebraic concepts unfamiliar to many programmers. Most mainstream developers are familiar with JSON, and it is supported by many libraries and tools. Because of their simplicity, JSON-based feature models can be readily exchanged among developers and easily interpreted.

   In Chapter 6, we first design an approach that can encode an arbitrary "traditional" feature model accurately in a JSON document in a manner that is equivalent to the RDB encoding defined in Chapter 3. We then design and implement programs that can translate a valid RDB encoding of a feature model to an equivalent JSON encoding and vice versa. In addition, we design operations to create, modify, and delete features in a JSON-encoded feature model.

A preliminary version of this research appears in the proceedings of the ACMSE 2021 conference [118].

5. ***Encoding feature models in document-oriented databases using MongoDB.***

   In Chapter 7, we first design an approach that can encode an arbitrary "traditional" feature model accurately in a document-oriented MongoDB database in a manner that is equivalent to the RDB and CSV encodings defined in Chapter 3. We then design and implement operations to load a feature model into a database; empty a database; create, modify, and delete features in an encoded feature model; and generate a product configuration form from the encoded model. Furthermore, we show that the approach is practical by using the implementations in the experiments in Chapter 9.

   A preliminary version of this research appears in the proceedings of the ACMSE 2021 conference [118].

6. ***Encoding feature models in graph databases using Neo4j***

   In Chapter 8, we first design an approach that can encode an arbitrary "traditional" feature model accurately in a graph-oriented Neo4j database in a manner that is equivalent to the RDB and CSV encodings defined in Chapter 3.

   We then design and implement operations to load a feature model into a database; empty a database; create, modify, and delete features in an encoded feature model; and generate a product configuration form from the encoded model. Furthermore, we show that the approach is practical by using the implementations in the experiments in Chapter 9.

7. ***Comparing relational, document, and graph database encodings of feature***

***models by performance and other factors.***

In Chapter 9, we evaluate the relational, document-oriented, and graph database encodings (defined in Chapters 3, 7, and 8) against sets of objective and subjective criteria. We select the criteria carefully to help us determine which encodings are "best" from various perspectives.

For the objective evaluation, we define, conduct, and analyze the results from a set of experiments to determine how well each database encoding performs selected operations as the feature models increase in size. For example, one operation we select is the generation of the product configuration Web form designed in Chapter 5. We want to minimize the time this operation takes for large feature models.

For the subjective evaluation, we identify several issues of interest to software developers, evaluate how well each database encoding handles each issue, and then analyze the results to determine how suitable each encoding is for the development of feature-modelings applications. For example, one issue we consider is the ease of installation. Software developers want the installation process for the software they use to be convenient and trouble-free.

Given the results of the evaluations, we can then state a few rules of thumb to guide decisions about which encoding is best under what circumstances. By considering a typical usage scenario, we can then suggest which encoding may be "best" for the feature-modeling application.

## 1.5 Dissertation Structure

The remainder of this dissertation has the following structure:

- Chapter 2 presents more detail on the context of this research and examines other related work.

- Chapter 3 defines our approach to encoding feature models in relational database tables.

- Chapter 4 presents our design for the Web user interface for constructing correct feature models and storing them in the database tables described in the previous chapter.

- Chapter 5 presents our design for a Web user interface for configuring valid products from a feature model created and encoded as described in the preceding chapters.

- Chapter 6 defines our approach to using JSON to represent feature models and translating JSON feature models to and from feature models in relational databases.

- Chapter 7 defines our approach to encoding feature models in document-oriented databases using MongoDB.

- Chapter 8 defines our approach to encoding feature models in graph databases using Neo4j.

- Chapter 9 reports on our systematic comparison of the relational, document-oriented, and graph-based databases encodings of feature models.

- Chapter 10 reviews the evaluation of the research questions and summarizes the contributions.

- Chapter 11 lists several new research questions for future investigation.

# CHAPTER 2

# BACKGROUND

As discussed in Section 1.1, our research builds on a body of previous research on software product lines (SPLs) and feature models (FMs). This chapter surveys key aspects of this work.

## 2.1 Software Product Lines

A key to successfully reusing software is capturing detailed knowledge about the software's application area, called its *domain* [23]. Reusing domain knowledge is the leading strategy for achieving effective software reuse in systems development.

Consider a software company that wants to build software management systems for school libraries. Instead of developing different applications from scratch for different libraries, and instead of randomly selecting some previously built artifacts to use, the company can apply a systematic reuse strategy that manages the process of reusing already implemented projects to develop new systems.

The company can create a software library (i.e., a suite of data and programming code) that contains the company's already implemented software parts (i.e., components, functions, algorithms, and design patterns), classify them based on their functionalities (networking domain, database, etc.), and then develop new artifacts specifically for reuse as components in future systems.

To capture and classify existing software parts, developers can analyze already implemented systems to identify which software parts are common among them and which are different. Common software parts can be incorporated into future projects in the same area while variable parts can be customized to yield different software products.

To develop reusable artifacts from scratch, the company must analyze its operations and determine what types of systems it develops, what software parts comprise those systems, and which parts would be sufficiently common across those systems to justify the costs of developing them as reusable software parts.

After capturing already existing software parts and developing new reusable software parts, the company can construct a product line of software management systems for school libraries (i.e., a family of related software systems) which are developed to provide solutions within the same problem area (i.e., managing school libraries). The company can then combine reusable software parts from this product line with variable software artifacts to yield different software applications for different clients.

The process of developing reusable software artifacts for a domain is called *domain engineering* [34]. The process of analyzing related software systems to identify and capture common and variable characteristics between them is called *domain analysis*.

The software product line approach uses both domain analysis and domain engineering processes to build software systems that share common functionalities to provide solutions for specific domain areas (i.e., banks) instead of creating different systems one by one from scratch.

*Product lining*, as a general term, is an approach to producing a group of related products that share common features. The concept of *software product* line is similar. It is a strategy based on understanding and capturing knowledge about a family of related software systems in a certain domain area to emphasize and discover common and variable parts; the common parts across the family are used to build a software platform [55], which serves as a baseline for all systems in the family while allowing variations among the family members to yield different products. Clements and Northrop define a software product line as follows [31]:

> "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."

Software core assets, (i.e., software parts), are not only restricted to source code fragments or previously implemented algorithms, but they also include reusable software components, requirements documents and specifications, design patterns, software architectures, and documentation.

### 2.1.1 History of Software Product Lines

The basic concept of a software product line is not new. Parnas called the concept a program family and described it in 1976 as follows [101]:

> "A set of programs constitutes a family whenever it is worthwhile to study programs from the set by first studying the common properties of the set and then determining the special properties of the individual members."

Although the idea was presented in the mid-1970's, the actual construction of software product lines started in the early 2000s when software companies shifted their focus tp systematic reuse strategies [96]. Software reuse practices have been adopted since the earliest days of programming, but in an ad hoc manner without following planned reuse methods. Since the 1950s, programmers have used source code fragments, subroutines, and other general software components gathered from already built systems and stored in reusable software libraries. Later, organizations started to shift their focus into more systematic reuse strategies, especially in the emerging field of software product line techniques. Figure 2.1, inspired by Northrop [96], shows the history of software reuse practices with software product line as the most recent reuse technique.

### 2.1.2 Benefits of Software Product Lines

Software product lines extend the software reuse process to cover all aspects of the software development lifecycle. Clements and Northrop [31] identify several strategic business benefits of software product lines:

- Improvement in product quality

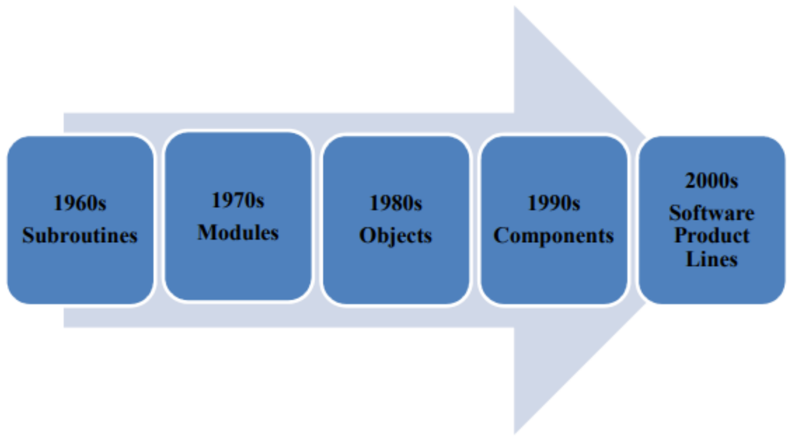


Figure 2.1. History of Software Reuse From Ad hoc to Systematic

- Reduction in development and maintenance effort
- Faster time-to-market and time-to-revenue
- Reduction in development costs and risks
- Higher customer satisfaction
- Help in finding and erasing redundant implementations

### 2.1.3 Software Product Line Principles

In the literature, many researchers (e.g., [12, 14, 34, 106]) agree that the process of software product line engineering can be divided into two subprocesses:

**Domain engineering,** which concerns building a common platform (i.e., reusable software assets)

**Application engineering,** which concerns building customer-specific applications (i.e., mass customization)

Figure 2.2 shows the three main activities of software product line development as described by Clements and Northrop [97]. It depicts the relationship between the domain and

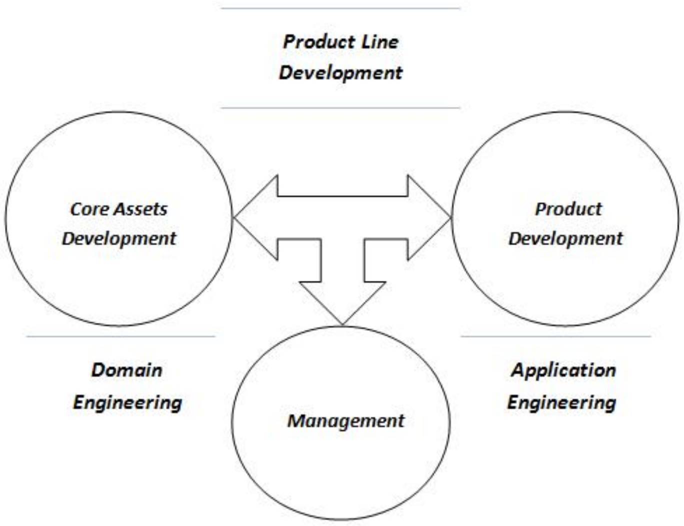


Figure 2.2. Domain and Application Engineering as Two Sub-processes of an SPL

application engineering subprocesses in terms of managing the development of both core assets and customer specific applications. Managers must define the business goals, control the risks involved in family-oriented development, analyze the potential market for products in the family, and keep the work within budget and one time.

#### 2.1.3.1 Domain Engineering

Domains are areas that group a particular set of systems or parts of systems together. Most software systems are developed to perform tasks related to some business areas (e.g., banking systems) and can be categorized based on their areas of functionality (e.g., database systems). Business areas and areas of functionality represent domains where a set of systems is grouped and dedicated for finding solutions specific to those areas.

Software systems deployed to provide solutions in a specific domain share common features and have similar development lifecycles. A software company that has built several software systems within a domain can capture its knowledge about the domain and use it to develop a family of systems.

The concept of *domain engineering* is similar; it captures the domain knowledge acquired from building several similar software systems in the form of reusable software assets and uses those assets for developing new software systems within the domain. This speeds the production process. In the literature, several definitions of the domain engineering process are proposed. All describe the process as a method that systematically reuses domain knowledge [34, 106] define the process of domain engineering as follows:

> "Domain Engineering is the activity of collecting, organizing and storing past experience in building systems or parts of systems in a particular domain in the form of reusable assets (i.e., reusable work products), as well as providing an adequate means for reusing these assets (i.e., retrieval, qualification, dissemination, adaptation, assembly, etc.) when building new systems."

As mentioned before, the software product line engineering process consists of domain engineering as the process of developing reusable core assets and application engineering as the process of developing individual products. Domain engineering consists of three main phases: domain analysis, domain design, and domain implementation [34]. The following subsections introduce each phase.

###### 2.1.3.1.1 Domain Analysis

*Domain analysis* (also called product line analysis) is the process of analyzing a set of related software systems in a domain to determine what common features they share and what variable features differentiate them from each other. The term domain analysis was coined by Neighbors [85], who explores the concept of a family of software applications developed within an area of functionalities.

Domain analysis process originated from software reuse research [127]. It was proposed as a method for identifying and gathering information from a set of software systems that share some common features; it then organizes and represents that information in a meaningful way in order to reuse it again for building new software systems [108].

There are two main purposes for performing the process of domain analysis according to Czarnecki and Eisenecker [34] and Prieto-Díaz [108]:

- Selecting and defining the domain scope
- Acquiring appropriate domain knowledge by collecting information related to the domain scope and integrating it into the domain model

Selecting and defining the domain scope requires performing business and risk analysis to select a domain that satisfies the company's objectives and goals. Information related to a specific domain is collected from different sources such as existing systems, users, domain experts and analysts, textbooks and standard documents, and experiments and prototypes. Domain analysis is performed by domain analysts. Neighbors [86] defines a domain analyst as "a person who examines the needs and requirements of a collection of systems which seems similar."

The domain analysis phase produces a domain model, a conceptual model that captures the ideas of the problem domain by representing the common and variable features of software family members in the domain. The domain modeling process produces several models and consists of several activities. These activities involve defining the domain scope and vocabulary, describing the domain concepts and their corresponding attributes, identifying relationships among features and concepts, and describing the dependencies and constraints among the variable features.

The literature varies in how it identifies the domain model's elements. However, the following are the most commonly identified elements of the domain modeling process:

**Domain scoping.** Domain scoping is the activity of determining the boundaries of a domain by finding out which systems and components belong to it. Domain scoping also gives examples of software applications already in the domain and outside the domain, gives rules of inclusion and exclusion to determine which software systems belong to the domain, and determines external systems that may interact with the domain [34].

**Domain lexicon.** A domain lexicon is a data dictionary that defines the domain vocabulary and terminology to make the communications easier between programmers and users. The domain information can be obtained from different sources such as textbooks, already existing system designs and applications, surveys and articles written by domain experts, and documented requirements and manuals [62].

**Commonality and variability analysis.** Commonality is a list of assumptions that is true for all members in the family, while variability is a list of assumptions that is true only for some members in the family. This activity is performed by domain analysts using feature modeling techniques, which are presented in the following section.

**Notations.** Notations are used to visually represent domain concepts and their attributes, relationships between concepts, and dependencies between them. This activity is performed using a wide variety of well-known models. Kang et al. [62] present several models used in domain analysis. Briefly, these are:

- Feature models, which are the fourth element of domain engineering process
- Context models, which are usually used by requirements analysts to determine if a particular application ordered by a customer is within the domain boundary for which products related to the domain are available
- Dataflow models, which usually explain how data is processed and show how dataflows between the selected domain and other domains and how they communicate
- Entity-Relationship models, which are usually used by requirements analysts to gain knowledge about a domain's entities and their inter-relationships
- Architectural models, which are usually used by domain designers for designing software applications

**Feature modeling.** The most important output of the domain analysis phase is the *feature model*. A feature model is a representation of all related software applications (i.e., the family of systems) in a domain. In the context of a software product line, the term *feature* is defined as "a characteristic of the system that is visible to the end user."

[62]. In a product line, each member is defined by a set of features that differentiate it from other members in the family of products within the domain. Features in feature models are represented at the highest level of abstraction. Users prefer to understand a particular product in the form of user-visible aspects, avoiding the complex details of internal functions that other models, such as dataflow diagrams, show. Once a particular application is defined by domain scoping, requirements analysts use feature models to discuss the application's features with users.

Since most of the work in the domain analysis phase is performed during the domain modeling process, some authors prefer to call this phase the domain modeling phase [56]. Domain modeling uses feature models, which are presented in the next section.

##### 2.1.3.1.2 Domain Design

The second phase of domain engineering is domain design. This is the activity of mapping the configurable requirements and the domain model generated from the domain analysis phase to technical solutions. This can be performed by creating a common product line architecture, which provides a common, high-level structure for the family of software systems in the target domain.

Concrete software applications can be assembled manually or automatically, where software applications can be entirely generated using techniques such as generative programming through generator tools.

##### 2.1.3.1.3 Domain Implementation

The third and final phase of domain engineering is implementing the reusable software components and their interfaces in addition to the generic architecture designed during the domain design phase. All of those implemented artifacts are used in the application engineering process for building concrete systems. Tools such as generators for automatic component assembly or domain specific languages are used to implement components. If

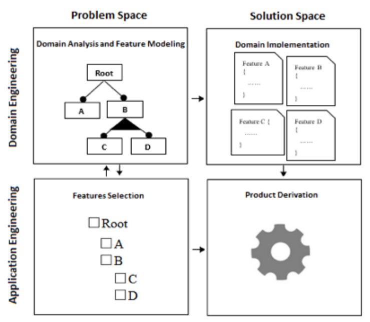


Figure 2.3. Problem and Solution Spaces

some products created for customers need additional features based on customers' requests, then custom development has to be carried out with implementation tools.

### 2.1.3.2 Application Engineering

*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 users a clear description of the product line features. The *solution space* represents features as source code and converts the users' selections into concrete products. Figure 2.3 emphasizes this and shows the development of an SPL consisting of the two processes: domain engineering and application engineering.

A problem space contains the domain concepts that application, whereas the solution space contains the implementation.

## 2.2 Feature Models

A *feature model* captures all the design choices in one high-level description [17, 41, 114]. 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 [30].

### 2.2.1 Traditional Feature Models

A feature model documents the product line architecture resulting from the domain analysis [17]. 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 the 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 commonalities 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 et al. [62] propose feature models for use in the Feature-Oriented Domain Analysis (FODA) method. Other researchers have extended the feature modeling notation in various ways [27, 28, 34, 35, 36, 103].

To develop reusable core assets for use in future systems, however, software product line engineering must manage the commonalities and variabilities across all products in a product line within a domain perspective. After determining the commonalities and variabilities, artifacts that share the most common features across products are chosen as candidates to be reused as core assets.

Other artifacts that are not part of the specified commonality will not be considered as part of the reusable core assets, but instead will be stored in the company's reusable library as variation points. Variable artifacts will be used to create different products that satisfy different customer's needs by using the application engineering process.

As a simple example, consider a company that manufactures cellular phone products. An example of a common core asset across cellular phone products would be a speaker or a microphone since it is mandatory for each phone to have a speaker or a microphone. In comparison, a Bluetooth service supporting a cellular phone is optional since many types of cellular phones do not support the Bluetooth feature. Thus, the Bluetooth artifact is considered a variable feature that would not be part of the common core assets. It would instead, be stored in the company's reusable library, which would be available to developers who wish to include it with phone products that support the Bluetooth service.

In order to determine which artifacts are candidates for the product line's reusable core assets and which are considered variable, feature models are used. Feature models support software product line engineering process by representing similarities and differences for a set of related products in a domain.

Feature diagrams, as presented in by the FODA method [62], are graphical notations that visually represent feature models in the form of tree-like diagrams (tree of features). All feature diagrams start with a node at the top of the diagram as a root node. Root nodes are called *concepts* and each concept represents a certain domain or a complete product line [14].

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 2.4. There are four kinds of parent-child relationships in feature models: *mandatory*, *optional*, *alternative*, and *OR* features [18, 34, 62].

**Mandatory** features are software artifacts that must appear in all possible configurations (generated products) of a product line. A *mandatory* feature can be 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

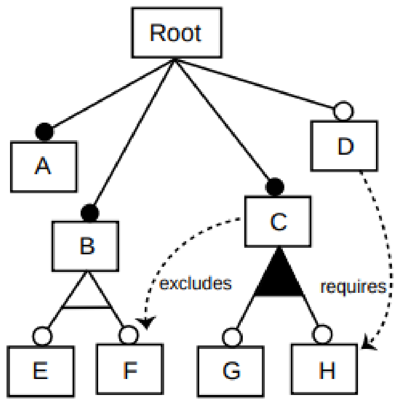


Figure 2.4. A Basic Feature Model

top of the node in a feature diagram. In Figure 2.4, 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 2.4, 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 2.4 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 2.4, features G and H are *OR* features.

A *cross-tree constraint* is represented by dotted edge. In Figure 2.4, 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 2.4:

- Feature 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.

- Feature 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.

In the approach described in Chapter 3 of this dissertation, we encode traditional feature models in relational database tables and extend the models to include feature descriptions and enhanced relationships among the features [116].

### 2.2.2 Cardinality–Based Feature Models

The *cardinality-based feature model* is an extension to the basic feature model proposed by Kang et al. [62] and Griss et al. [53]. The primary motivation is to model some cases of *OR* and *alternative* which are difficult to express in basic feature models.

Riebisch et al. [112] extended the basic feature model by replacing the *OR* and *alternative* relationships with *multiplicities* similar to those used in the Unified Modeling Language (UML). This extended feature model keeps FODA's original *mandatory* and *optional* features but generalizes the *OR* and *alternative* relationships as *set* relationships.

A *set* relationship is a set of child features where some number of features in the set is to be included in the software product when their parent feature is also included. Thus an *alternative* relationship is a *set* relationship with a cardinality `<1-1>`, and an *OR* relationship is a *set* relationship with a cardinality `<1-*>`. Consider Figure 2.5.

- Features E and F are *alternative* features. Because an *alternative* relationship denotes a 1-to-1 selection (i.e., only one selection), the relationship between feature B and the set consisting of the two features E and F is indicated by the cardinality annotation `<1-1>`.

- Features G and H are *OR* features. Because an *OR* relationship denotes a 1-to-many selection, the relationship between feature C and the set consisting of the two features G and H is indicated by the cardinality annotation `<1-2>`.

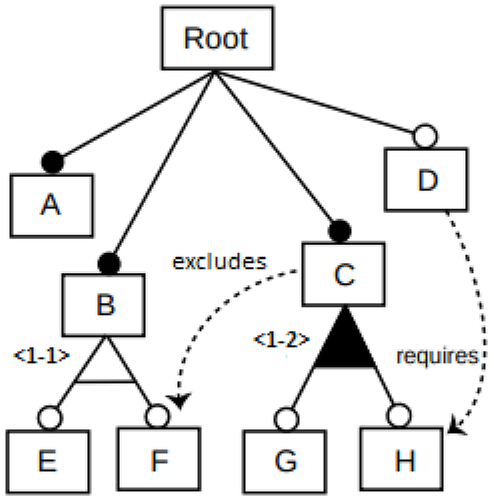


Figure 2.5. Feature Model with Multiplicities

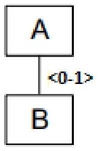


Figure 2.6. Cardinality Relationship Between Features A and B

Czarnecki et al. [35] further extend the Riebisch et al. [112] approach by also generalizing the *mandatory* and *optional* relationships using multiplicities. Their *feature cardinality* approach generalizes the *mandatory* relationship with a cardinality `<1-1>` and the *optional* relationship with a cardinality `<0-1>`. In Figure 2.6, the *optional* feature is replaced by the cardinality `<0-1>` to indicate an *optional* relationship.

In the approach proposed for this dissertation research project, we do not use multiplicities or cardinalities. Our design allows very large software product lines to be encoded in relational database tables as feature models and relationships are easily interpreted and imposed. In future research, we plan to consider adding these features as an extension to the approach taken for this dissertation.

### 2.2.3 FeatureIDE

Liech et al. [66] designed and developed FeatureIDE [3, 75] as an Eclipse plugin for feature-oriented software development. FeatureIDE supports all phases of software product line development. It uses a graphical editor for the *domain analysis* phase to manipulate feature models and their relationships.

FeatureIDE requires developers to learn special software product line techniques for the *domain implementation* phase. These include the feature-oriented programming [10] and aspect-oriented programming [65] paradigms. Our approach is not tied to any implementation language. As much as feasible, our approach will enable the use of a wide range of programming languages, general-purpose or domain-specific.

### 2.2.4 pure::variant

The `pure::variant` system [109] provides integrated tools to support all development phases of software product lines. In the domain analysis phase, its *Feature Model* editor is used to build a feature model. In the domain design phase, its *Family Model* editor is used to describe the variable family architecture and link it with the feature model through appropriate rules. The domain implementation phase generates a *Variant Result Model* from the *Family Model*, where a *Variant Description Model* is used to express the problems to be solved in terms of selected features.

This approach requires considerable software expertise to use effectively. In addition, it does not help users discover created feature models with incorrect configurations. In our approach, we use familiar technologies (relational database tables and Web interfaces) to represent and manipulate feature models.

### 2.2.5 Specific Programming Languages

Günther and Sunkle [54] 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.

This approach requires the mastery of the Ruby language and its reflexive metaprogramming facilities. Our approach allows feature models to be created and modified dynamically without being 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.

### 2.2.6 Propositional Formulas and Formal Methods

To support manipulation and analysis using automated tools, some researchers express feature models using formal specifications. Batory et al. [19] represent a feature model as a language generated by a formal grammar. A sentence in the language corresponds to a product in the SPL. Checking the validity of a product configuration is thus a matter of parsing the sentence. In other work, Batory [17] encodes a feature model in a propositional formula. A variable in the formula represents a feature. The variable has the value *true* if the feature is selected or *false* if it is not. The formula uses logical operators to encode the relationships between features, such the *OR* and *alternative* relationships. Checking the validity of a product configuration is thus a matter of evaluating the corresponding propositional formula. The configuration is valid if and only if the resulting value is *true*.

Other works express feature models using formal methods and map them to a constraint satisfaction problems. SPL configuration problems can be automatically diagnosed using a constraint problem solver [132]. Sree-Kumar et al. [121] use Alloy, a formal modeling language, to enable formal analysis and error checking. We agree that the use of grammars and propositional formulas can help users understand feature models.

Software developers should have basic familiarity with these concepts, but many may not be comfortable using them as intensely as required by some of the tools for representing feature models. Our approach provides a simpler environment to create and manipulate feature models.

# CHAPTER 3

# ENCODING FEATURE MODELS IN RELATIONAL DATABASES

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 [116].

This chapter addresses specific Research Question 1 from Section 1.3: ***Can relational database tables be used to accurately encode feature models?***

To answer this question, we show how to represent a feature model as a directed acyclic graph encoded in three relational database tables. The overall objective of this novel approach is to enable wider use of SPLs by identifying relevant concepts, defining systematic methods, and developing practical tools that leverage familiar technologies.

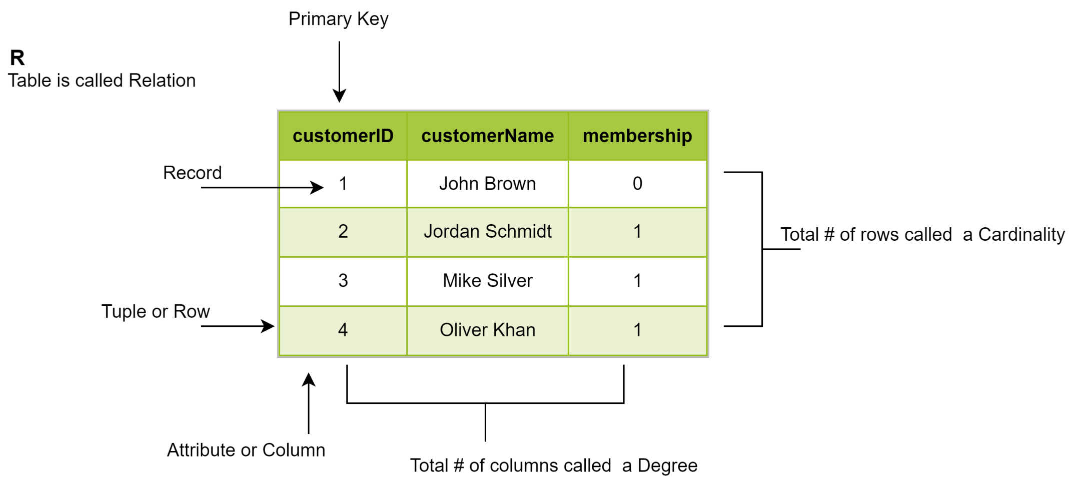
We published a preliminary version of this chapter in 2017 [116].

## 3.1 Relational Databases in a Nutshell

A database organizes a collection of related data. A relational database organizes a collection of data into tables with rows (called records) and columns (called fields or attributes). According to Gillensonm [45], a database management system (DBMS) is a system that is designed to serve two main purposes:

- Manipulating data in the database (i.e., deleting, updating, inserting)

- Providing various ways to view the data in the database



| customerID | customerName | membership |
|---|---|---|
| 1 | John Brown | 0 |
| 2 | Jordan Schmidt | 1 |
| 3 | Mike Silver | 1 |
| 4 | Oliver Khan | 1 |

Figure 3.1. A Relational Database Model

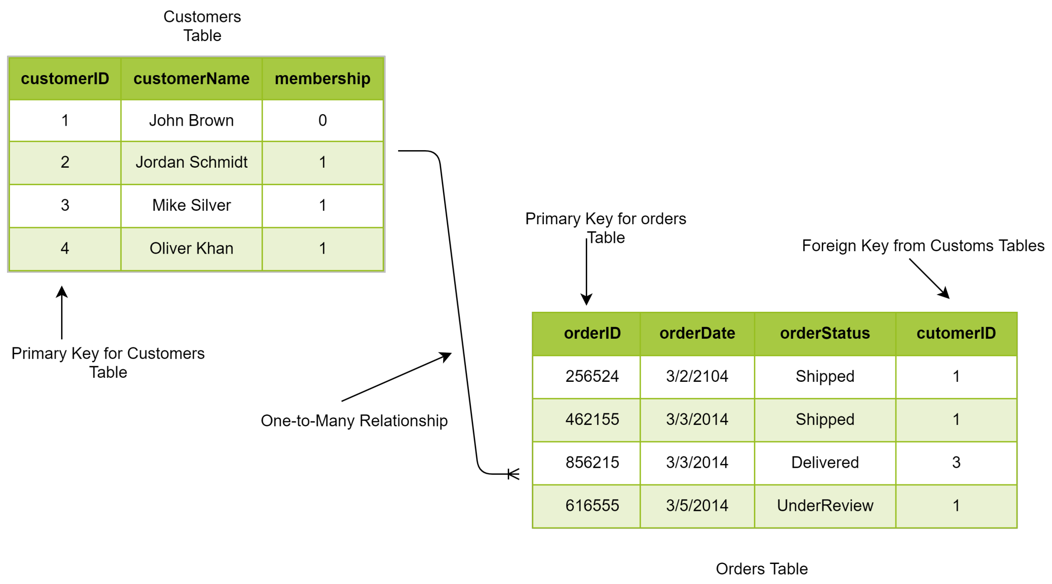
The relational database model represents a database as a collection of relations. Mathematically, a relation is a subset of the Cartesian product of two or more attributes. Each relation is represented in the model as a table.

Figure 3.1 shows the table that represents a relation named R. The table has a column for each attribute: **customerID**, **customerName**, and **membership**. The attribute names are shown in the top row of headings. The table also has a row for each tuple in the relation. This table has three columns and four rows (not counting the headings).

The total number of columns in a relation is called its *Degree*, while the total number of rows is called its *Cardinality degree*.

Tables link to one another using key fields. A *primary key* is a table column or a group of columns that uniquely identifies each row within the table. In Figure 3.1, **customerID** is the primary key for this relation because it uniquely identifies each row, that is, no two rows have the same value for **customerID**,

Figure 3.2 shows a relationship between the two tables: **Customers** and **Orders**. A *foreign key* is a table column or a group of columns that references the primary key in another table, thus creating a link between the two tables. In Figure 3.2, the **customerID** attribute, which is the primary key for the **Customers** table, appears as an attribute in the **Orders** table and is used as a reference key for the **Orders** table. The relationship between


Customers Table
Primary Key for Customers Table
One-to-Many Relationship
Primary Key for orders Table
Foreign Key from Customs Tables
Orders Table

| customerID | customerName | membership |
|---|---|---|
| 1 | John Brown | 0 |
| 2 | Jordan Schmidt | 1 |
| 3 | Mike Silver | 1 |
| 4 | Oliver Khan | 1 |

| orderID | orderDate | orderStatus | cutomerID |
|---|---|---|---|
| 256524 | 3/2/2104 | Shipped | 1 |
| 462155 | 3/3/2014 | Shipped | 1 |
| 856215 | 3/3/2014 | Delivered | 3 |
| 616555 | 3/5/2014 | UnderReview | 1 |

Figure 3.2. One-to-many Relationship Between Two Tables

the **Customers** and **Orders** tables is *one-to-many*, which means there are many orders for one customers. As shown in Figure 3.2, **cutomerID 1** has *three* orders in the **Orders** table. Foreign keys can be a group of columns that uniquely identify each row.

### 3.1.1 Structured Query Language (SQL)

Tables in a relational database can be manipulated using the *Structured Query Language* (SQL), a language for editing, querying, and updating data in a database. According to Gillensonm [45], SQL has the following main components:

- A definition language (DDL) component for creating database tables.
- A data manipulation language (DML) component for data manipulation.
- A data control language (DCL) component to provide security.

For instance, when connecting to a relational database management system such as MySQL or PostgreSQL, a user can create a *database* using the command:

```
CREATE DATABASE customersOrders;
```

The database name should be unique within the database.

Users can also create *tables* using the SQL command:

```
CREATE TABLE Customers (
    customerID int,
    customerName varchar(255),
    membership bit,
);
```

In the table created through SQL command above:

- `customerID` column is of type `int` and will hold an integer.
- `customerName` column is of type `varchar` and will hold characters, where the maximum length for this field is equal to 255 characters.
- `membership` column is of type `bit`, an integer data type that can take a value of 1, 0, or `NULL`.

SQL provides *queries*, which retrieve the data based on specific criteria. The following list shows some SQL queries to manipulate the **customersOrders** database:

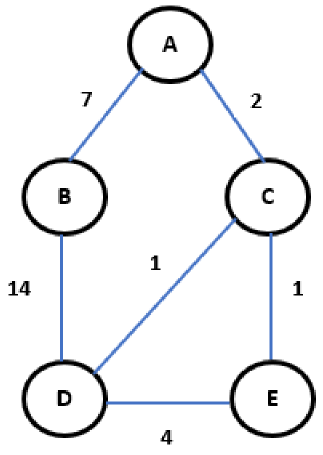
```
1 = SELECT *, Customers WHERE customerName= "John␣Brown"
2 = DELETE FROM Customers WHERE customerID = 1;
```

The first statement is a query to return all (the `*` indicates all) rows with `customerName` equal to "`John Brown`". The second statement deletes a customer data specified by the `WHERE` clause, which extracts only records that fulfill a specified condition (in this example, `customerID` = 1).

For more detail on the SQL syntax, queries, clauses, expressions, and statements, consult a reference book such as Gillensonm [45].

## 3.2 Feature Models in Database

As noted in Chapter 1, a objective of this work is to enable wider use of SPLs by identifying relevant concepts and defining systematic methods. This leads us to design



|   | A | B | C | D | E |
|---|---|---|---|---|---|
| A | 0 | 7 | 2 | 0 | 0 |
| B | 7 | 0 | 0 | 14 | 0 |
| C | 2 | 0 | 0 | 1 | 1 |
| D | 0 | 14 | 1 | 0 | 4 |
| E | 0 | 0 | 1 | 4 | 0 |

Figure 3.3. Adjacency Matrix for a Graph

a novel approach to specification of feature models: encoding the models in a relational database.

### 3.2.1 Feature Model as Adjacency Matrix

To manipulate the feature models in a relational database, we must take into account that we must preserve the structure of the feature model's hierarchical data. Thus, we must store both the *parent-child* and the *cross-tree* relationships in tables in a way that conforms the feature model's tree structure.

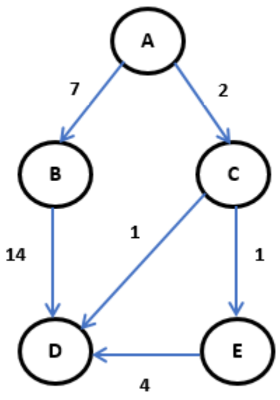
For this purpose, we consider the feature model as a graph and represent it using an *adjacency matrix.* According to Mukherjee and Mukherjee [83], an adjacency matrix of a graph ***G*** with respect to a listing of n vertices (***1, 2, ...., n***) is an **n x n** matrix, denoted by ***X(G)***, and defined as

$$X(G) = [X_{ij}]$$

where

$X_{ij}$ = 1, if ($v_i$, $v_j$) is an edge, i.e., $v_i$ is adjacent to $v_j$

$X_{ij}$ = 0, if there is no edge between v_i and v_j



$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 7 & 2 & 0 & 0 \\ B & 0 & 0 & 0 & 14 & 0 \\ C & 0 & 0 & 0 & 1 & 1 \\ D & 0 & 0 & 0 & 0 & 0 \\ E & 0 & 0 & 0 & 4 & 0 \end{array}$$

| Node 1 | Node 2 | Value |
|---|---|---|
| A | B | 7 |
| A | C | 2 |
| B | D | 14 |
| C | D | 1 |
| C | E | 1 |
| E | D | 4 |

Figure 3.4. Labeled Digraph, Adjacency Matrix, and RDB table

If an edge between vertex $v_i$ and $v_j$ exists, where **i** is a row and **j** is a column, then the value of $x_{ij}= 1$. If no edge exits, then value of $x_{ij}= 0$.

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*. A *directed graph* is a graph in which all the edges are directed from one vertex to another. Figure 3.3 shows the representation of a directed graph with **n** nodes as an **n** × **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.

As shown in Figure 3.4, 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 3.4 shows how we can use an adjacency matrix to represent a directed-graph in a relational database table. Because the graph is directed, only one row is needed for each pair of features.

We adopt the adjacency matrix to encode feature models in a relational database system such as MySQL [39] or PostgreSQL [98]. We use a simplified **Search Engine** product line adapted from Mendonça [76] to illustrate our approach. Figure 3.5 shows a feature diagram that represents all design choices—expressing the commonalities as *mandatory* features and

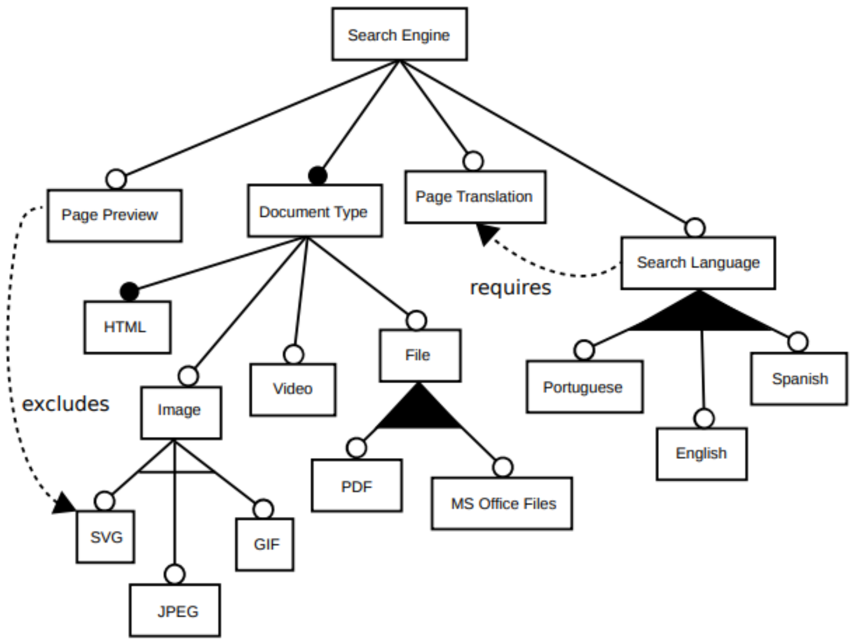


Figure 3.5. Feature Model For a Search Engine SPL

the variabilities as *optional*, *alternative*, and *OR* features. The diagram also shows cross-tree constraints between features represented by *requires* and *excludes* relationships.

The **Search Engine** product line includes search engines that support a variety of user 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 the following features:

- The **Page Preview** functionality adds an outline of the search results as thumbnails next to the Web page 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 **MS Office Files** (i.e., Microsoft Word, Excel, PowerPoint, and Access).

- The **Search Engine** can *translate* pages from one language to another (i.e., **Page Translation**).
- A search can be done in any of three **Search 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 the *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 the **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:

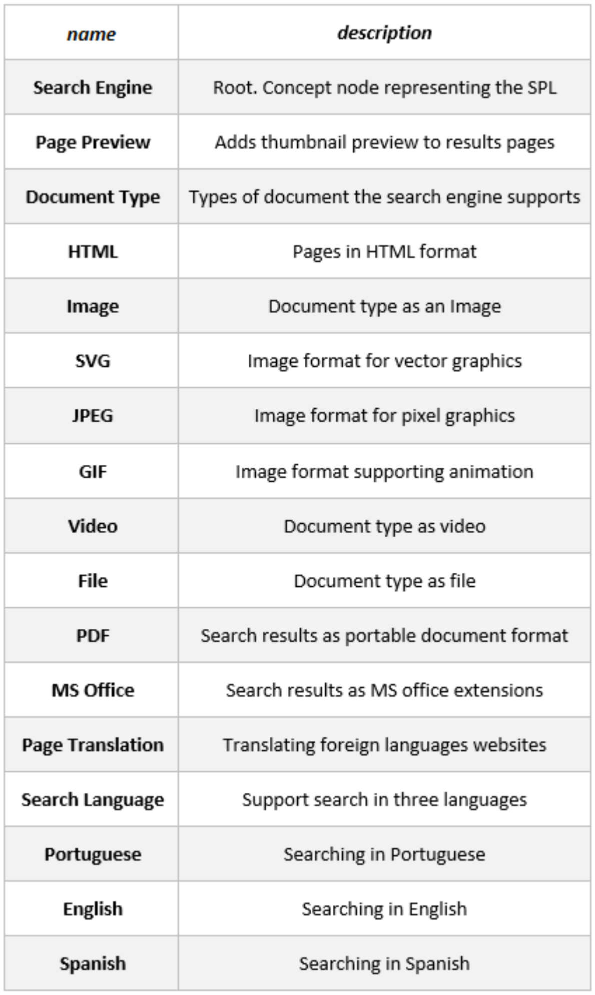
```
{   Search Engine , Document Type , HTML , Image , SVG ,
    File , PDF , Search Language , English , Spanish ,
    Page Translation }
```

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

We encode feature diagrams in a relational database using the adjacency matrix approach as described in the next subsection.

### 3.2.2 Feature Model Encoded in Relational Database Tables

We propose a novel approach that conceptualizes a feature model as a graph, represents the graph as an adjacency matrix, and encodes the matrix in three relational database



| *name* | *description* |
|---|---|
| **Search Engine** | Root. Concept node representing the SPL |
| **Page Preview** | Adds thumbnail preview to results pages |
| **Document Type** | Types of document the search engine supports |
| **HTML** | Pages in HTML format |
| **Image** | Document type as an Image |
| **SVG** | Image format for vector graphics |
| **JPEG** | Image format for pixel graphics |
| **GIF** | Image format supporting animation |
| **Video** | Document type as video |
| **File** | Document type as file |
| **PDF** | Search results as portable document format |
| **MS Office** | Search results as MS office extensions |
| **Page Translation** | Translating foreign languages websites |
| **Search Language** | Support search in three languages |
| **Portuguese** | Searching in Portuguese |
| **English** | Searching in English |
| **Spanish** | Searching in Spanish |

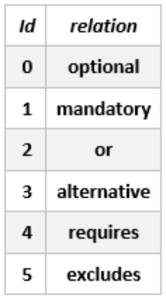
Figure 3.6. feature Table

tables in Third Normal Form [32]. Our design consists of three database tables: *feature*, *featuresRelations*, and *Relationships* tables. The following subsections introduce each one and explain its part in the design.

### 3.2.2.1 Feature Table

The first table in our design is the *feature* table. The table consists of two columns as shown in Figure 3.6.

- The first column is the **name** field, which is the primary key for the table. There is a row in the *feature* table if and only if there is feature in the feature model with the same feature **name**. As described above, a *feature* is a user-visible behavior of a



| Id | relation |
|---|---|
| 0 | optional |
| 1 | mandatory |
| 2 | or |
| 3 | alternative |
| 4 | requires |
| 5 | excludes |

Figure 3.7. Relationships Table

system. Therefore, the feature's name should clearly identify its functionality for both developers and users.

- The second column is the **description** field, which gives a general description of the feature. In Figure 3.6, we use simple descriptions for demonstration purposes.

#### 3.2.2.2 Relationships Table

The second table in our design is the *Relationships* table. This table records the possible types of relationships between features. The table consists of two columns, **id** and **relation**, as shown in Figure 3.7. The **id** is the primary key; it assigns an integer code to the relationship name given in the second column. There is exactly one row in the table for each possible relationship type in feature models.

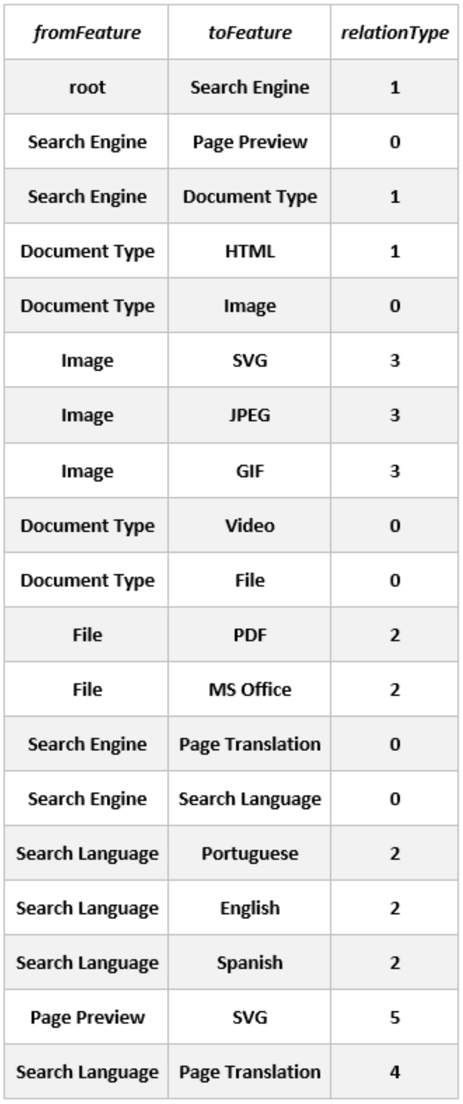
- A **mandatory** feature represents the commonalities across all products in the product line.

- The **OR**, **alternative**, and **optional** features are variation points–features that might or might not be selected in the generated product.

- The **requires** and **excludes** assertions are *cross-tree constraints* between features.

#### 3.2.2.3 featuresRelations Table

The third table in our design is the **featuresRelations** table. This table records the relationships between features in the feature model. As shown in Figure 3.8, the table



| *fromFeature* | *toFeature* | *relationType* |
|---|---|---|
| root | Search Engine | 1 |
| Search Engine | Page Preview | 0 |
| Search Engine | Document Type | 1 |
| Document Type | HTML | 1 |
| Document Type | Image | 0 |
| Image | SVG | 3 |
| Image | JPEG | 3 |
| Image | GIF | 3 |
| Document Type | Video | 0 |
| Document Type | File | 0 |
| File | PDF | 2 |
| File | MS Office | 2 |
| Search Engine | Page Translation | 0 |
| Search Engine | Search Language | 0 |
| Search Language | Portuguese | 2 |
| Search Language | English | 2 |
| Search Language | Spanish | 2 |
| Page Preview | SVG | 5 |
| Search Language | Page Translation | 4 |

Figure 3.8. featuresRelations Table

consists of three columns: **fromFeature**, **toFeature**, and **relationType**.

This table represents the feature model as the directed graph described above. In the **featuresRelations** table, a row represents an "edge" that relates the feature given in the first column **fromFeature** to the feature given the second column **toFeature**. The type of this relationship is given in the third column **relationType**. There is a row in the table if and only if there is an edge in the directed graph (i.e., feature model).

The **fromFeature** and **toFeature** columns hold feature **name** 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 **features-Relations** table.

In the **featuresRelations** table, the primary key is a composite of the **fromFeature** and **toFeature** columns. In Figure 3.8, the **fromFeature** and **toFeature** columns together form a primary key that uniquely identifies each row in the table.

#### 3.2.2.4 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 Section 3.2, 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 before. 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 *requires* and *excludes* relationships.. The feature model specification process also encodes the intended semantics as it builds the database tables.

Figure 3.9 shows part of the feature model from Figure 3.5. It illustrates the feature model as a DAG with labeled edges. Syntactically, our design treats the *excludes* relationship as unidirectional. However, semantically, we treat it as bidirectional. As shown in Figure 3.9, if feature **Page Preview** *excludes* feature **SVG**, then the directed edge points to **SVG** and one row in the **featuresRelations** table is enough to represent this relationship.

During product configuration, if a user selects **SVG** first, then the **Page Preview** feature

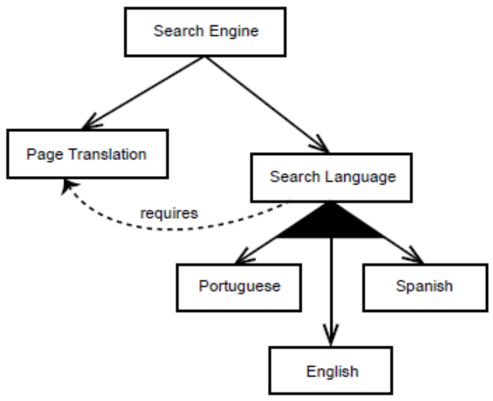


Figure 3.9. Feature Model as DAG with Labeled Edges

cannot be selected later, and vice versa. Currently, our SPL feature models have a single root. As shown in Figure 3.8, 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.

The approach we propose for this dissertation research project does not support feature models with more than one root or more than one parent for a child feature. However, 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.

In future research, we plan to consider adding these features as an extension to the approach taken for this dissertation.

## 3.3 Feature Models in CSV Files

A comma-separated values (CSV) file is a plain text file consisting of a sequence of lines. Each line is a sequence of values separated by commas. We can consider each line as representing a record and each value on the line as representing a field of the record.

Given its simple structure, a CSV file is a convenient format for transferring data—in particular tabular data—among computer applications. Most relational database systems can export a table to a CSV file with one line for each row and one value for each column (i.e, field or attribute). Similarly, they can import a CSV file with that format into a table. In particular, MySQL can export tables to and import tables from CSV files. In addition, the MongoDB and Neo4j database systems discussed in Chapters 7 and 8, respectively, have similar capabilities to read and write CSV files.

Given the support for CSV files in the three database systems we use in this dissertation, we exploit this format to define a simple encoding for feature models as CSV files. Each line of the CSV file (except the first) contains a sequence of three values recording exactly the same information that is recorded in a row of the **featuresRelations** table defined in Section 3.2:

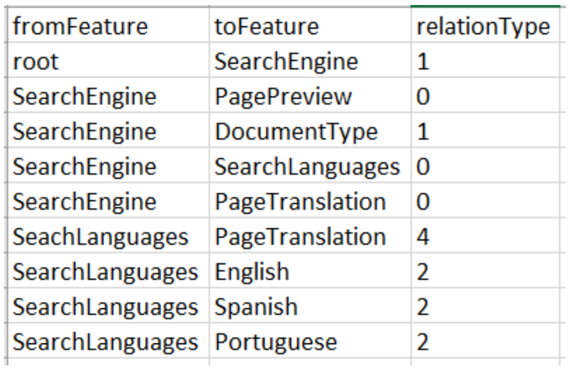
1. `fromFeature`—a valid feature name string that denotes the "parent" feature of the relationship

2. `toFeature`—a valid feature name string that denotes the "child" feature of the relationship

3. `relationType`—an integer code in the inclusive range 0 to 5 that denotes the relationship between and `fromFeature` and `toFeature`

The meanings of the `relationType` values are the same as given in the **Relationships** table. The first line of the table is a line containing the three field names `fromFeature`, `toFeature`, and `relationType`.

In addition, the collection of all lines in the CSV file (not including the first line) has the same semantic constraints that the rows of the **featuresRelations** table do. Each line is unique and it defines an edge in the feature model. The order of the lines is arbitrary. Thus, a CSV file and the corresponding **featuresRelations** table encode equivalent feature models. We sometimes refer to such a CSV file as the feature model's *CSV encoding.*



| fromFeature | toFeature | relationType |
|---|---|---|
| root | SearchEngine | 1 |
| SearchEngine | PagePreview | 0 |
| SearchEngine | DocumentType | 1 |
| SearchEngine | SearchLanguages | 0 |
| SearchEngine | PageTranslation | 0 |
| SeachLanguages | PageTranslation | 4 |
| SearchLanguages | English | 2 |
| SearchLanguages | Spanish | 2 |
| SearchLanguages | Portuguese | 2 |

Figure 3.10. CSV Structure For Search Engine Feature Model

Figure 3.10 shows a portion of a CSV encoding for the Search Engine feature model shown in Figure 3.5. The corresponding **featuresRelations** table is shown in Figure 3.8.

To import a CSV file that encodes a feature model into a corresponding table in an MySQL database, we use the following SQL query:

```
LOAD DATA INFILE '{file_name}' INTO TABLE tableName
FIELDS TERMINATED BY ',' LINES TERMINATED BY '\n'
IGNORE 1 ROWS
```

The above query reads the CSV file from the file `file_name` and loads its contents into the table `tableName`. As indicated by the clause *LINES TERMINATED BY ’*

*n’*, the query breaks the file into lines at the newline characters. The first line of the CSV file gives the three field headings; the above query assumes it is writing into a correctly structured table and skips this line by using the clause `IGNORE 1 ROWS`. Otherwise, each line of the CSV file gives a row of data to be written into the table. As indicated by `FIELD TERMINATED BY ’,’`, the fields on the line are terminated by a comma or by the end of the line. The `LOAD` statement reads the CSV file values in sequential order and writes them into the table.

A similar query to the above could be written to export a table to a CSV file. The export mechanism just needs to sequentially write each row of the table (beginning with the headings row) as a line in a text file with the field values separated by commas.

However, one operation we do use in Chapter 9 is the emptying of a database table that holds a feature model. We can do so with the following simple query:

```
TRUNCATE tableName
```

We implemented and tested these operations using Python 3.8 and XAMPP with the MariaDB database system, a fork of the MySQL relational database management system.

## 3.4 Evaluation and Conclusion

This chapter addresses Research Question 1 from Section 1.3: ***Can relational database tables be used to accurately encode feature models?***

To answer this question, we first designed a relational database to store an arbitrary "traditional" feature model. This database design consists of three tables defined as follows:

- A *feature* table with two fields, **name** and **description**, where **name** is the table's primary key. We populate this table with exactly one row for every feature existing in the feature model.
- A *Relationships* table with two fields, **id** and **relation**. This table defines a list of the identifiers for all the feature model's relationships. Ids are integer codes assigned to the various relationships defined in the feature model. These consist of the following:
  - `0` for *optional* relationships
  - `1` for *mandatory* relationships
  - `2` for *OR* relationships
  - `3` for *alternative* relationships
  - `4` for *requires* cross-tree constraints
  - `5` for *excludes* cross-tree constraints
- A *featuresRelations* table with three fields, **fromFeature**, **toFeature**, and **relationType**. For this table, the primary key is the composition of the values of the first two fields.

The **fromFeature** and **toFeature** fields are foreign keys; both must be feature names from the *feature* table (and thus features appearing in the feature model).

The **relationType** field is also a foreign key; it is an **id** value from the *Relationships* table (and thus an integer code for a relationship type in the feature model).

There is a row in the *featuresRelations* table if and only if there is an "edge" in the feature model from the feature named in the **fromFeature** field to the feature named in the **toFeature** field that has the type of relationship designated by the value of the **relationType** field.

Thus, the relational database encoding of the feature model is equivalent to the conceptual feature model.

In Section 3.3, we also designed a simple encoding of feature models as CSV files. This encoding is equivalent to the RDB encoding also defined in the chapter. It primary purpose is to provide a convenient mechanism for loading the same feature model into the three different database systems we compare in Chapter 9.

Second, we implemented this database design using the MySQL relational database management system with software written in the PHP programming language. We used the `XAMPP` [9] open-source, cross-platform Web server, which consists of the Apache HTTP Server [68], the MariaDB database [111], and interpreters for the PHP and PERL programming languages.

Third, we tested this implementation by encoding the feature model shown in Figure 3.5 in the database implementation.

In this chapter, we have thus demonstrated that the answer to Research Question 1 is "Yes". We have encoded an arbitrary "traditional" feature model accurately as a directed acyclic graph in three relational database tables [116]. Furthermore, we have shown that the design is practical by providing a proof-of-concept implementation and applying it to an example.

CHAPTER 4

# CONSTRUCTING VALID FEATURE MODELS

This chapter addresses specific Research Question 2 from Section 1.3: ***Can mainstream Web and relational database technologies be used to construct correct feature models interactively and incrementally?***

To answer this question, we show how to design a Web interface that automates the creation, modification, and deletion of features in a feature model. The design uses a dynamic Web interface enabling the creation, modification, and deletion of features and the definition of relationships and constraints among the features via Web forms.

Feature model abstractions are often difficult for mainstream developers to specify and maintain because most tools rely on specialized theories, notations, or technologies. To address this issue, this chapter presents a novel design that uses mainstream Web technologies to enable users to construct syntactically and semantically correct feature models which builds on the RDB design [116] from Chapter 3. The Web interface and relational database designs can form parts of a comprehensive, interactive environment that enables mainstream developers to specify, store, and update feature models and use them to configure members of product families.

We published a preliminary version of this chapter in 2020 [117].

## 4.1 Web Technologies in a Nutshell

*Web technologies* refer to specialized languages, protocols, and software programs used to implement applications on the World Wide Web (i.e., WWW or the Web) . The Web is based on the *client-server model* [49] as shown in Figure 4.1. The clients and servers communicate

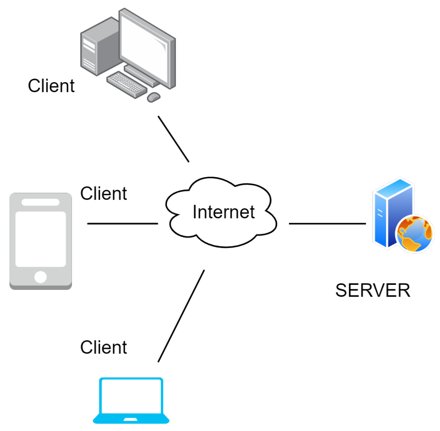


Figure 4.1. Client-Server Model

over the network using a communication protocol called the *Hypertext Transfer Protocol* (HTTP) [68]. Users interact with *client* programs (e.g., Web browsers or mobile devices) that send HTTP requests to Web *servers*. A server hosts information or computational resources on a *Web site*. It listens for requests from clients and sends an appropriate response, such as sending back *Web pages* or the results of computations.

A Web site is a collection of Web pages identified by a domain name and existing in at least one Web server. Each Web page has a name by which it can be accessed, called a *Uniform Resource Locator* (URL). A client program requests a Web page by supplying its URL to the server, with perhaps other information describing the specifics of the request. The software running on the server takes the request and sends back (or "publishes") the corresponding Web page and causes other desired effects. A Web page is a *hypermedia* document that may contain references to other Web pages. The user can subsequently request that the linked Web pages be published.

### 4.1.1 Client-Side and Server-Side Programming Languages

Websites are created using Web programming languages. These languages can be divided into two groups: *client-side* and *server-side* languages.

- A *client-side language* runs on the users' machines (typically in a browser). The browser interprets client-side languages through the browser's engine (e.g., Chrome

V8). JavaScript engines are not limited to browsers. For instance, the *Chrome V8* [50] engine is a core component of the Node.js runtime system [107]. *Client-side languages* include *JavaScript*, *WebAssembly*, *HTML*, *CSS*, and *AJAX* (for asynchronous Web applications).

- A *server-side language* runs on a Web server. It is used to program the server's responses to the clients' requests. A server-side language provides the interface to the client-side programs and controls access to an organization's private data and processing resources. On a Web server, the output from the execution of the server-side language program, forms—in whole or in part—the HTTP response to the client. The output from the server-side program may include HTML code, images, or other types of data. Unlike the client-side, there is wide range of server-side languages; prominent languages include *PHP*, *Python*, *Ruby on Rails*, *Java*, *JSP*, and *JavaScript*.

A *software framework* is an abstraction in which common code provides generic functionalities that can be selectively overridden or specialized by custom code to provide application-specific functionalities. A software framework makes it easier to create, maintain, and scale a Web application. Examples of *client-side frameworks* are *AngularJS*, *VueJS*, *ReactJS*, and *Bootstrap* (which supports both CSS and JavaScript). Examples of *server-side frameworks* are *Meteor*, *Ruby on Rails* (Ruby), *NodeJS* (Javascript), and *Django* (PHP). Other examples includes *libraries* that can be imported and used. An example a *client-side library* is *jQuery* for JavaScript. An example of a *server-side library* is *Faker*, a PHP library which allows developers to generate dummy content for Web applications.

### 4.1.2 Web Forms

A *Web form* (i.e., *HTML form*) is a client-side document with embedded *input controls*. Each control enables the user to supply data that are sent in a request to the server. Figure 4.2 shows a simple Web form for user registration.

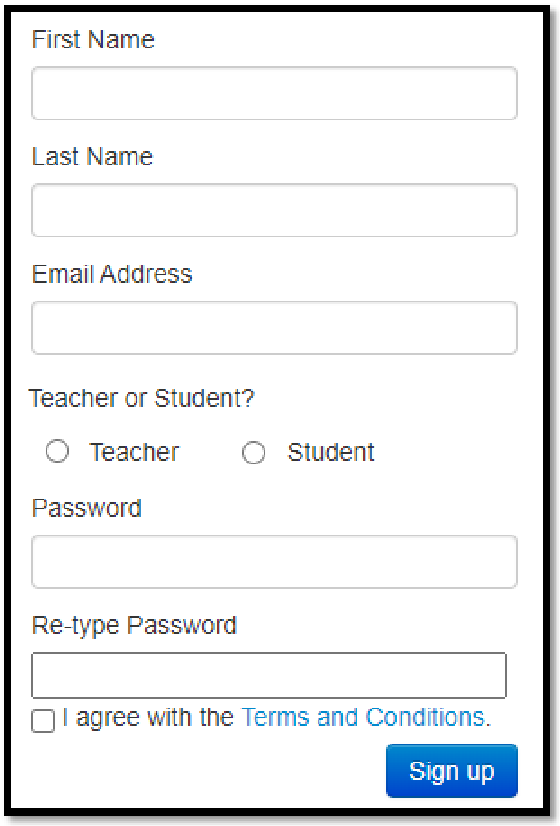


Figure 4.2. Simple Web Form For User Registration

- The controls with the labels **First Name**, **Last Name**, **Email Address**, **Password**, and **Re-type Password** are *text input* controls. A user can fill an input field with text and then the client-side program will typically validate this text to ensure that it has the appropriate format (e.g., for an email address).
- The control allowing selection of **Teacher** or **Student** is a *radio button*, which allows the user to select only one of the choices.
- The control allowing agreement with the **Terms and Conditions** is a *checkbox*, which just allows the user to select (i.e., check) or deselect (i.e., uncheck) agreement.
- After supplying correct and valid information, the user can submit the whole form by selecting the **Sign up** button. This *clickable button* control causes the client-side program to send the data entered in the form to the server-side program for further processing. In the case of this registration form, the server program will likely store the information in the user registration database after further validation.

The data entered on a Web form can be validated using some combination of client-side and server-side processing. The developer of the client-side program can attach some of the built-in HTML5 validation functions to a control (e.g., to disallow an empty field). In addition, the developer can implement custom client-side validation by writing appropriate JavaScript code. Usually, the forms sent to a server for further processing are checked again by the server-side program. This is done to ensure security and perhaps to carry out checks that are difficult to do in client-side programs.

In our approach, we provide a comprehensive example of building a user interface for creating and manipulating feature models and show how to validate users' inputs using vanilla JavaScript, jQuery, Bootstrap framework (for CSS), CSS3, and HTML as client-side programming languages, and PHP and MySQL on the server-side.

## 4.2 Feature Model Web Interface Design

When feature models grow large in size (i.e., in the number of features), they need to be represented in a way that makes the variability management reliable and convenient. This includes support for creating features, deleting features, defining relationships between features, building up a feature model, and selecting a valid set of features to form a specific product configuration. We address this need by proposing a novel design based on mainstream Web technologies. Our design uses a dynamic Web interface to enable the creation, modification, and deletion of features via Web forms.

This Web-based user interface for product creation and configuration extends the work in Chapter 3, which presents a novel approach to specification of feature models: encoding them in a relational database (RDB) [116]. The RDB design uses three tables to store the features and their hierarchical and cross-tree relationships. Using RDB tables in this way separates the concept of a feature from its implementation, which makes the feature model easy for both developers and end-users to understand.

The distinction between a feature and its implementation is useful when performing automated analyses and when reasoning about the set of different products that can be generated from the SPL using the feature model's configurations of products.

A significant benefit is the ability to use the well-known database query language (SQL) for reasoning about feature models. In this chapter, we exploit this distinction and design a dynamic user interface that collects the needed information from the users and ensures the resulting feature model is both syntactically and semantically correct. The interface also interactively guides the user to configure valid members of the product family represented by a feature model stored in the database.

In the following subsections, we present an interface design for creating and manipulating feature models. The interface supports the creation, modification, and deletion of features while constructing the SPL's feature model.

### 4.2.1 Feature Creation

Feature creation can include a new feature being inserted to a feature model, or a root feature representing the start of an SPL's concept. Figure 4.3 shows a design for the user interface's *Feature Creation* tab. It consists of a Web form with components requiring entry of the following information: the new feature's name, its parent feature's name, the type of its relationship with its parent, the other features to be *required*, and the other features to be *excluded*. The Web form's implementation must ensure that these data and their relationships with each other are valid and that only valid data are entered into the database tables.

The **Feature Name** component uses an *HTML input control* to get the name of the feature to be added. This component enables users to define features and add them to the database tables encoding the feature model. Each feature's name must have a valid format and be unique within the model. The interface ensures the uniqueness of the entered feature by performing checks on both the client side (using JavaScript, HTML, and CSS) and the server side (using PHP and MySQL). To create an SPL feature model, the user first adds the SPL's concept feature and then recursively adds children to the previously created features.

The **Feature Parent** component uses an *HTML select control* to associate a feature with its parent. This control displays a drop-down list from which the user selects the parent feature's name from among the previously defined features. This part works as a decision-choice that affects the information supported in the **Feature Type** component's form (described below). An implementation of the user interface allows each feature to have the following groups of children: *mandatory* or *optional* (which fall into their own group), *OR* (at least one feature selected), and *alternative* (exactly one feature selected). If the parent feature already has children, then the implementation must identify the types of the existing relationships between the parent and its children. It then activates or deactivates the checkboxes and radio buttons in the **Feature Type** component accordingly. If the parent feature does not have children, then all checkboxes and radio buttons in the **Feature Type**

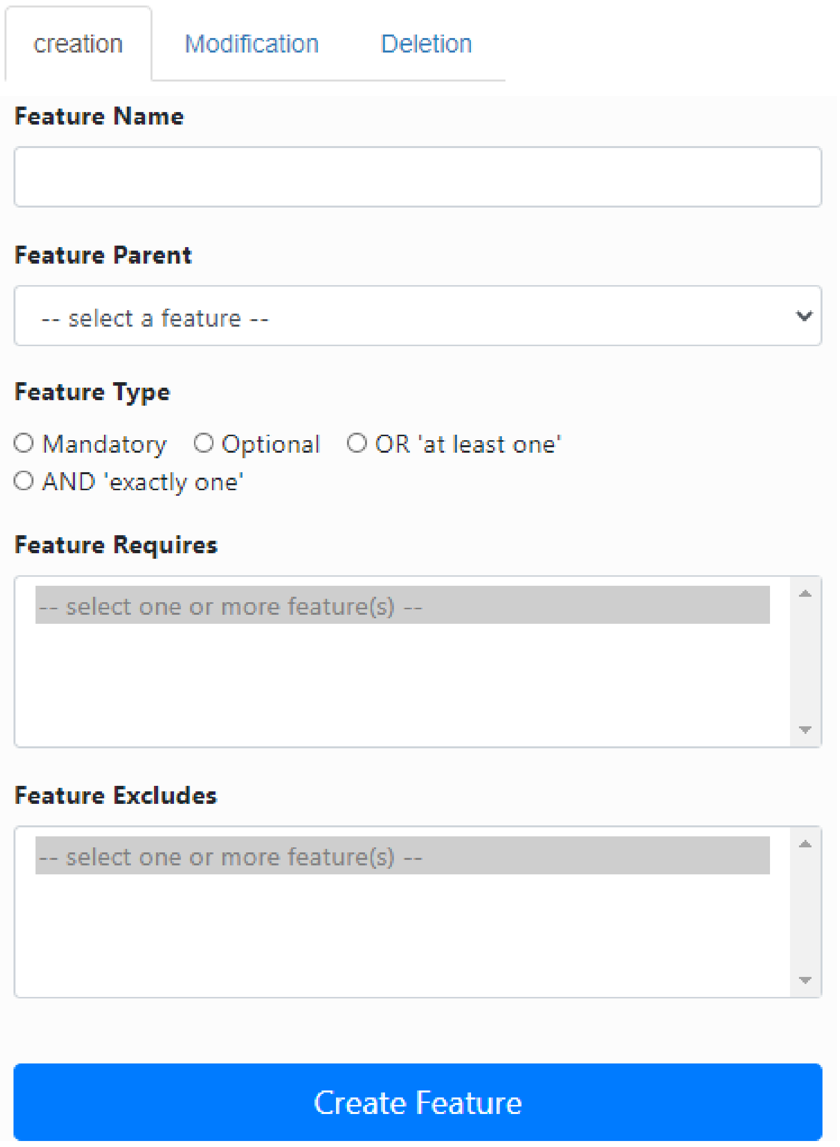


Figure 4.3. Feature Creation Tab

component are activated.

The **Feature Type** (Relationships) component uses a *composite control* to assign the relationship between a newly inserted feature and its parent. This composite control uses an *HTML list control* to group together two radio buttons and two checkboxes. The user can always choose between *mandatory* and *optional* for a feature. Once the user selects the parent feature (in the **Feature Parent** component described above), an implementation of the user interface must identify the existing relationships between the parent and its children. If the existing relationships include an *OR*, then the *OR* radio button is activated and the *alternative* radio button is deactivated because a feature cannot have two groups of *OR*/*alternative* relationships. If the relationships include an *alternative*, the *OR* radio button is similarly deactivated. If the relationship between the parent and children is only *mandatory* or *optional*, then both the *OR* and *alternative* radio buttons are deactivated while activating *mandatory* and *optional* checkboxes.

The **Feature Requires** and **Feature Excludes** component uses an *HTML multiple-selection list control* to define the *requires* and *excludes* relationships between features. The user can select `0-n` previously defined features for both the *requires* and *excludes* fields. The *requires* and *excludes* relationships must obey the following rules:

- A *mandatory* feature cannot be excluded or included. It must be independent from all other features, except its parent. Therefore, if the user chooses *mandatory* as the relationship between a newly created feature and its parent, then the *require/exclude* options are disabled. Since the new feature is *mandatory*, it cannot *require* or *exclude* other features. If the user chooses a relationship other than *mandatory*, then the list of possible required/excluded features cannot include any *mandatory* features.

- A feature cannot *require* or *exclude* an ancestor. The implementation must check this by recursively constructing the path from the new feature to the root (i.e., concept feature) of the feature model.

- A feature cannot *require* or *exclude* a sibling. An implementation must perform this check in order to remove the siblings from the lists of possible existing features to *require/exclude.*

- *Requires* and *excludes* relationships are mutually exclusive. If a user selects feature B to be *required* by feature A, then feature B cannot subsequently be chosen to be *excluded* by feature A. Similarly, if feature A *excludes* feature B, then B cannot later be *required.* This is to ensure correct choices when the user constructs cross-constraints relationships.

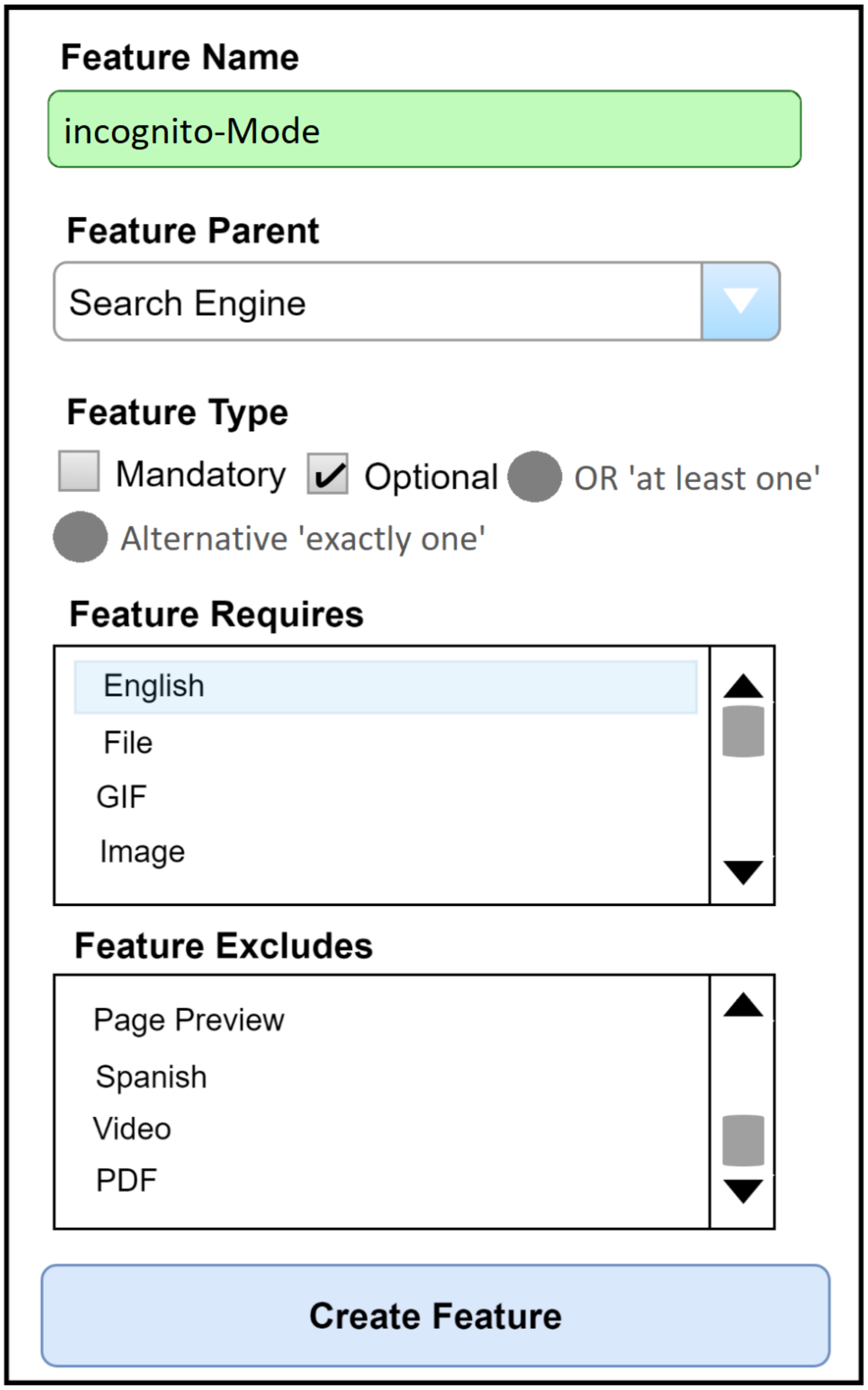
Figure 4.4 illustrates how a user can add feature *Incognito-Mode* to the feature model using the Web interface. The user interface guides the user to construct a syntactically and semantically correct (i.e., valid) feature model by going through the following steps:

1. The user enters the new feature's name *Incognito-Mode.* The user interface checks to ensure that name is not already defined.

2. The user interface lists all available parent features including the root. The user selects the new feature's parent from the list.

3. Once the user selects the feature's parent, the user interface checks that parent's relationships with its children. If the relationship is *mandatory* or *optional*, then both the *OR* and *alternative* radio buttons are deactivated while activating *mandatory* and *optional* checkboxes. If the parent feature has no children, the user can choose any of the relationships available, as all of them are activated. Note that the *mandatory* and *optional* relationships are represented by checkboxes while *or* and *alternative* are represented by radio buttons. This to allow the user to select an *OR* or *alternative* relationship while identifying whether the feature is *mandatory* or *optional.*

   In Figure 4.4, since the user selects **Search Engine** as the parent feature, its relationship with its children is either *mandatory* or *optional.* Therefore, the Web form



Feature Name

incognito-Mode

Feature Parent

Search Engine

Feature Type

☐ Mandatory ☑ Optional ● OR 'at least one' ● Alternative 'exactly one'

Feature Requires

English
File
GIF
Image

Feature Excludes

Page Preview
Spanish
Video
PDF

Figure 4.4. Creating a New Feature *incognito-Mode*

deactivates the *alternative* and *or* relationships and activates the *mandatory* and *optional.* Although they are represented as checkboxes, the user interface ensures that the user does not select both *mandatory* and *optional.* If one is selected, the other is deactivated.

4. After the user identifies the new feature's parent, the user interface lists all other features in the feature model that can be required by that feature and enables the user to select one or more for the *requires* relationship. It does not list any ancestors of the new feature. It also ensures that the selections obey the rules given above for the *requires* and *excludes* relationships.

5. Once the user identifies which features are *required* by the new feature, the user interface lists all other features in the feature model that can be *excluded* and enables the user to select one or more for the *excludes* relationship. It does not list any features that were selected to be *required* as possibilities for *excludes.* It also ensures that the selections obey the rules given above for the *requires* and *excludes* relationships. Figure 4.5 shows an algorithm to validate these cross-tree constraints.

Although the user interface checks for mutual exclusivity, it cannot detect all possible semantic errors. For example, consider a feature X that is *required* by some optional feature A and *excluded* by some other optional feature B. Suppose that both features A and B are selected to be in a particular product during the product configuration phase (described in Chapter 5). Should feature X be included or excluded? In this case, the product configuration interface notifies the user of this semantic anomaly and allows the user to modify this relation again during the product configuration.

### 4.2.2 Feature Modification and Deletion

The user interface's *Feature Modification* tab enables a previously defined feature to be changed. The user needs to enter the feature name. The form provides autocompletion

```
Requires/Excludes
Data: AJAX call to requires.php file, posting feature parent, selected at
      feature-Parent Web component form
Output: Listing both requires and excludes in requires and excludes components in
        the Web form and handle their selections
1  if parent exists in the feature model then
2      if parent is selected in feature Parent Component then
3          allFeaturesArr ← array that holds all features in the feature model except
            the root;
4          ascendantsArr ← fetchAscendants(parent); // Recursive function
            to get parent's ascendants up to the root
5          descendantsArr ←fetchDescendants(parent); // Recursive function
            to get parent's children and their descendants traversing the
            leaf nodes (features)
6          mandatoryArr ← list all mandatory arrays in the feature model;
            // Mandatory features cannot be excluded since they appear in
            every different final product
7          notToIncludeExcludeArr ← [created feature, parent, ascendantsArr and
            descendantsArr items, mandatoryArr items ];
8          requiresArr ← Filter notToIncludeExcludeArr and allFeaturesArr
            arrays and remove duplicates;
9          LIST requiresArr items in requires drop-down list
10     if feature(s) is selected from Requires list then
11         selectedRequired ← array holding features selected by user as required
            features;
12         excludesArr ← Compare requiresArr and selectedRequired arrays and
            remove all matching elements; // feature cannot be required and
            excluded at the same time
13         LIST excludesArr items in excludes drop-down list
14     SAVE user selections and update the database tables that encode the feature
        model
15 else
16     Invalid POST variable; // Check AJAX post again (front-end) or how
        POST being handled (back-end)
```

Figure 4.5. Algorithm Handling Requires and Excludes Relationships

functionality to guide the user with suggestions while typing. If the feature is found, the implementation of the user interface must fetch the information about the corresponding feature and populate the *Feature Creation* form accordingly. The user can then modify the feature's name, reattach it to a different parent feature, modify the type of relationship to its parent, and change cross-constraints relationships as needed. The same validation rules applied to these values during feature creation apply during feature modification.

When the user changes the parent to reattach the feature to a different feature, the Web interface verifies that the parent feature already exists in the feature model. The Web interface checks whether the selected parent feature has children and considers the parent's existing relationships with its children in configuring this component of the form. The existing relationships can be (a) a mix of *mandatory* and *optional* children, (b) an *OR* group, or (c) and *alternative* group. If the existing relationships are a mix of *mandatory* and *optional*, the interface deactivates both the *OR* and alternative choices for the new feature. If the existing relationships indicate an *OR* group, the interface automatically selects the *OR* choice and deactivates the alternative choice for the new feature.

The Web interface rechecks the validity of all the *requires* and *excludes* constraints in the reattached feature and in all of its descendant features. If there are invalid cross-tree constraints, the interface displays a warning message beneath the feature with the incorrect constraint. If feature A, required by feature B, gets deleted from the model, then its relationships with all other features would drop. In this case, if the user selects feature B, the user interface does not show a message to indicate that this feature is required by feature B, which no longer exists.

The user interface's *Feature Deletion* tab enables a previously defined feature to be removed from the SPL. It operates similarly to the *Feature Modification* tab by allowing the user to enter the feature name using the autocompletion functionality. An implementation of the interface must allow any feature to be deleted, even a *mandatory* feature.

If the deleted feature has no children, the user interface just deletes the feature and updates the feature model to reflect the change. If the deleted feature has children, the user interface determines what other features can be assigned as their new parent. It then prompts the user to select the new parent. If the user decides not to select a new parent, then all the children and their descendants are also deleted from the SPL. If the deleted feature is the root of the feature model, the user interface asks the user whether or not to delete the entire SPL. For children that are reassigned, the Web interface rechecks the validity of all the *requires* and *excludes* constraints in the reattached feature and in all of its descendant features. If there are invalid cross-tree constraints, the interface displays a warning message beneath the feature with the incorrect constraint. If feature A, required by feature B, gets deleted from the model, then its relationships with all other features would drop. In this case, if the user selects feature B, the user interface does not show a message to indicate that this feature is required by feature B, which no longer exists.

## 4.3 Evaluation and Conclusion

This chapter addresses Research Question 2 from Section 1.3: ***Can mainstream Web and relational database technologies be used to construct correct feature models interactively and incrementally?***

To answer this question, we first designed a dynamic Web interface that enables users to construct and modify correct feature models and store them in the RDB tables as described in Chapter 3. This includes the creation of new features, the modification or deletion of existing features, and the definition of the relationships among features. *The Web interface must disallow any addition, deletion, or modification of the feature model that would cause it to become syntactically or semantically incorrect.*

The Web interface consists of three Web forms: *Feature Creation*, *Feature Modification*, and *Feature Deletion*. Let us examine how each preserves the correctness of the feature model.

The *Feature Creation* form enables the creation of a valid new feature, including a new root of an empty feature model. This Web form asks the user for information about the new feature, validates that information, and, once complete, stores the new feature description in the RDB tables. This form requires the user to carry out the following sequence of actions:

1. Enter the name for the new feature. The Web interface validates the format of the name and then verifies that it is not already used by another feature in the feature model.

2. Select the new feature's parent. The Web interface verifies that the parent feature already exists in the feature model. If the new feature is the first to be inserted into the feature model, the new feature's parent is set to `NULL`, which makes the new feature the root of the feature model.

3. Choose the type of relationship between the new feature and its parent. The Web interface checks whether the selected parent feature has children and considers the parent's existing relationships with its children in configuring this component of the form. The existing relationships can be (a) a mix of *mandatory* and *optional* children, (b) an *OR* group, or (c) and *alternative* group. If the existing relationships are a mix of *mandatory* and *optional*, the interface deactivates both the *OR* and alternative choices for the new feature. If the existing relationships indicate an *OR* group, the interface automatically selects the *OR* choice and deactivates the alternative choice for the new feature. If the existing relationships indicate an *alternative* group, the interface automatically selects the *alternative* choice and deactivates the *OR* choice for the new feature.

4. Identify what existing feature(s) are *required* by the new feature. The Web interface disallows selection of any of the new feature's ancestors or descendants.

5. Identify what other feature(s) are *excluded* by the new feature. The Web interface disallows selection of any of the new feature's ancestors and descendants. It also

disallows selection of any feature selected above to be *required* by the new feature.

Given a correct feature model stored in the database, the Feature Creation form thus creates a valid new feature and inserts it into the feature model while preserving the correctness of the model.

The *Feature Modification* form enables an existing feature to be modified in valid ways. This Web form asks the user for information about the changes needed, validates the information, and, once complete, stores the modified feature description back into the RDB tables. This form requires the user to carry out the following sequence of actions:

1. Enter the name of a feature to be modified. The Web interface, of course, only allows an existing feature to be modified.

2. Change the selected feature's name. As in the Feature Creation tab, the Web interface validates the format of the name and then verifies that it is not already used by another feature in the feature model.

3. Reattach the selected feature to a different parent. As in the Feature Creation tab, the Web interface verifies that the parent feature already exists in the feature model. The Web interface rechecks the validity of all the *requires* and *excludes* constraints in the reattached feature and in all of its descendant features. If there are invalid cross-tree constraints, the interface displays a warning message beneath the feature with the incorrect constraint.

4. Change the type of the relationship between the selected feature and its parent. The Web interface carries out the needed validity checks, which are similar to those described above for the "Choose the type of relationship" component of the Feature Creation tab.

5. Choose different features for for selected feature's *requires* relationships. The Web interface carries out the needed validity checks, which are similar to those described

above for the "Identify ... *required*" component of the Feature Creation tab.

6. Choose different features for the selected feature's *excludes* relationships. The Web interface carries out the needed validity checks, which are similar to those described above for the "Identify ... *excluded*" component of the Feature Creation tab.

Given a correct feature model stored in the database, the Feature Modification form thus modifies an existing feature in valid ways and inserts it back into the feature model while preserving the correctness of the model.

The *Feature Deletion* form enables an existing feature to be deleted from the feature model. This Web form asks the user for information about the feature and the how to modify the feature model to compensate for its removal, validates the information, and, once complete, stores feature description back into the RDB tables. This form requires the user to carry out the following sequence of actions:

1. Enter the name of a feature to be deleted. The Web interface, of course, only allows an existing feature to be deleted.

2. Choose what to do with the selected feature's children (if any). The choices are (a) to delete them along with the selected feature or (b) to reassign them to a different parent with possibly different relationship types. For children that are reassigned, the Web interface rechecks the validity of all the *requires* and *excludes* constraints in the reattached feature and in all of its descendant features. If there are invalid cross-tree constraints, the interface displays a warning message beneath the feature with the incorrect constraint.

Given a correct feature model stored in the database, the Feature Deletion form thus removes an existing feature while preserving the correctness of the model.

Second, we implemented the interactive Web interface design using appropriate client-side and server-side programs. The implementation updates the feature model (as encoded in the RDB) to accurately reflect the user's requests while also ensuring that the feature model stays syntactically and semantically correct (i.e., valid). A key aspect of ensuring correctness is validation of the data being entered or selected by the user as described above.

The client-side implementation uses HTML5, CSS, JavaScript, jQuery, AJAX, and the Bootstrap CSS framework. It uses HTML5's builtin validators to ensure the correctness of data such as the format of feature names.

The server-side implementation uses PHP and MySQL. This implementation ensures the following:

- The uniqueness of a feature name.
- The consistency of the relationship types. For example, if attaching a feature to a group of *alternative* features, then the user cannot choose an *OR* relationship, and vice versa.
- The correctness of the *requires* relationships. Because the parent, ancestor, and descendant features cannot be *required* by a feature, these are not provided in the list of possible features to be displayed by the client-side form.
- The correctness of the *excludes* relationships. Because the parent, ancestor, and descendant features cannot be *excluded* by a feature, these are not provided in the list of possible features to be displayed by the client-side form.

Third, we tested the implementation by using using it to create new features, modifying existing features, and deleting existing features, including assigning orphan features to new parents.

In this chapter, we have thus demonstrated that the answer to Research Question 2 is "Yes". We have proposed a novel design based on mainstream Web and relational database concepts. Our design uses three dynamic Web forms that incrementally construct feature models by interactively gathering information about the features and their relationships from the user [117]. These feature models are stored in a relational database designed according to our approach [116] described in Chapter 3. We have implemented the Web interface design using mainstream relational database and Web technologies. The implementation validates its inputs and ensures that the feature model stored in the database is syntactically and semantically correct at any time.

Chapter 5 introduces the live-preview page, an extension to the Web interface presented in this chapter. The live-preview page displays the feature model as a directory structure and allows the user to select a combination of features to customize and configure into a specific product.

# CHAPTER 5

# CONFIGURING VALID PRODUCTS

This chapter addresses specific Research Question 3 from Section 1.3: ***Can mainstream Web and relational database technologies be used to configure correct products corresponding to a feature model?***

To answer this question, we show how to represent feature models as composite directory list controls in Web forms. The list presents all features in a feature model in addition to their parent-child and cross-tree constraints relationships. The objective of this novel approach is to interpret the syntax and semantics of feature models (as described in the previous chapters) and generate Web forms that enable the selection of any valid combination of features. A form should interactively guide users to configure valid members of the product family represented by a feature model.

We published preliminary versions of this work in 2017 [116] and 2020 [117].

## 5.1 Product Configuration in SPL Development Phases

So far, we have primarily considered the *upper-left quadrant* of Figure 2.3 on page 19: how to represent the problem space during the domain engineering process. Section 2.2 defined the syntax and semantics of feature models. Chapter 3 followed by demonstrating how to encode feature models in relational database tables. Then, Chapter 4 demonstrated how to construct syntactically and semantically correct feature models using Web forms.

The feature model also provides the structure for the domain implementation (i.e., the *upper-right quadrant* of Figure 2.3). We plan to address that issue in the future, but it is beyond the scope of this dissertation research project.

Using a feature model, how can we address the *lower-left quadrant* of Figure 2.3, 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?

This chapter answers that question by extending the Web interface design given in Chapter 4 with a new *live-preview* Web form. The content of this new form presents the current structure of a feature model as a directory list. The form is updated continuously during feature construction and modification to reflect the changes caused by the user's activities. This form also enables the user to configure specific products from a feature model by selecting features and defining their relationships using the form's controls.

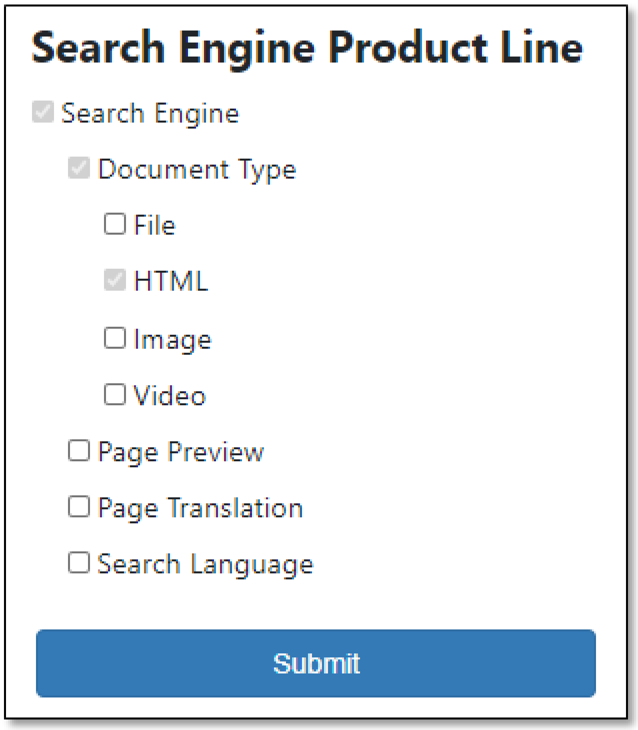
## 5.2 Product Configuration

A valid product from an SPL is one that conforms syntactically and semantically to the SPL's feature model. The feature model specifies all valid combinations of features. As described in Chapter 3, our approach encodes a feature model conveniently in a relational database. Thus the product configuration process must enable application engineers to select any possible combination of features from the database and then validate the selections made.

### 5.2.1 Live-Preview Product Configuration Form

As a proof of concept, we developed a Web application 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 live-preview Web form that shows the feature model and enables the selection of any valid combination of features. The application's current implementation uses PHP, MySQL, and SQL on the server side and HTML, CSS, and JavaScript on the client-side.

Figure 5.1 shows a form generated by the Web interface. This example lists all choices that are initially available for selection from the **Search Engine** product line shown in Figure 3.5. A Web form generated for a feature model shows:


Search Engine Product Line

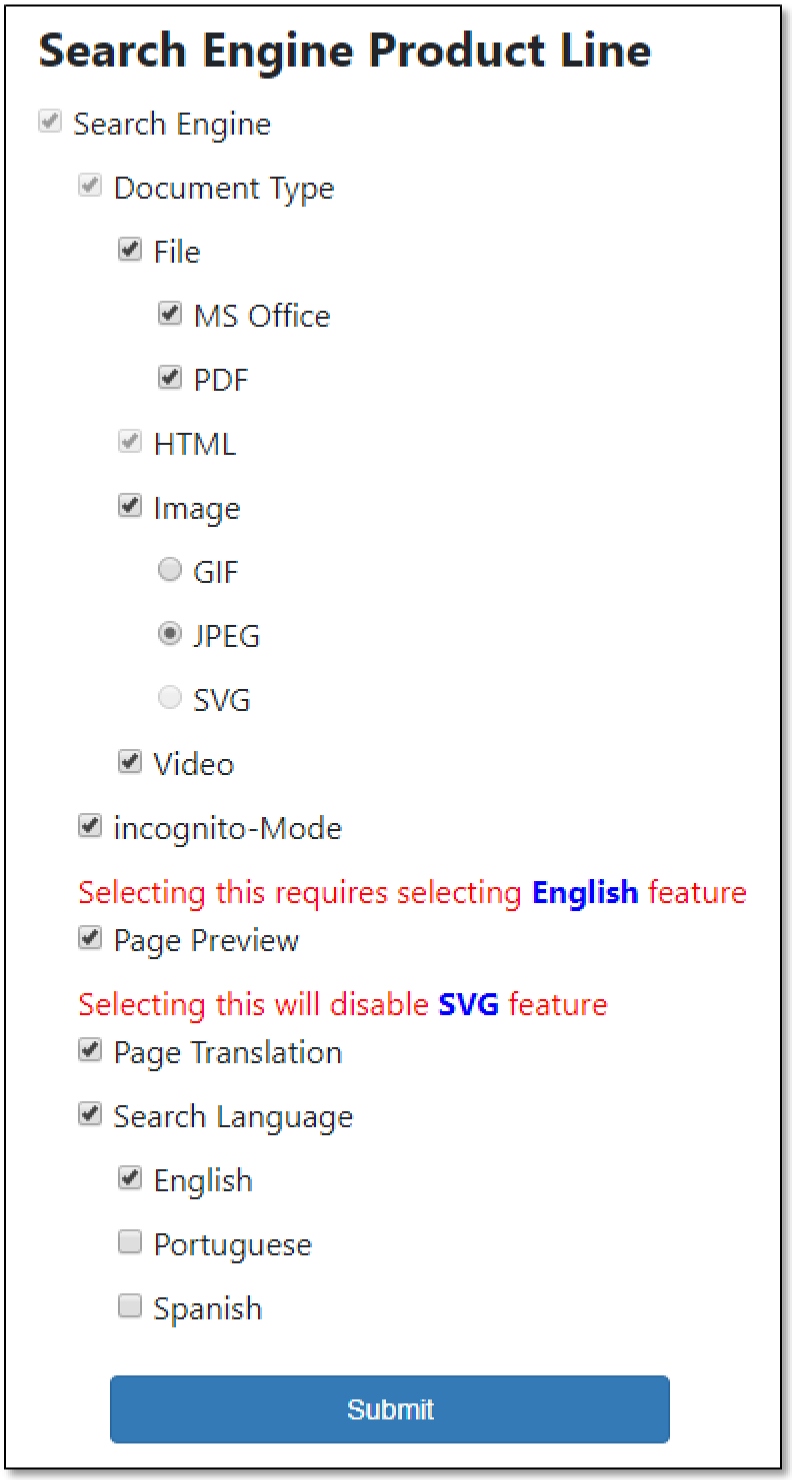
- [✓] Search Engine
  - [✓] Document Type
    - [ ] File
    - [✓] HTML
    - [ ] Image
    - [ ] Video
  - [ ] Page Preview
  - [ ] Page Translation
  - [ ] Search Language

Figure 5.1. Generated Directory List Representing the Feature Model

- *Mandatory* features as pre-selected checkboxes because they must exist in every configured product
- *Optional* features as checkboxes that enable the user to either include that feature in or exclude it from the configured product

The form in Figure 5.1 shows the checkboxes for the root **Search Engine**, its *mandatory* child feature **Document Type**, and its *mandatory* grandchild feature **HTML** as selected. The form also shows the checkboxes for the three *optional* children of the root—**Page Preview**, **Page Translation**, and **Search Language**—and the three optional children of **Document Type** as being available for selection. No other feature is currently shown; they are not available for selection because they are descendants of *optional* features.

Figure 5.2 shows the Web form for the **Search Engine** product line after the user has selected the **File**, **Image**, and **Search Language** features. The form has expanded the feature model's structure to show the controls for the children of the newly selected features,



Search Engine Product Line

[✓] Search Engine
- [✓] Document Type
  - [✓] File
    - [✓] MS Office
    - [✓] PDF
  - [✓] HTML
  - [✓] Image
    - ○ GIF
    - ● JPEG
    - ○ SVG
  - [✓] Video
- [✓] incognito-Mode

Selecting this requires selecting **English** feature

- [✓] Page Preview

Selecting this will disable **SVG** feature

- [✓] Page Translation
- [✓] Search Language
  - [✓] English
  - [ ] Portuguese
  - [ ] Spanish

Figure 5.2. Expanded Features in Feature Models and Their Relationships

indenting them appropriately. Note that this specification also includes the checkbox for the **Incognito-Mode** feature created in Figure 4.4. The Web form generated for a feature model shows:

- *OR* relationships as a group of checkboxes so the user can select one or more child features from a group
- *Alternative* relationships as a group of radio buttons so the user can select only one child feature from the group

As we discussed in Chapter 4, the Web interface for feature model construction checks for mutual exclusivity of the cross-tree constraints as it builds the model. However, it cannot detect all semantic errors involving features that might or might not be selected. For example, consider a feature X that is *required* by some optional feature A and *excluded* by some other optional feature B. Suppose that both features A and B are selected to be in a particular product. Should feature X be included or excluded? In this case, the product configuration interface notifies the user of this semantic error and allows the user to modify this relationship during product configuration. The interface detects this error by checking the feature's relationships (stored in RDB tables). If the feature is linked with both the *requires* and *excludes* relationships, then the Web form flags that incident and notifies the user when configuring the product. It gives the user the choice of selecting either feature A to require feature X or feature B to exclude it.

A generated Web form does not show the *requires* and *excludes* relationships as a part of the directory list structure. Instead it displays an explanatory message beneath the selected control, showing a warning message in *orange* and an error message in *red*. The intention of a warning message is to guide the user to make appropriate choices. The intention of an error message is to alert the user that the configuration is incorrect.

In the **Search Engine** example, the **Search Language** feature *requires* the **Page Translation** feature. If a user selects **Search Language** first, the Web form displays a warning message explaining that this feature *requires* the selection of **Page Translation**. If **Page Translation** is selected first, then the Web form does not show a warning message because the *requires* relationship is in one direction.

In the example, the **Page Preview** feature *excludes* the **SVG** feature. In this case, if the user selects either one, the Web form displays a warning message specifying this rule and disables the selection of the other feature. The Web form validates the selected choices when the user clicks the **Submit** button to submit the configuration. The validation is based on the feature model's syntax and semantics as encoded in the database. If the user submits the configuration while ignoring the warning messages, the system detects this by preventing the submission and showing the error messages to alert the user that the configuration is incorrect, thus preventing the user to submit until fixing the error.

### 5.2.2 Algorithms for Live-Preview Form

Figures 5.3 and 5.4 show the algorithms for interacting with the database tables, fetching the data, and displaying a Web form as a directory list for product configuration.

The first algorithm, the *Product Configuration Algorithm* shown in Figure 5.3, has an input argument that provides the database system credentials. The output is the product configuration interface shown as a Web form structured as a directory list structure. The algorithm accesses the database tables that encode the **Search Engine** product line.

The algorithm starts by verifying the user's *credentials* to connect to the database. If not successful, it throws an exception and exits. If successful, it performs an SQL query to retrieve the *root*. The root is a conceptual node programmed to equal `Null` (i.e., it has no parent). The feature model's first feature (i.e., the top node in the tree) is a child of root. The reason for this extra feature (root) is to help when searching for features (i.e., if the feature parent is null, then it's the top and starting node in the feature model tree). The algorithm then iterates through the database rows returned and assigns the variable *toFeature* to the returned value at line 6. *toFeature* in this case is the **Search Engine**

```
Web Form Creation to Create a Configured Product
Data: servername, username, password, dbname
Output: Web form constructed from the feature model which is stored in the database
1 connection ← mysqli_connect(servername, username, password, dbname);
2 if !connection then
      // wrong credentials
3     return;
4 sql ← “SELECT * FROM featuresRelations WHERE fromFeature = ‘root’ ";
5 result ← mysqli_query(connection, sql);
6 while row ← mysqli_fetch_assoc(result) do
7     toFeature ← row[‘toFeature’];
      // call displayfeature algorithm
8     displayfeature(toFeature, ‘root’ ,0 , row[‘relationType’], true);
```

Figure 5.3. Product Configuration Algorithm

feature.

The algorithm then calls the *displayfeature* algorithm shown in Figure 5.4. This algorithm has the following parameters:

- *toFeature*, the top node feature in the feature model
- *startNode*, the root conceptual feature (i.e., the parent of the top-level node)

  The initial argument is a string `"root"`, which is the parent of the top node.
- *depth*, the indentation level for displaying the feature
- *relationType*, the type assigned to the parent-to-child relationship

  Initially this is the relationship between the *root* and the **Search Engine** concept node.
- *showChildren*, a Boolean whose value determines whether to display the feature's children (expand the list) or not (collapse)

The *displayfeature* algorithm receives the top node data as arguments. It starts by checking whether the *relationType* argument is one of the integer values defined in the **Relationship** table, as shown in Figure 3.7. If the integer passed is 4 or 5, which represent the

```
Displaying Feature Model As Directory-List Web Form
Data: toFeature, startNode, depth, relationType, ShowChildren
Output: Feeds Product Configuration Algorithm, shown in Figure 5.3, with proper arguments to
        draw features
if relationType == 4 OR relationType == 5 then
    // Cross-tree constraints.
    return;
Make sure startNode exists in the system;
Check if startNode is equal to fromFeature and relationType is requires or excludes;
Record relationships to be stored in data attributes for HTML;
Record ShowChildren to be used in JavaScript;
check relations;
if relationType == 1 then
    // mandatory feature.  Disabled, checked, and its children are shown
    CSS checked and disabled for HTML element;
    ShowChildren ← true;
if relationType == 2 then
    // OR group
    type ← 'radio';
type optional is default for optional and alternative features;
type ← 'checkbox';
Indent form element based on depth parameter;
echo an HTML div with appropriate info for JavaScript, CSS, and HTML;
Use SQL to fetch all from featuresRelations table where fromFeature is equal to startNode;
Assign toFeature to fetched result;
Increment depth for Form indentation structure;
recursive call;
displayfeature(toFeature, startNode, depth, row2['relationType'], ShowChildren);
```

Figure 5.4. Display Feature Algorithm

*requires* and *excludes* relationships, respectively. Then the relationship between the parent and child is a *requires* or *excludes*. In this case the algorithm exits with no results because such features cannot be added to the parent-child structure being constructed.

The algorithm checks whether the *startNode* passed is a *valid* feature in the database. The first feature passed as *startNode* must be the top node after the root, which is the **Search Engine** node. The algorithm then performs recursive calls on children down to the leaves.

## 5.3 Evaluation and Conclusion

This chapter addresses Research Question 3 from Section 1.3: ***Can mainstream Web and relational database technologies be used to configure correct products corresponding to a feature model?***.

This chapter builds on the research reported in the preceding chapters. Chapter 2 defines traditional feature models. Based on that definition, Chapter 3 presents a design that accurately encodes these feature models in relational database tables. Building on this database design, Chapter 4 then designed a Web interface that enables users to construct syntactically and semantically correct feature models and store them in the database.

To answer Research Question 3, this chapter extends the Web interface design from Chapter 4 with a new *live-preview* Web form design that supports product configuration. The content of this new form presents the current structure of the feature model as a directory list. The form is updated continuously during feature construction and modification to reflect the changes caused by the user's activities. This form also enables the user to configure specific products from the feature model by selecting features and defining their relationships using the form's controls.

Given a syntactically and semantically correct feature model stored in the relational database, a live-preview Web form must have the following properties and behaviors:

- It does not change the feature model.

- It shows a subset of the features from the feature model and shows all the relationships among them. It does not show any feature or relationship unless it is in the feature model.

- Using an HTML directory list, it shows all features from the feature model that have been selected or deselected for the current product or are currently available for selection. It does not show any other features or relationships.

  Note: As described in the *excludes* item below, a feature that is *excluded* by some selected feature shows as deselected, but it has been deactivated so it is not available for selection.

- If a parent feature is selected, the form shows all of the parent's children as selected or deselected for the current product or as currently available for selection. The top-level feature is permanently selected.

- It represents a *mandatory* feature as a checkbox that is permanently selected.

- It represents an *optional* feature as a checkbox that is initially deselected but that can be subsequently selected.

- It represents an *OR* group as a group of checkboxes, all of which are initially deselected. Each checkbox can be subsequently selected independently of the others in the group.

- It represents an *alternative* group as a group of linked radio buttons, all of which are initially deselected. When one of the buttons is selected, the others remain deselected. One of the radio buttons in the group must be selected to configure a complete product from the feature model.

- If a selected feature *requires* another feature, then the form displays a warning message beneath the feature's checkbox or button to remind the user about this cross-tree constraint. The user must manually select the feature before the configuration can be completed correctly.

- If a selected feature *excludes* another feature, then the form displays a warning message beneath the feature's checkbox or button to remind the user about this cross-tree constraint. The *excluded* feature is deactivated; that is, it shows as deselected, but it is not available for selection by the user. If the excluded feature is not available to be deselected, then the form displays a warning message.

- If the same feature is both *required* and *excluded*, then the form displays a warning message and asks the user to resolve this conflict by choosing whether to include the feature or not.

- When the user triggers the **Submit** button, the Web interface checks to make sure the product configuration is complete and correct. If it is not, the interface displays appropriate error messages and waits for the user to correct the errors. To be complete, a radio button in every alternative group must be selected, all the *requires* and *excludes* cross-tree constraints must be satisfied, and all conflicts between *requires* and *excludes* resolved.

The product configuration consists of the set of all selected features shown in the form at the point of a successful submission. Because the form executes according to the above properties and behaviors, this set of features thus corresponds to a correct product from the product line represented by the feature model.

Given the above design, we then implement a server-side Web application that traverses the feature model's graph and generates the live-preview Web form. The server-side application is implemented in PHP and uses SQL to access and manipulate the MySQL database storing the feature model. The generated (client-side) Web form is implemented in HTML5, CSS, JavaScript, jQuery, AJAX, and the Bootstrap CSS framework. In addition, it uses

HTML5's builtin validators to ensure the correctness of data such as the format of feature names.

Finally, we tested the implementation using the **Search Engine** SPL described in Chapter 3 (and pictured in Figure 3.5). We had encoded this feature model in the RDB using the Web interface from Chapter 4. Two of the live-form examples from the testing appear in this chapter as Figure 5.1 and Figure 5.2.

In this chapter, we have thus demonstrated that the answer to Research Question 3 is "Yes". We have designed and implemented a Web form that, given a syntactically and semantically correct feature model stored in the relational database [116, 117], enables the user to select any set of features from the feature model that corresponds to a correct configuration of a product.

# CHAPTER 6

# REPRESENTING FEATURE MODELS IN JSON

This chapter addresses specific Research Question 4 from Section 1.3: ***Can JSON technologies be used to represent feature models correctly and enable them to be exchanged in textual form?***

To answer this question, we explore a novel approach that encodes feature models using JavaScript Object Notation (JSON) [33, 73]. JSON is a simple, pervasive, machine-independent, text-based language that is commonly used for transmitting and storing structured data. Given its prominence in Web applications, most mainstream developers are familiar with JSON, and it is supported by many libraries and tools.

The contributions of this chapter include:

- *How to accurately encode feature models in JSON.* Section 6.3 defines the syntax and semantics of our JSON-encoded feature models.

- *How to translate valid RDB-encoded feature models to and from JSON.* Chapter 3 encodes feature models in relational database (RDB) tables. Section 6.4 specifies the algorithms to translate the RDB encoding to and from the JSON encoding.

- *How to ensure validity of JSON-encoded feature models while creating, modifying, and deleting features.* Chapter 4 presents algorithms for creating, modifying, and deleting features using the RDB encoding. Section 6.5 presents similar algorithms for the JSON encoding. Our approach aims to separate the feature concept from its implementation by using the JSON notation.

A preliminary version of this work appears in the proceedings of the ACMSE 2021 conference [118].

We introduced the CSV encoding in Section 3.3 as a simple exchange format among database systems. Why introduce a different exchange format in this chapter? CSV files have a simple, but quite general syntax. The specific syntax and semantics of the CSV encoding are just assumptions shared by the various programs that read and write that format. The JSON encoding in this chapter seeks to build on the richer, but still simple, JSON syntax to encode more syntactic and semantic information about feature models directly into the document, relying less on the shared assumptions among programs. In the future, we plan to extend this work to exploit JSON Schema [38, 60, 104] to express the syntax and semantics of feature models more completely.

Before we look at this research, let us first examine the background concepts and technologies in Section 6.1 and then define a novel feature model example in Section 6.2.

## 6.1 JSON in a Nutshell

JavaScript Object Notation (JSON) is a lightweight format designed for human-readable data interchange [33, 73]. It is a convenient format for publishing and exchanging data, as it combines the flexibility of the Extensible Markup Language (XML) with data structures such as records, objects, and arrays [13]. To manage JSON data, users can use schema languages such as *JSON Schema* [60], in addition to type abstractions provided by modern programming and scripting languages such as Swift and TypeScript [13]. Most programming platforms (e.g., JavaScript, PHP, Python, Ruby, Java, and .NET) also have libraries that support the parsing and formatting of JSON data.

### 6.1.1 XML Concepts

For purposes of comparison, consider XML [24, 51], the other well-known data-interchange language. Both JSON and XML are text-based languages that use Unicode encodings. Both have hierarchical structures that can conceptually be interpreted as tree structures with nodes and edges.

XML is "a metalanguage for creating markup languages" [84]. To design a language in the XML family, a designer must choose a specific set of names for the language's elements and attributes.

A *well-formed* XML *element* consists of either (a) an *empty-tag* or (b) a *start-tag* followed by the corresponding *content* and *end-tag* [24]. We can form an XML start-tag by enclosing its name in a pair of angle brackets (i.e., between characters "`<`" and "`>`"). We can form the corresponding end-tag by adding the character "`/`" following the opening angle bracket. For example, the start-tag for an element named "`nodeName`" is "`<nodeName>`" and the corresponding end-tag is "`</nodeName>`." The content of the element consists of all the text (if any) between the start-tag and the end-tag. Any XML elements occurring in the content must themselves be well-formed. An empty-tag for name "`nodeName`" has the form "`<nodeName/>`", where the character "`/`" precedes the closing angle bracket. As the name implies, an empty-tag has no content.

An XML *attribute* associates specific properties with an element. An attribute has the form "`name = value`", where "`name`" is the attribute's name and "`value`" is its value. The name must be a quoted string. A list of zero or more attributes may be added to start-tags and empty-tags but not to end-tags. The list appears after the element name and before the closing angle bracket. The order of the attributes in the list has no meaning. However, an attribute name may appear only once in the list.

Consider the example in Figure 6.1. It shows a list of two *customers* represented as a tree. The *customers* node is the *parent* of the *customer* nodes. Each *customer* node is, in turn, the parent of the *customerID*, *firstName*, *lastName*, and *Email* nodes. These *children* of the *customer* nodes have values.

To represent the list from Figure 6.1 using a simple XML language, we can choose the tree representation's node names as the element names and choose not to use attributes.

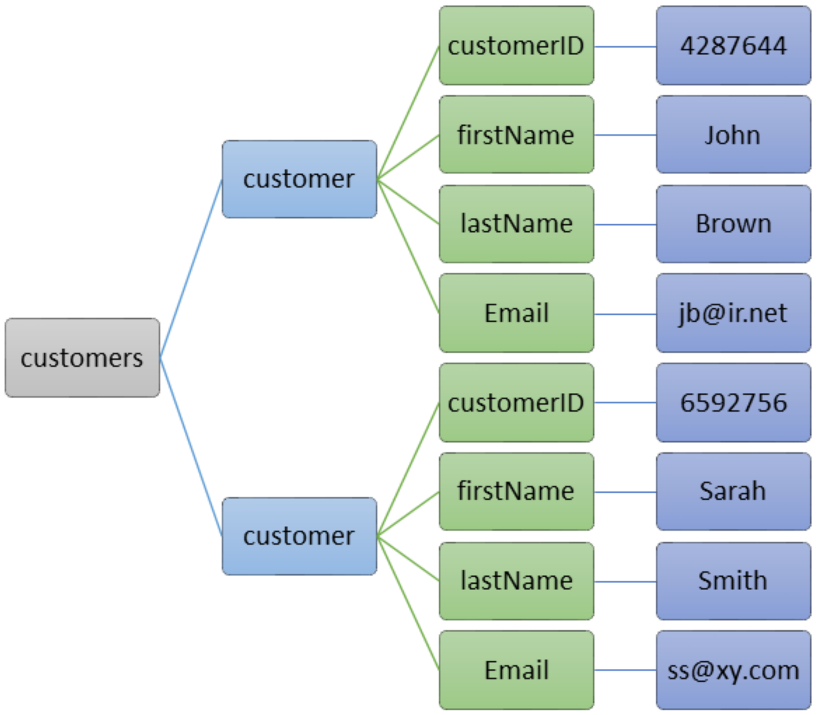


Figure 6.1. A List of Two Customers Represented as a Tree

```
<customers>
    <customer>
        <customerID> 4287644   </customerID>
        <firstName>  John      </firstName>
        <lastName>   Brown     </lastName>
        <Email>      jb@ir.net </Email>
    </customer>
    <customer>
        <customerID> 6592756   </customerID>
        <firstName>  Sarah     </firstName>
        <lastName>   Smith     </lastName>
        <Email>      ss@yx.net </Email>
    </customer>
</customers>
```

Figure 6.2. A List of Two Customers Represented as a Tree in XML

```
<json>       ::= <array>    |  <object>  |  <primitive>
<primitive> ::= <string>   |  <number>  |  <boolean>
                | 'null'
<boolean>    ::= 'false'  |  'true'
<array>      ::= '[' ']'  |  '[' <seq> ']'
<seq>        ::= <json>  |  <json> ',' <seq>
<object>     ::= '{' '}'  |  '{' <pairlist> '}'
<pairlist>  ::= <pair> ',' <pairlist>
<pair>       ::= <name> ':' <json>
<name>       ::= <string>
```

Figure 6.3. JSON Syntax in BNF

The snippet in Figure 6.2 shows a possible XML representation of the list. This XML example includes the XML elements "`customers`", "`customer`", "`customerID`", "`firstName`", "`lastName`", and "`Email`" with three levels of nesting. The elements at the most deeply nested level have plain text content.

### 6.1.2 JSON Concepts

Now consider JSON. Like XML, JSON is a textual language for data interchange [33, 73]. Like XML, JSON is a "metalanguage" than can be specialized to represent custom languages within the larger family of languages. However, unlike XML, JSON has a relatively simple syntax that should be familiar to most programmers. It is more or less based on a subset of the JavaScript programming language. The syntax is also easy for machines to parse.

We can express the general syntax of JSON in Backus-Naur Form (BNF) as Figure 6.3 shows, where the lexical tokens `<number>` and `<string>` are defined similarly to those in C or Java. Whitespace can be inserted between any pair of tokens (or at the beginning or the end of the JSON document). (We constructed this BNF specification from the syntax diagrams at `http://json.org` [33].) A JSON document consists of an *array*, an *object*, or a *primitive value* [33, 73]. A JSON array consists of a sequence of zero or more JSON *values* enclosed in square brackets and separated by commas. A JSON *object* consists of a collection of zero or more *name-value pairs* (also called *properties*) enclosed in curly braces and separated

```
{   "customers": {
        "customer":
            [
                {   "customerID":  "4287644",
                    "firstName":    "John",
                    "lastName":     "Brown",
                    "Email":        "jb@ir.net"
                },
                {   "customerID":   "6592756",
                    "firstName":    "Sarah",
                    "lastName":     "Smith",
                    "Email":        "ss@yx.net"
                }
            ]
        }
}
```

Figure 6.4. A List of Two Customers Represented as a Tree in JSON

by commas. Each name and its corresponding value are separated by a colon. A name must be enclosed in a pair of double quotation marks, and it should be unique within the object's collection. A JSON value can be any JSON array, object, or *primitive value*. The set of primitive types is limited to strings, numbers, Booleans (i.e., "`false`" and "`true`"), and "`null`". A string value is a sequence of zero or more characters enclosed in a pair of double quotation marks. The numbers include both integer and floating point formats.

The code snippet presented in Figure 6.4 represents the list shown in Figure 6.1 in a JSON format. The JSON document consists of an object having just one name-value pair. This pair maps the name `"customers"` to a value that is an array. The array includes two values, both of which are objects with the same structure. Each object has four name-value pairs that map the names of fields to their string values.

### 6.1.3 Comparing JSON and XML

Which is better for our purposes, JSON or XML?

Various researchers compare JSON and XML. For example, Zunke et al. [133] compare the performance of the two notations and ul Haq et al. [125] analyze them comprehensively in the context of Web technologies. The main advantages of XML over JSON are:

- XML is preferable for complex structures and validation. XML is more suited for applications manipulating various data types [25].
- XML has mature standards for expressing the structure of the document such as XML schema. These standards enable the XML document to be validated [130].
- The XML schema tools [4] are more mature for document validation than the JSON Schema tools are [60]. The JSON Schema standard is still in work (at draft 8 [60]) and thus the tools are experimental.

The advantages of using JSON over XML are:

- JSON is widely supported and requires no use of add-on software libraries [115].
- JSON is less verbose than XML.
- JSON is faster. That is, JSON documents can be parsed quickly and easily compared with the slow, cumbersome parsing of XML documents.

JSON is simpler than XML. It is a simple, text-based language that represents data using a nested combination of data structures common to most programming languages, sets of name-value pairs and sequences of values. It is both readable by humans and easy for computers to parse and map to and from internal data structures. JSON is thus a convenient notation for transmitting and storing structured data.

JSON is better supported by client-side Web software than XML. It is essentially a subset of JavaScript, which is supported by all browsers. By using JSON, we avoid the need for add-on libraries to access data from a browser's Document Object Model (DOM). JSON is estimated to parse up to one hundred times faster than XML in modern browsers [16, 61]. Given its prominence in Web applications, it is a good choice for our research; most mainstream developers are familiar with JSON and it is supported by many libraries and tools.

For our purposes, JSON has several advantages over XML. So, we tentatively adopt it to express our feature models as structured text to enable them to be conveniently exchanged and archived. However, JSON also has several shortcomings that make some aspects of this research difficult, requiring us to devise workarounds. JSON supports fewer data types than XML or general purpose programming languages. The JSON Schema is not yet an unambiguous, finalized standard, which means that validation programs may not be consistent in their results for some border cases [104]. It also has mixed support for name uniqueness specifications. It can specify that the items within an array must be unique, but it does not support similar specifications for object and property names.

## 6.2 Raster/Vector Processing Feature Model

In this section, we formulate a new feature model that we use in this and the following chapters. Figure 6.5 shows an example feature model for a raster/vector image manipulation SPL. This simple feature model uses the Geospatial Data Abstraction Library (GDAL) and OpenGIS Simple Features Reference Implementation (OGR) libraries [43] in Python 3.8. This feature model documents the common and variable aspects of a set of applications developed by the National Center for Computational Hydroscience and Engineering (NCCHE) at the University of Mississippi.

Figure 6.5 depicts an SPL with the product line concept *RasterVectorProcessing*. This feature indicates the purpose of the SPL. The figure shows a mandatory *Library* feature with two children.

- The *GDAL* library, shown as a mandatory feature, is selected by default.
- The *OGR* library, shown as an optional feature, can either be selected or not selected to be in a product.

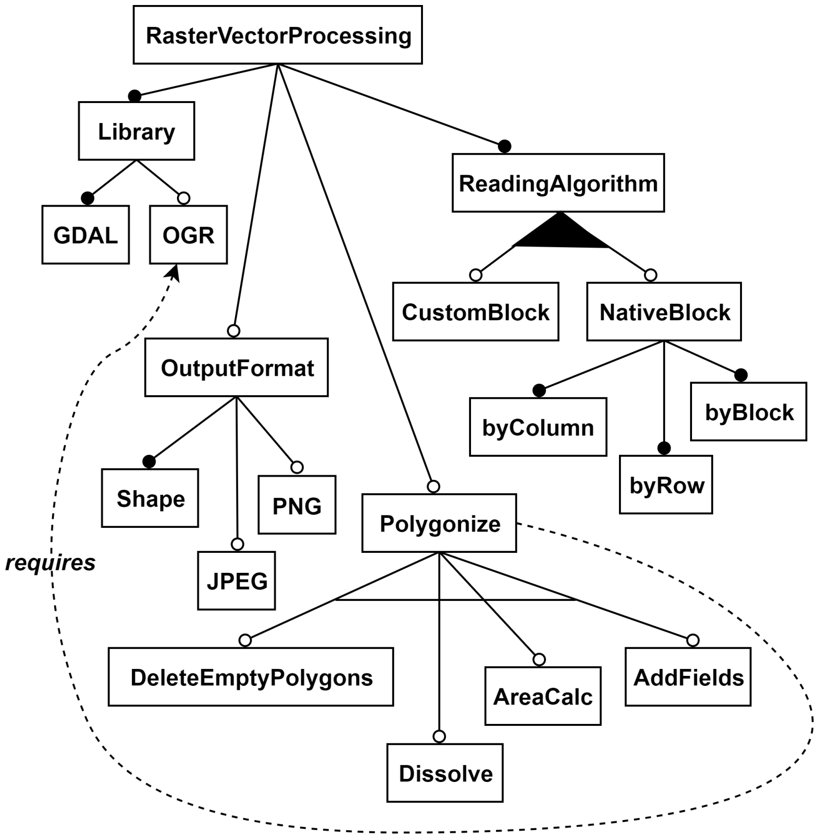


Figure 6.5. Feature Model for a Raster/Vector Image Manipulation SPL

The *RasterVectorProcessing* image manipulation SPL accepts raster files (as results of flood simulations) to perform calculations to determine flood hazard risks and potentially lethal flood zones. For small rasters, the SPL's *GDAL* feature includes *gdal_calc.py*, a command line raster calculator that uses NumPy [99] array syntax. For larger rasters, the SPL offers the mandatory *ReadingAlgorithm* feature for reading raster files. This feature offers two mechanisms in an OR relationship, enabling the user to select one or both.

- The *CustomBlock* feature encapsulates an algorithm that determines the best block size (tile) to read the rows and columns in a raster file, thus enhancing the read/calculate time.
- The *NativeBlock* feature uses whatever the raster's reading mechanism to read column-by-column, row-by-row, or using the native block size retrieved from the raster band.

The SPL offers an optional *Polygonize* feature, which converts the calculated raster areas into polygons and creates a shape file. This feature has four operations from which the user can select only one, because the children are grouped in an alternative relationship. These operations edit the shape file.

- *DeleteEmptyPolygons* deletes empty polygons from the shape file.
- *Dissolve* dissolves polygons and merges them into one larger polygon.
- *AreaCalc* calculates the area of a polygon.
- *AddFields* adds fields (e.g., id, description) to an outputted raster file.

These operations under the *Polygonize* feature *require* the *OGR* library for accessing and manipulating vector shape files

The SPL also offers an output format through the optional *OutputFormat* feature, which has the shape file as a default and two optional features to include *PNG* and/or *JPEG* output files.

## 6.3 Encoding Feature Models in JSON

This section presents the chapter's first contribution: how to accurately encode feature models in JSON.

The JSON-based language [61] defined in this chapter can serve as a precise medium for communication of feature models among independent tools and work sites. This language can allow these to work in isolation from each other and to communicate feature models among themselves using a portable, text-based format. It can make extending the system with future tools convenient and provide a system-independent format for archiving feature models.

Figure 6.6 shows a JSON encoding of part of the feature model from Figure 6.5. In this encoding, we represent a feature as a JSON object [61] with the following properties:

- `id`, which is the feature's unique name string
- `type`, which is the string `mandatory`, `optional`, or `root`
- `parent`, which is the feature's parent's name string
- `relation`, which is the string `OR`, the string `alternative`, or an empty string
- `requires`, which is an array of zero or more feature names
- `excludes`, which is an array of zero or more feature names
- `children`, which is an array of zero or more feature objects

As we see in Figure 6.6, the outer layer of the JSON structure for a feature model is an object representing its concept (root) node. This feature always has the value of its `type` property set to `root`, its `relation` property set to an *empty string*, and its `requires` and `excludes` properties set to *empty arrays* (i.e., `[]`). No other feature can have type `root`. Its `children` property is set to an array holding its child features.

```
{
    "id": "RasterVectorProcessing",
    "type": "root",
    "parent": "",
    "relation": "",
    "requires": [],
    "excludes": [],
    "children": [
      {
        "id": "Library",
        "type": "mandatory",
        "relation": "",
        "requires": [],
        "excludes": [],
        "children": [
          {
            "id": "GDAL",
            "type": "mandatory",
            "relation": "",
            "requires": [],
            "excludes": [],
            "children": []
          },
          :::::
        ]
      },
      {
        "id": "Polygonize",
        "type": "optional",
        "relation": "",
        "requires": [
          "OGR"
        ],
        "excludes": [],
        "children": [
          {
            "id": "DeleteEmptyPolygons",
            "type": "optional",
            "relation": "alternative",
            "requires": [],
            "excludes": [],
            "children": []
          },
          :::::
        ]}]
}
```

Figure 6.6. Example of a JSON-encoded Feature Model

```
function encodeFM(current_feature)
    examine the current_feature in the feature model
        let id, type, parent, relation, requires, and excludes
            be current_feature's JSON property values
        let childrenFM be an array of all the child features
            for current_feature
        let children be the array of JSON objects resulting
            from applying encodeFM to each element of childrenFM
        return the JSON-encoded feature model object with the
            properties id, type, relation, requires, excludes,
            and children as described above

function encode_all(feature_model)
    if feature_model is empty then
        return '{}'
    else
        return encodeFM(root_of(feature_model))
```

Figure 6.7. JSON `encodeFM` Function

In the JSON encoding, the arrays `requires`, `excludes`, and `children` denote sets. They cannot have repeated elements.

The feature names in a `requires` or `excludes` array must be `id`s for defined features that do not have the type `mandatory`. Mandatory features are preselected and cannot be deselected when configuring a product. In addition, a feature can neither require nor exclude one of its ancestors in the feature model.

Now, let us consider how to encode the entire feature model in JSON. We can describe that process with the recursive function `encodeFM` defined in Figure 6.7. Function `encodeFM` takes an arbitrary feature `current_feature` from a valid feature model and returns that feature and all its descendants encoded in the JSON structure described in Figure 6.6. If we apply the process described by `encodeFM` function to the top-level feature in a valid feature model (e.g., as depicted by a valid feature diagram), we can construct the encoding of the entire model. In Figure 6.7, we show this as the `encode_all` function. It is easy to see that the feature model's JSON encoding (returned by `encode_all`) is equivalent to the original feature model.

A JSON-encoded feature model must conform to the feature model's syntax and semantics. In the future, we plan to define an appropriate JSON Schema [60, 104] to be able to validate much of the feature model encoding using standard JSON validators (e.g., Ajv [105]). However, JSON Schema cannot express some constraints such as the uniqueness of feature names within the model and the restrictions on the cross-tree relationships. For these aspects, we expect to need a custom semantic validator.

## 6.4 Translating Feature Models

This section presents the chapter's second contribution: how to translate valid RDB-encoded feature models to and from JSON.

This chapter describes an approach to feature modeling with similar goals to our approach in Chapter 3. Chapter 3 uses a mainstream relational database (RDB) to encode a feature model as a directed acyclic graph. For the purposes of this chapter, that design consists of three tables.

- The **feature** table defines the set of features, representing each feature by a unique `id`.
- The **featuresRelations** table specifies the `relationType` between features `fromFeature` and `toFeature`. The relation types include hierarchical (`mandatory`, `optional`, `OR`, and `alternative`) and cross-tree (`requires` and `excludes`) relationships.
- The **Relationships** table lists the static set of possible relationships between features.

In addition, Chapter 4 specifies algorithms that generate a dynamic, Web-based user interface that enables users to construct and modify valid RDB-encoded feature models. Similarly, Chapter 5 specifies algorithms that extend the user interface to enable users to configure valid products.

This chapter seeks to provide a JSON encoding for feature models that can also serve as an exchange and archival mechanism for the RDB-encoded feature models. This section presents this chapter's second contribution: how to translate valid RDB-encoded feature

```
rdbTojsonTranslator
  Data: valid RDB-encoded feature model
  Output: returns feature and all its descendants encoded in JSON
  function ENCODE(feature)
  if feature exists in RDB feature model then
      // fetch feature's id from feature table
      // fetch id's parent from toFeature column of featuresRelations table
      // fetch type of id-parent relationship from relationType column of
         featuresRelations
      // collect arrays of id's requires, excludes, and child feature
         relationships from featuresRelations
      // call ENCODE on each child feature and collect resulting JSON objects
         in children array
      // return JSON object with properties id, type, parent, relation,
         requires, excludes, children
  else
      // ERROR (should not occur)
  end function
```

Figure 6.8. RDB-to-JSON Feature Model Translator

models to and from our JSON-encoded models. Together, the two translators enable the RDB-based and JSON-based tools to be used as a part of an integrated system.

Figure 6.8 sketches the RDB-to-JSON translation algorithm. It is a recursive algorithm that does a depth-first traversal of the parent-child tree encoded in the RDB. During the traversal, it gathers information about the tree's nodes and edges that it subsequently uses to construct equivalent structures in the JSON-encoded tree.

If we apply the ENCODE function from Figure 6.8 to the root feature of a valid RDB-encoded feature model, then its return value is a valid JSON-encoded feature model that is equivalent to the RDB-encoded feature model.

If we assume that a JSON document correctly encodes a valid feature model (e.g., is an output of the RDB-to-JSON translator), the JSON-to-RDB translator works similarly to the RDB-to-JSON translator. (We leave the syntactic and semantic validation of JSON-encoded feature models for future work.) As shown in Figure 6.9, the algorithm traverses the JSON-encoded tree and gathers information about the tree's nodes and edges that it subsequently

```
jsonTordbTranslator
  Data: valid JSON-encoded feature model
  Result: adds JSON feature and all its descendants to RDB
  procedure DECODE(feature)
  if feature is a valid JSON feature object then
      // fetch id, parent, requires, excludes, and children from feature object
      // create new row of feature table for id
      // create new row of featuresRelations table with id in fromFeature,
         parent in toFeature, and type in relationType column
      // for each feature A that requires (or excludes) feature B, create new
         row in featuresRelations with A in fromFeature, B in toFeature, and
         requires (or excludes) in relationType column
      // call DECODE for each object in children array
  else
      // ERROR (should not occur)
  end procedure
```

Figure 6.9. JSON-to-RDB Feature Model Translator

uses to populate the **feature** and **featuresRelations** tables [116]. The **Relationships** table is a static table that is the same for all feature models.

If we call the DECODE procedure from Figure 6.9 with a valid JSON-encoded feature model as its argument and with an "empty" database, on return the database represents a feature model that is equivalent to the argument. By an "empty" database we mean that the **feature** and **featuresRelations** tables have no rows and that the **Relationships** table is prepopulated with the static definitions of the relationships.

## 6.5 Manipulating JSON-Encoded Feature Models

This section presents the chapter's third contribution: how to ensure the validity of JSON-encoded feature models while inserting, modifying, and deleting features.

In this section, we define operations to *create*, *modify*, and *delete* features. All three operations preserve the validity of the JSON-encoded feature model. If initiated with a valid model, each terminates with a valid model that has been updated appropriately. These JSON operations have the same functionality as the corresponding RDB operations defined in Chapter 4. In this section, we focus on the algorithm to create a new feature.

```
createFeature
   Data: name, type, parent, relation, requires, excludes, children
 1 newFeatureObj ← {'id': name, 'type': type, 'parent': parent, 'relation': relation, 'requires':
     requires, 'excludes': excludes, 'children': children}
 2 if feature is unique then
 3     if parent is empty string AND type == 'root' then
 4         numOfKeys ← get number of JSON object's keys
 5         if numOfKeys returns 0 then
 6             newFeatureObj ← {'id': name, 'type': 'root', 'relation': '', 'requires': '',
                 'excludes': '', 'children': children}
 7             write newFeatureObj to to JSON feature model file
 8             return
 9     if type is 'optional' or 'mandatory' AND relation is 'OR' or 'alternative' or "" then
10         if if parent is valid feature in feature model then
11             parJSON ← lookup parent object in the JSON structure
12             if parJSON does hasChildren then
13                 relationship ← parJSON.children.relation
14                 if relationship == relation then
15                     desArr ← getDescendants(parJSON, parent)
16                     ascArr ← getAscendants(parJSON, parent)
17                     mergeArr ← merging ascArr and desArr
18                     Push newly created feature's id and parent to mergerArr
19                     requireExclude(requires, 'requires', mergerArr)
20                     requireExclude(excludes, 'excludes', mergerArr)
21                     assign new feature to parent in newFeatureObj
22             else
23                 write newFeatureObj to JSON feature model file
```

Figure 6.10. Operation to Create a Feature

Figure 6.10 shows the *feature creation* algorithm that adds a new feature to the JSON-encoded feature model. Its inputs are the properties of a feature object as described in Figure 6.6.

The operation first verifies that the feature's name is unique among the defined features. Then the operation checks whether a parent feature is passed. If a parent is not passed and the model does not already have a root, then the new feature becomes the root (concept) node. If a root already exists, then the operation exits with an error. If a parent is passed and the parent feature exists, then the new feature becomes a regular child feature of that

```
getDescendants
  Data: parentObj
  Output: array of feature descendants up to root
1 listOfChildArray ← get list of parent's children
2 tempArray ← []
3 for ( item in listofChildArray ) {
4     tempArray.push(item)
5     childObj ← lookup item (child object) in JSON file
6     if child has property .children then
          // recursive call for child
7         getDescendants(childObj)
8 return tempArray
```

Figure 6.11. Algorithm to Get a Feature's Descendants

parent. If the parent does not exist, then the operation exits with an error. If no error has occurred, then the operation checks the correctness of the `type`, `relation`, and `parent` properties passed. If the feature model is empty, the user can leave out the `parent` property.

If the `parent` property is passed, the algorithm retrieves the `parent` object from the JSON-encoded model to determine what types of relationships exist between the parent and its children. The operation then checks whether the relationship matches what the user passes in the `relation` property. After passing these checks, the operation determines the newly created feature's ascendants and descendants by passing the parent object to two algorithms: *getDescendants* and *getAscendants*.

Figure 6.11 shows the *getDescendants* algorithm. It first stores the child features in an array. Then, for each item in the array, it checks whether that item has children. If the item does have children, the algorithm gets that item's object and then calls itself recursively with that object as its argument. The algorithm then returns an array that has all descendant features from the feature being created down to the leaves. The *getAscendants* algorithm has similar steps but instead, looks for the property `parent` instead of `children`.

The *create* operation (Figure 6.10) merges the arrays returned by the *getAscendants* and *getDescendants* algorithms and then passes the result to a third algorithm (shown in

```
requireExclude
  Data: requires or excludes arg, 'requires' or 'excludes' as strings, mergerArr
  Output: Require or exclude operation gets accepted and updated in JSON structure or an
          error is shown
1 for ( item in requires or excludes ) {
      // mergerArr contains ascendants, descendants, parent, created feature
2     if feature to get required/excluded not in mergerArr then
3         itemObj ← get feature's object from JSON structure
4         if itemObj exists in JSON structure then
              // check itemObj's property type to identify if it's mandatory or
                 optional
5             if itemObj.type == 'mandatory' then
                  // can't require or exclude mandatory features
6                 continue
7             else
8                 update properties requires and excludes in newFeatureObj defined in the
                    creation algorithm
```

Figure 6.12. Algorithm to Enforce Cross-tree Constraints

Figure 6.12) that enforces the `requires` and `excludes` constraints. This algorithm first iterates through the items in the require (or exclude) argument's array. If an item is in the merged array (which holds ascendants, descendants, parent, and the feature being created), the algorithm skips this item; otherwise, the algorithm continues to process the item. The next step is to retrieve the item's object from the JSON structure, if it exists. Then the algorithm checks the item object's `type` property to ensure that no mandatory feature is required or excluded. If the `type` property is optional, then the feature to be required or excluded passes all the checks and the algorithm pushes the update to the JSON structure. The algorithm skips any item whose `type` property is mandatory.

The *modify* operation is similar to the feature creation operation. When modifying a feature property such as `id`, `type`, or `relation`, the operation applies the same checks that are applied in feature creation, but no new JSON feature is created and stored. Instead, the operation first determines whether the feature to be modified actually exists in the JSON structure. If the feature exists, the operation retrieves its object and then checks the `id` property against the features in the feature model. After completing the requested modifications (if correct), the operation updates the object and stores it back in the JSON structure.

The *delete* operation takes one additional step. If the deleted feature is the root of the feature model, the operation allows the user to either create another root or delete the whole feature model. If the deleted feature has children, then the user has the choice of either assigning the children to another existing feature or deleting the feature along with all its descendants.

As a proof of concept, we implemented and tested these operations using both Python 3.8 and the JavaScript (ECMAScript 2017) in a Chrome browser version 87.0.4280.88. Both programs performed these operations on a JSON document that had been deserialized into a programming language data structure. After each operation, the updated JSON document was serialized back into an external file.

## 6.6 Evaluation and Conclusion

This chapter addresses Research Question 4 from Section 1.3: ***Can JSON technologies be used to represent feature models correctly and enable them to be exchanged in textual form?***

In this chapter, we demonstrate that the answer is "Yes". In this research, we first design the following:

1. A JSON-based language for encoding valid feature models.

   In Section 6.3, we define the syntax and semantics for a custom JSON language to represent "traditional" feature models and design an algorithm to encode a valid feature model (e.g., as represented by a feature diagram) in the language. We thus argue that the JSON encoding for the feature model is semantically equivalent to the original model.

2. Algorithms to translate a RDB encoding of a valid feature model to the JSON encoding and vice versa.

   In Section 6.4, we design two algorithms for translating back and forth between the RDB encoding of feature models defined in Chapter 3 and the JSON encoding defined in Section 6.3. Figure 6.8 defines the RDB-to-JSON translation algorithm and Figure 6.9 the JSON-to-RDB translation algorithm.

   Given that both the RDB and JSON encodings are equivalent to the conceptual feature model, they are also equivalent to each other. To evaluate the RDB-to-JSON conversion algorithm, we argue that it correctly maps from an arbitrary valid RDB-encoded feature model to the corresponding JSON-encoded feature model. Similarly, to evaluate the JSON-to-RDB conversion algorithm, we argue that it correctly maps from an arbitrary valid JSON-encoded feature model to the corresponding RDB-encoded feature model.

3. Algorithms for manipulating valid JSON-encoded feature models.

   In Section 6.5, we designed algorithms for creating new features and modifying and deleting existing features in JSON-encoded feature models. These algorithms have the same functionality as the algorithms given in Chapter 4 for the RDB encoding, except that these algorithms access the JSON data structure. If we use these algorithms to implement the Web forms defined in Chapter 4, then the Web forms must have the same properties and behaviors. Thus, the correctness arguments for the JSON algorithms are essentially the same as the one given in Section 4.3 for the corresponding algorithm from Chapter 4.

Second, we implemented the designs for the JSON encoding, the RDB-to-JSON and JSON-to-RDB translators, and the algorithms for creating, modifying, and deleting features. We implemented (and tested) these operations using both Python 3.8 and the JavaScript (ECMAScript 2017) in a Chrome browser version 87.0.4280.88. Both programs perform these operations on a JSON document that has been deserialized into a programming language data structure. After each operation, the updated JSON document is serialized back into an external file.

Third, we tested the above implementations. We first converted the RDB-encoded *Raster-VectorProcessing* feature model (shown in Figure 6.5) to JSON using the RDB-to-JSON translator. We then converted the JSON encoding back to an RDB encoding using the JSON-to-RDB translator. We also tested the create, modify, and delete operations by performing the operations on the JSON encoding. The feature model, stored in an external JSON file, was updated after each operation..

In this chapter, we have thus demonstrated that the answer to Research Question 4 is “Yes”. We have designed an approach that can encode an arbitrary “traditional” feature model accurately in a JSON document in a manner that is equivalent to the RDB encoding defined in Chapter 3. We have also designed and implemented programs that can translate a valid RDB encoding of a feature model to an equivalent JSON encoding and vice versa. In addition, we have operations to create, modify, and delete features in a JSON-encoded feature model.

# CHAPTER 7

# ENCODING FEATURE MODELS IN DOCUMENT-ORIENTED DATABASES

This chapter addresses specific Research Question 5 from Section 1.3: ***Can a document-oriented NoSQL database be used to accurately encode feature models?***

To answer this question, we explore a novel approach that encodes valid feature models for storage in document-oriented databases and preserves the model's validity while creating, modifying, deleting, and extracting information about features. We use MongoDB, a source-available, cross-platform, document-oriented database system.

A preliminary version of this work appears in the proceedings of the ACMSE 2021 conference [118].

## 7.1 Document-Oriented Databases in a Nutshell

Since the 1970s, relational databases (RDBs) have been the most prominent approach to organizing large data collections [128]. Following this approach, in Chapter 3 we describe how to use the rows and columns of three RDB tables to encode feature models.

However, in recent years, a number of alternative storage structures have emerged. These are often grouped under the broad term NoSQL [74]. In this chapter, we investigate the type of NoSQL databases called document-oriented databases. In Chapter 8, we investigate another type called graph databases.

A *document-oriented database* (also called a document store) is useful for storing semistructured data [1, 26] sets—that is, data sets that lack the stable tabular structure needed by RDBs but exhibit some useful, perhaps evolving, internal structure. For example, a document-oriented database may be used to store hierarchical structures such as those expressed in JavaScript Object Notation (JSON) [33, 73] or Extensible Markup Language (XML) [24, 51].

In this chapter, we investigate the document-oriented database MongoDB [15, 80], which stores data in Binary-JSON (BSON) documents with optional schemas. BSON is the binary representation of JSON-like documents that MongoDB uses to store data. MongoDB organizes the documents into collections (in contrast to the tables used in RDBs like MySQL); one database can contain many collections [80]. Our approach encodes a feature model as BSON documents and manipulates the models using the MongoDB Query Language (MQL).

MQL is a rich query language for fetching and manipulating documents. It includes the usual CRUD (Create, Read, Update, and Delete) operations plus text search, geospatial, and other useful queries [80].

A feature model is primarily a tree structure. MongoDB is thus a good choice for representing and storing feature models. We can store tree structures in MongoDB using the following patterns (or data models) [80]:

- *Model Tree Structures with Parent References* pattern, which organizes the documents into a tree-like structure with a parent reference associated with each child's document

- *Model Tree Structures with Child References* pattern, which organizes the documents into a tree-like structure with child documents attached to the parent's document

- *Model Tree Structures with Array of Ancestors* pattern, which organizes the documents into a tree-like structure using an array to record the path from the node back to the root

To represent the hierarchical nature of feature models using MongoDB model tree structures, we choose to apply the first two patterns, using model tree structures with both parent and child references. The third pattern, which lists an array of ancestors, is helpful for some feature model query operations, but it requires that the ancestor array exist at all times.

This makes importing a complete database difficult. We do that for the experiments in Chapter 9.

In the following section, we demonstrate that it is possible to encode feature models in MongoDB databases using the model tree structures it provides.

## 7.2 Encoding Feature Models in MongoDB

We encode feature models in MongoDB databases using a collection of documents equivalent to the RDB encoding's **featuresRelations** table (and, hence, equivalent to the CSV encoding) defined in Chapter 3. Each document in the collection specifies the same unique relationship between two features given by a row in **featureRelations** table (or by a line in the CSV file). That is, the document corresponds to a directed edge in the conceptual feature diagram. It consists of the following three properties:

- `fromFeature`, whose value is a valid feature name string that denotes the "parent" feature of the relationship
- `toFeature`, whose value is a valid feature name string that denotes the "child" feature of the relationship
- `relationType`, whose value is an integer code in the inclusive range 0 to 5 that denotes the relationship between and `fromFeature` and `toFeature`

The meanings of the `relationType` values are the same as given in the RDB encoding's **Relationships** table.

Figure 7.1 shows a part of a MongoDB collection that encodes the feature model for the Raster/Vector Image Manipulation SPL from Figure 6.5. It shows the features *Raster-VectorProcessing* and *VisualizingData* as separate documents. Each document refers to the other using its parent (`fromFeature`) or child (`toFeature`) property while the `relationType` property describes the relationship between the child feature and its parent. In the example, the relationship type `0` indicates that the child is an *optional* feature. Features connected

```
{
    fromFeature: "root",
    toFeature: "RasterVectorProcessing",
    relationType: "1"
}
{
    fromFeature: "RasterVectorProcessing",
    toFeature: "VisualizingData",
    relationType: "0"
}
```

Figure 7.1. Parent and Child Features' Documents

together via the *requires* or *excludes* relationships are represented in similar documents, with the `relationType` set to `4` or `5`. Figure 7.2 shows a portion of the feature model for the Raster/Vector Image Manipulation SPL as displayed by the MongoDB Compass GUI

## 7.3 Loading and Emptying Feature Models

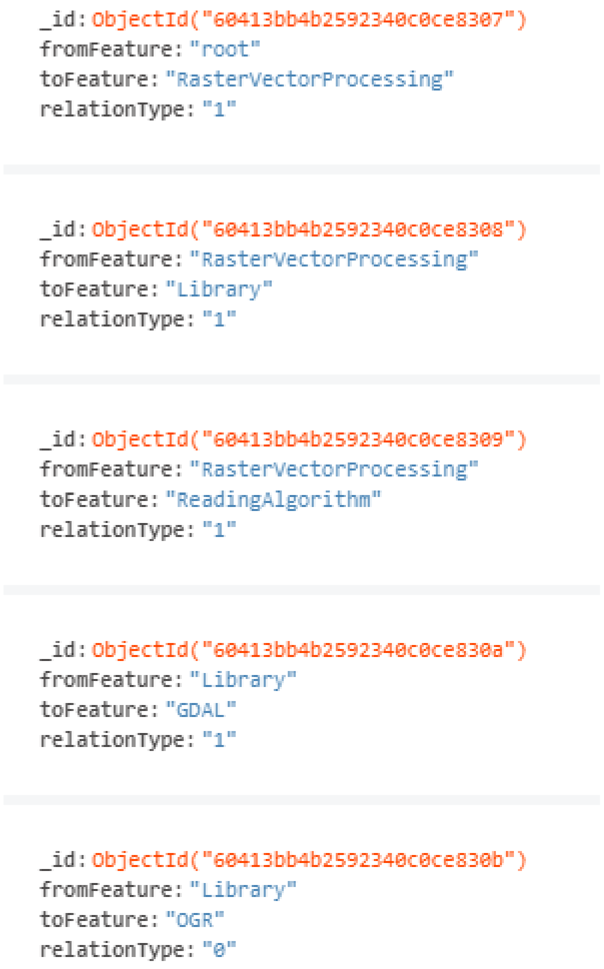
As noted in Section 7.2, a MongoDB collection that encodes a feature model is equivalent to the corresponding RDB and CSV encodings, as they are described in Chapter 3. Thus, we can conveniently import a feature model into MongoDB using the corresponding CSV file. It can be loaded using the following MQL query:

```
mongoimport -d dbName -c collectionName
--type CSV --file fileName --headerline
```

This query is a MongoDB shell command that loads a CSV file into a collection `collectionName`, which resides in the database `dbName`. The command `-headerline` ensures that the first line of the CSV file, which contains the headings (field names) is not loaded as a feature document.

We use the above import operation to *load* feature models in the experiments in Chapter 9. In the experiments, we also use an operation to *empty* a MongoDB collection. That operation can be done with the following query using the MongoDB Compass GUI:

```
dbName.collectionName.remove( { } )
```



```
_id: ObjectId("60413bb4b2592340c0ce8307")
fromFeature: "root"
toFeature: "RasterVectorProcessing"
relationType: "1"
```

```
_id: ObjectId("60413bb4b2592340c0ce8308")
fromFeature: "RasterVectorProcessing"
toFeature: "Library"
relationType: "1"
```

```
_id: ObjectId("60413bb4b2592340c0ce8309")
fromFeature: "RasterVectorProcessing"
toFeature: "ReadingAlgorithm"
relationType: "1"
```

```
_id: ObjectId("60413bb4b2592340c0ce830a")
fromFeature: "Library"
toFeature: "GDAL"
relationType: "1"
```

```
_id: ObjectId("60413bb4b2592340c0ce830b")
fromFeature: "Library"
toFeature: "OGR"
relationType: "0"
```

Figure 7.2. Features for the Raster/Vector Image Manipulation SPL Stored in MongoDB

This query drops the collection from the database. If this command runs on MongoDB shell, then the database that contains the collection must be specified with the command `use dbName`. This allows the query to be executed using *db.collectionName.remove()*.

In the following sections, we show how we manipulate feature models using MongoDB and its query language MQL.

## 7.4 Creating Features in MongoDB

In this section, we develop algorithms for creating, modifying, and deleting features in MongoDB similar to the algorithms defined in Section 4.3. These MongoDB algorithms have the same functionality as the corresponding RDB algorithms except that they access the MongoDB database using MQL queries instead of the MySQL database using SQL queries. In this section, we focus on the algorithm to create a new feature.

To add a new feature to a feature model in MongoDB, we need the new feature's name, its parent's name, its relationship type with its parent, its requires list, and its excludes list. These values must be passed as arguments `fName`, `fParent`, `relationType`, `requires`, and `excludes`, respectively, to the `createFeature`) algorithm shown in Figure 7.3. This algorithm specifies how to insert a new feature into the MongoDB-based feature model encoding.

The `createFeature` algorithm uses three queries that are done using MQL queries.

1. Checking whether a feature with name `fName` does not already exist.

   We construct the needed MQL query by substituting the feature name for `fName` in the following template:

   ```
   {$or: [
       { fromFeature : fName },
       { toFeature: fName}
   ]}
   ```

```
Procedure createFeature(fName, fParent, relationType, requires, excludes)
    Ensure that fName doesn't exist in the feature model
    Ensure that fParent exists in the feature model
    Ensure that relationType matches the parent and its children
    Ensure that requires arg not empty
    arr1 = fetch all features that have mandatory relationship
    arr2 = fetch all ancestors of parent arg
    arr3 = concatenate arr1 and arr2 and add fName
    let arr4, reqArr = []
    foreach item in requires:
        if item not in arr3
            Append item to reqArr
    Ensure that excludes arg not empty
    arr4 = concatenate arr3 and reqArr
    foreach item in excludes:
        if item not in arr4
            push item to excArr
    insert to the database the document:
        {"fromFeature": fParent, "toFeature": fName,
         "relationType": relationType}
    foreach item in reqArr:
        insert to the database the document:
            {"fromFeature": fName, "toFeature": item,
             "relationType": "4"}
    foreach item in excArr:
        insert to the database the document:
            {"fromFeature": fName, "toFeature": item,
             "relationType": "5"}
```

Figure 7.3. Create Feature in MongoDB

This MQL query checks whether the feature with the name `fName` exists either as a parent (`fromFeature`) or as a child (`toFeature`). It uses the MongoDB `find()` method to check whether there is a document that contains that property defined in the template. If the query returns a nonzero value, then the algorithm exits with the error "feature name already exists".

2. Checking whether the `parent` and `relationtype` arguments are correct.

   This can be done using the query template defined in the previous item by replacing `fName` with the value of the `fParent` argument. The `relationType` value can then be retrieved from the returned data.

3. Fetching parent's ancestors. Finding ancestors is a recursive algorithm that takes the parent feature (`fParent`) as an argument and returns all the ancestors up to the root document. We construct the needed MQL query by substituting the parent's feature name for `fParent` in the following template:

```
{"$and":[
    {"toFeature": fParent },
    {"relationType": { "$in": [ "0", "1", "2", "3"] }}
]}
```

   This MQL query checks whether there is a document with the parent feature argument stored as its `child` while `relationType` is not equal to requires (4) or excludes (5) since the parent-child relationships in feature models don't include cross-tree constraints. If an ancestor is found, the algorithm stores the ancestor feature along the path and then calls itself recursively with the *toFeature* property from the returned document as the new value for `fParent` until generating an array of all ancestors from the parent argument up till root.

## 7.5 Generating the Product Configuration Form in MongoDB

The product configuration algorithm follows steps similar to the one shown in Figure 5.4. The algorithm starts by fetching the top-level document from the feature model by using the MQL query

```
{"fromFeature": "root"}
```

and passing the result to the display feature algorithm. The `featuredisplay` algorithm traverses the feature model in a depth-first fashion. It uses the same query to fetch the children of the current document.

## 7.6 Evaluation and Conclusion

This chapter addresses Research Question 5 from Section 1.3: ***Can a document-oriented NoSQL database be used to accurately encode feature models?*** In particular, this chapter uses the document-oriented database system MongoDB.

To answer this question, we first designed a document-oriented MongoDB database to store an arbitrary "traditional" feature model. This design stores the model in a collection of MongoDB documents. Each document within the collection consists of the three properties `fromFeature`, `toFeature`, and `relationType`. It defines "parent" feature `fromFeature` and "child" feature `toFeature` to have the relationship `relationType`. Within the collection, any two features have at most one such relationship defined.

Each document in the collection specifies the same kind of unique relationship between two features that a row in the RDB encoding's **featuesRelations** table and a line in the CSV encoding do (as defined in Chapter 3). Thus, if a MongoDB collection, a **featuresRelations** table, and an CSV file all specify the same set of feature-to-feature relationships, then all encode the same feature model. Moreover, each of these feature-to-feature relationships corresponds to a directed edge in the conceptual feature diagram. Thus, the encoding of the feature model in MongoDB is equivalent to the conceptual feature model.

Given the defined equivalence between the CSV encoding and the MongoDB encoding, we designed an operation to load a feature model into a MongoDB database from a CSV file. This operation produces a MongoDB collection that encodes the feature model defined in the CSV file. We also designed an operation to empty a MongoDB collection, thus removing any feature model it encodes. Both of these operations are used in the experiments in Chapter 9.

We also designed algorithms for creating new features and modifying and deleting existing features in MongoDB-encoded feature models. These algorithms have the same functionality as the corresponding algorithms defined in Chapter 4 for the RDB encoding, except that the MongoDB database is accessed using the queries defined in Section 7.4. If we used these algorithms to implement the Web forms defined in Chapter 4, then Web forms must have the same properties and behaviors. Thus, the correctness arguments for the MongoDB algorithms are essentially the same as the one given in Section 4.3 for the corresponding algorithm from Chapter 4.

In addition, we designed an algorithm that traverses the MongoDB-encoded feature model, determines the relationships and constraints between features, and generates a dynamic Web form. The algorithm has the same functionality as the one given in Figure 5.2 for the RDB encoding, except that the MongoDB database is accessed using the query defined in Section 7.5. This form enables a user to configure valid products from the SPL. This Web form must have the same properties and behaviors as the one given in Chapter 5. Thus, the correctness argument for the MongoDB algorithm is the same as the one given in Section 5.3 for the corresponding algorithm from Chapter 5.

Second, we implemented the MongoDB database design, the operation to load a database from the CSV file, the operation to empty a database, the algorithms for creating, modifying, and deleting features, and the algorithm to generate a product configuration form. For most of these, we developed Python 3.8 programs that access the MongoDB database using the `pymongo` driver. For others (such as the empty operation), we used both Python and the MongoDB Compass GUI.

Third, we tested the above implementations as a part of the experiments conducted in Chapter 9. These experiments loaded ten different feature models into the MongoDB database. For each stored feature model, the experiment created a new feature, generated the product configuration form, and then emptied the database. We ensured that each operation behaved as required.

In this chapter, we have thus demonstrated that the answer to Research Question 5 is "Yes". We have designed an approach that can encode an arbitrary "traditional" feature model accurately in a document-oriented MongoDB database in a manner that is equivalent to the RDB and CSV encodings defined in Chapter 3. We have also designed and implemented operations to load a feature model into a database; empty a database; create, modify, and delete features in an encoded feature model; and generate a product configuration form from the encoded model. Furthermore, we have shown that the approach is practical by using the implementations in the experiments in Chapter 9.

# CHAPTER 8

# ENCODING FEATURE MODELS IN GRAPH DATABASES

This chapter addresses specific Research Question 6 from Section 1.3: ***Can a graph-oriented NoSQL database be used to accurately encode feature models?***

To answer this question, we explore a novel approach that encodes valid feature models for storage in graph databases and preserves the model's validity while creating, modifying, deleting, and extracting information about features. We use the graph database system Neo4j [20, 94, 129].

## 8.1 Graph Databases in a Nutshell

A *graph* is a collection of *nodes* (or vertices) and *edges* connecting nodes. In computer science, graphs are abstract data types used to represent certain data structures such as hierarchical data [129]. A graph data structure uses nodes to store data entities and edges to store relationships between entities. An edge has a start node, an end node, a type, and a direction. As shown in Figure 8.1, graphs can be *directed*, where all edges have an associated a direction, or *undirected*, where no edge has a direction.

Graphs can be cyclic or acyclic. *Cyclic graphs* are directed graphs that contain at least one graph cycle, which is a path from at least one node back to itself. *Acyclic graphs* are directed graphs that contain no graph cycles.

In Chapters 3 and 7, we propose designs to represent the hierarchical data of feature models in relational and document-oriented databases, respectively. Although these approaches can precisely describe feature models, querying these databases (i.e., traversing a tree) can be a time-consuming process [128], especially when manipulating large feature models with thousands of features.

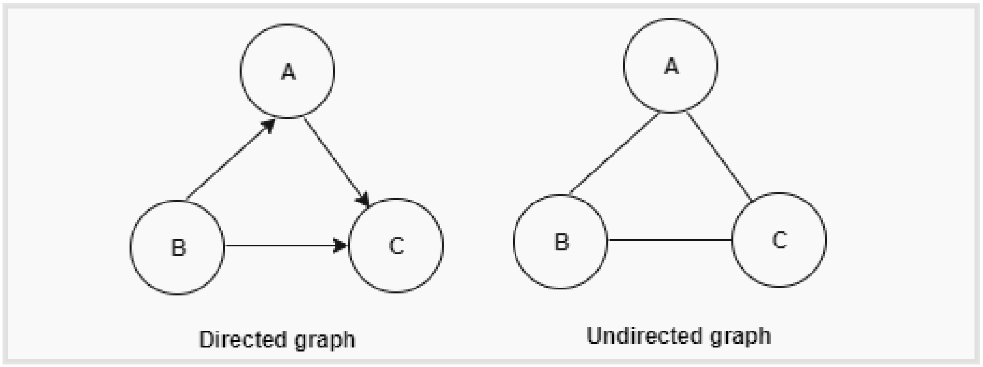


Figure 8.1. Directed and Undirected Graphs

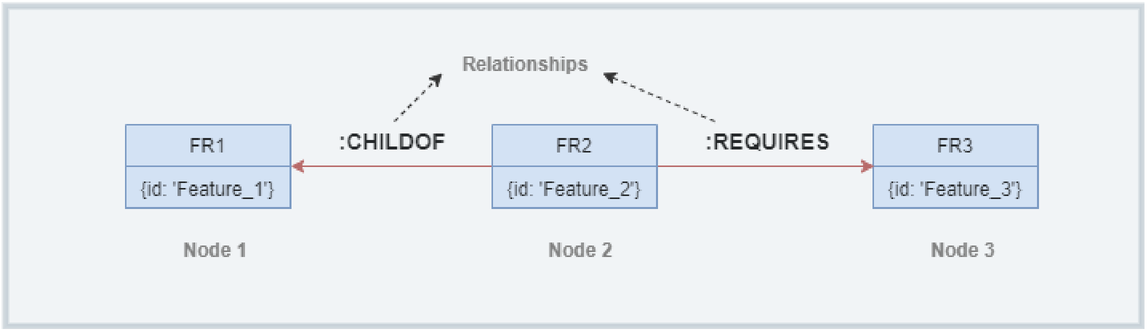


Figure 8.2. Neo4j Property Graph Concept

A *graph database* uses a graph structure to represent and store data. It defines nodes to store data entities and edges to store relationships between entities. A prominent example is the Network Exploration and Optimization 4 Java (Neo4j) [20, 129]—an open-source, NoSQL, graph database system created by Neo4j, Incorporated [88]. In this chapter, we investigate the use of the graph database Neo4j to store and manipulate feature models.

In Neo4j, data are organized in a *property graph*. The graph shown in Figure 8.2 has the three Neo4j nodes `FR1`, `FR2`, and `FR3` representing the data entities of the graph with the two Neo4j relationships `CHILDOF` and `REQUIRES` between them. A Neo4j relationship connects two nodes and always has a direction. For each Neo4j node and relationship, we can attach Neo4j properties to give more information about data entities and their relationships. In Figure 8.2, each node just has one property: `id`.

Neo4j uses *Cypher* [87], a rich query language to fetch data from the database. In the following section, we illustrate how to use Cypher queries to encode feature models in Neo4j databases.

## 8.2 Encoding Feature Models in Neo4j

As described in Chapter 3, a feature model is a *directed acyclic graph* (DAG) (i.e., a graph with no cycles) with labelled edges. A node represents a feature and thus has a name that is unique within the model. A directed edge represents the relationship between the features at its start and end nodes. It is labelled with the type of relationship that exists between the two features. The relationship between features may be one of the parent-child relationships (*mandatory*, *optional*, *OR*, and *alternative*) or one of the cross-tree constraints (*requires* and *excludes*). The feature model's DAG has exactly one node, called the *root*, that has no incoming edges. All other nodes have exactly one incoming edge labeled with a parent relationship but may have any number of (including zero) incoming edges labeled with cross-tree constraints.

The Neo4j database system is designed to store and manipulate graphs. Thus, encoding a feature model in Neo4j is a relatively straightforward process. It can encode feature models more directly than can MySQL and MongoDB, in which special data model designs are needed to store and interpret the hierarchical data.

Consider the *RasterVectorProcessing* feature model shown in Figure 6.5. To store this feature model in Neo4j, we design a Neo4j graph structure in which the nodes, relationships, and properties represent the model's syntax and semantics. We add a node to the Neo4j database for a feature name if and only if the same feature name appears in the feature model. Similarly, we add a relationship to thr Neo4j database if and only if there is a directed edge of the same type and direction between the same two features in the feature model.

As we did with the MongoDB encoding in Chapter 7, we use the CSV encoding (and, hence, the RDB encoding's **featureRelations** table) from Chapter 3 as a guide in designing the Neo4j graph encoding. By doing so, we make it easy to load a feature model into a Neo4j database from a CSV file. A line of the CSV encoding (or row of the **featureRelations** table) records the unique relationship between two features. We identify this relationship with three values:

- `fromFeature`, which is a valid feature name string that denotes the "parent" feature of the relationship (i.e., the start node of an edge in the DAG)
- `toFeature`, which is a valid feature name string that denotes the "child" feature of the relationship (i.e., the end node of an edge in the DAG)
- `relationType`, which is an integer code in the inclusive range 0 to 5 that denotes the relationship between and `fromFeature` and `toFeature`

The meanings of the `relationType` values are the same as given in the **Relationships** table. There is also a Neo4j relationship linking node `fromFeature` to node `toFeature` whose `rType` property is set to the `relationType` value.

In the Cypher query language, there are two ways of creating nodes and relationships:

- The `CREATE` query creates a distinct new node regardless of whether a previous node with the same name exists.
- The `MERGE` clause first checks whether a node with the exists. If it does not already exist, then it creates it as a distinct new node. If it does already exist, then it creates a distinct new node as with `CREATE`.

As we see in next section, the `MERGE` clause is the key to encoding a feature model accurately.

Suppose we have an arbitrary feature model M. By defining the Neo4j encoding as we do above, we can see that M's Neo4j encoding is equivalent to M's CSV and RDB encodings (as defined in Chapter 3). Given that M's CSV and RDB encodings are equivalent to M's MongoDB encoding (as defined in Chapter 7), then M's Neo4j and MongoDB encodings are also equivalent. All of these encodings of M are also equivalent to M's conceptual feature diagram. We exploit these equivalences and use the CSV encoding to load all three databases (MySQL, MongoDB, and Neo4j) for the experiments in Chapter 9.

## 8.3 Loading and Emptying Feature Models

In the operation to load a feature model from a CSV file into Neo4j, we use the `MERGE` clause to prevent the duplication of features in the graph. Building a feature model by loading data from a CSV file with headers `fromFeature`, `toFeature`, and `RelationType` requires a `LOAD` statement with three `MERGE` clauses:

```
LOAD STATEMENT
MERGE (parent:feature { id: line.fromFeature })
MERGE (child:feature { id: line.toFeature })
MERGE (parent)-[:Relation {rType: line.relationType}]->(child)
```

The first two `MERGE` clauses above create a parent and child node, respectively, each of which has an `id` property. The third `MERGE` clause creates the relationship between the parent and child nodes. This relationship's direction points from the parent to the child and has a property `rType`. This property is an integer value in the inclusive range 0 to 5, which are the codes for the feature relationships defined in the **featuresRelations** table from Chapter 3.

Figure 8.3 depicts the *RasterVectorProcessing* feature model as a Neo4j graph after it has been loaded from the CSV file using the `MERGE` clauses above. (The image was created using the Neoj4 Desktop [94], an application that can create and manipulate Neo4j databases locally.) For the *requires* and *excludes* relationships, the Neo4j relationships point toward the feature being required or excluded. For example, if `A` *requires* `B`, then the relationship is `A->B`.

To empty a database in Neo4j, we use the following query:

```
"MATCH (n) DETACH DELETE n"
```

This query deletes all nodes along with their relationships and empties the database.

Note that for performing queries in Neo4j, a targeted database should be started (running) through the Neo4j Desktop/server in order to perform such operations.

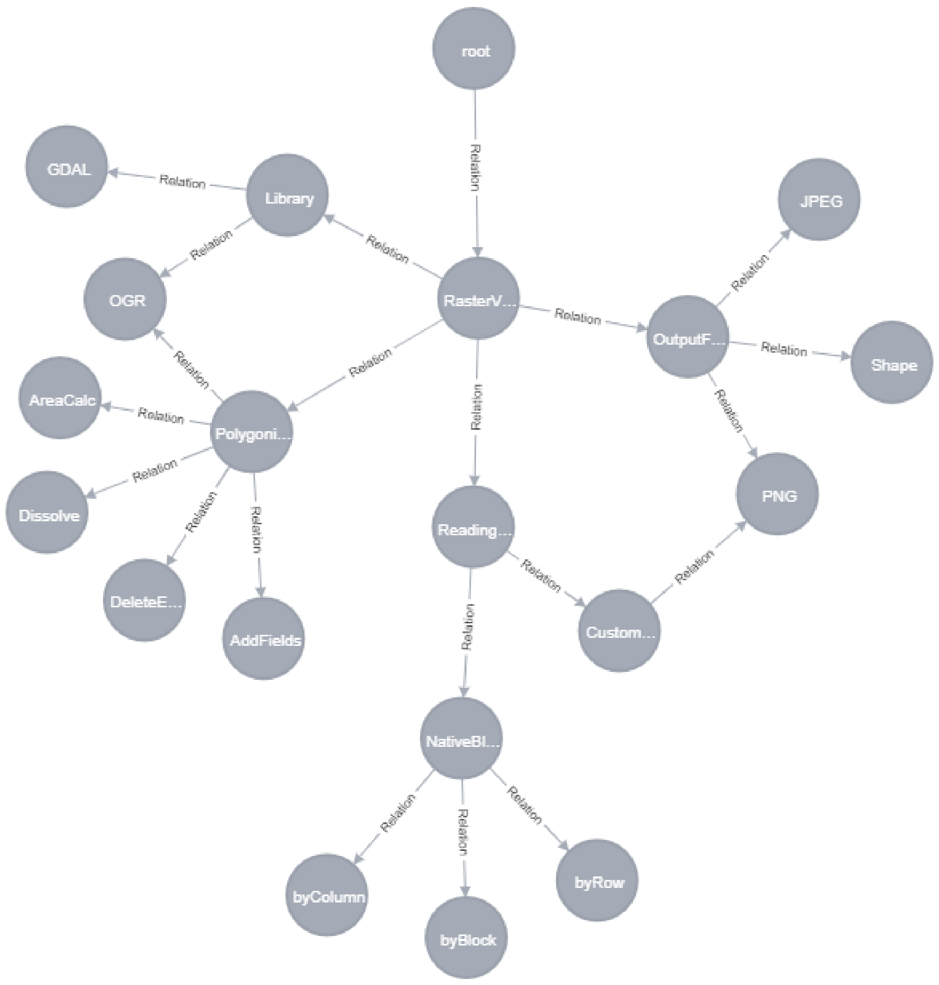


Figure 8.3. Feature Model for Raster/Vector Image Manipulation SPL in in Neo4j

In the following sections, we show how we manipulate feature models with Neo4j using its query language Cypher.

## 8.4 Creating Features in Neo4j

In this section, we develop algorithms for creating, modifying, and deleting features in Neo4j similar to the algorithms defined in Section 4.3. These Neo4j algorithms have the same functionality as the corresponding RDB algorithms except that they access the Neo4j database using Cypher queries instead of the MySQL database using SQL queries. In this section, we focus on the algorithm to create a new feature.

The differences between the Neo4j algorithm and the RDB-based algorithm from Chapter 4 are the implementations of the Cypher queries for the checks carried out in the algorithm. These include the following.

- Checking if the new feature name argument exists:

  ```
  "MATCH (n {id: featureName}) return n"
  ```

  The `MATCH` statement is similar to `SELECT` in SQL. It matches all nodes that has an `id` property with the `featureName` as its value. If the query returns a row of data, the algorithm exits with an error.

- Checking if the `parent` argument exists and if the `relationType` argument matches the parent's relationship with its children (if children exist):

  ```
  "MATCH (p:feature {id: parentName})-[r]->
  (feature) return feature.id, r.rType"
  ```

  The query above does three things. First, the `MATCH` statement checks if the `parent` argument exists (checked with the `id` property). Second, it searches for children (if they exist) and returns a list of rows that contains the child features (through `feature.id`) along with their relationships' types with the parent (through `r.rType`). If the parent exists and the relationship argument matches one of the relationships returned, then the check is passed.

- Finding all mandatory features to be excluded from the *requires* and *excludes* constraints:

  ```
  "MATCH (p)-[:Relation {rType: '1'}]->(c:feature) return c"
  ```

  This query uses the relationship property to search for mandatory features. This can be identified by the `rType` property that has a value of `'1'`, which indicates a mandatory feature.

- Finding parent's ancestors up until the root to be excluded from *requires* and *excludes* constraints:

  ```
  "MATCH p=()-[:Relation*]->(c:feature {id: parentName})
  return p"
  ```

  This query traverses the graph for any given node and returns a list of ancestors of the node up until the root. The (`*`) next to the `RELATION` statement is a variable-length pattern matching [90], which can describe the relationships and the intermediate nodes by specifying a length in the relationship description of a pattern [90]. In this query, the length is (`*`) with bounds, which returns a full path.

  This query illustrates one advantage of Neo4j over MySQL and MongoDB databases. To collect the ancestors of a given feature in a feature model encoded in MySQL or MongoDB requires recursive algorithms and `SELECT` statements. In Neo4j, this traversal can be expressed in one query.

## 8.5 Generating the Product Configuration Form in Neo4j

To generate a product configuration form from a feature model encoded in Neo4j, we develop an algorithm with similar steps to the algorithm shown in Figure 5.4.

The differences are in fetching the top node and passing it to the display function and in fetching the children for each passed feature to the function recursively in a depth-first search manner.

The first query is fetch the top node:

```
"MATCH (p {id: 'root'})-[:Relation]->
(c:feature) return c"
```

This query performs a `MATCH` statement to find the child of the `root` conceptual node. It returns the top feature *RasterVectorProcessing*, which represents the software product line concept.

The second query is to fetch the children for a feature:

```
"MATCH (p:feature {id: featureName})-[r]->
(feature) return feature.id, r.rType"
```

This query performs a `MATCH` query on a feature and returns a list of rows containing children along with their relationship `type` with the parent feature.

## 8.6 Evaluation and Conclusion

This chapter addresses Research Question 6 from Section 1.3: ***Can a graph-oriented NoSQL database be used to accurately encode feature models?*** In particular, this chapter uses the graph database system Neo4j.

To answer this question, we first designed a graph-oriented Neo4j database to store an arbitrary "traditional" feature model. This design stores the feature model's directed acyclic graph straightforwardly as a Neo4j graph. We add a node to the Neo4j database for a feature name if and only if the same feature name appears in the feature model. We attach the feature name to the node as the value of its `id` property. Similarly, we add a relationship to the Neo4j database if and only if there is a directed edge of the same type and direction between the same two features in the feature model. We attach the type to the relationship's `rType` property. Thus, the encoding of the feature model in Neo4j is equivalent to the conceptual feature model.

Each relationship in the Neo4j graph also specifies the same kind of unique relationship between two features that a row in the RDB encoding's **featuesRelations** table and a line in the CSV encoding do (as defined in Chapter 3). Thus, if a Neo4j graph, a **featuresRelations** table, and an CSV file all specify the same set of feature-to-feature relationships, then all encode the same feature model.

Given the defined equivalence relation between the CSV encoding and the Neo4j encoding, we designed an operation to load a feature model into a Neo4j database from a CSV file. This operation produces a Neo4j graph that encodes the feature model defined in the CSV file. We also designed an operation to empty a Neo4j database, thus removing any feature model it encodes. Both of these operations are used in the experiments in Chapter 9.

We also designed algorithms for creating new features and modifying and deleting existing features in Neo4j-encoded feature models. These algorithms have the same functionality as the corresponding algorithms defined in Chapter 4 for the RDB encoding, except that the Neo4j database is accessed using the queries defined in Section 8.4. Thus, the correctness arguments for the Neo4j algorithms are essentially the same as the one given in Section 4.3 for the corresponding algorithm from Chapter 4.

In addition, we designed an algorithm (similar to the one in Figure 5.4) that traverses the Neo4j-encoded feature model, determines the relationships and constraints between features, and generates a dynamic Web form like the one given in Figure 5.2. This form enables a user to configure valid products from the SPL. This Web form must have the same properties and behaviors as the one given in Chapter 5. Thus, the correctness argument for the Neo4j algorithm is the same as the one given in Section 5.3 for the corresponding algorithm from Chapter 5.

Second, we implemented the Neo4j database design, the operation to load a database from the CSV file, the operation to empty a database, the algorithms for creating, modifying, and deleting features, and the algorithm to generate a product configuration form. For most of these, we developed Python 3.8 programs that access the Neo4j database using the `neo4j` driver. For others (such as the empty operation), we used both Python and the Neo4j Desktop GUI.

Third, we tested the above implementations as a part of the experiments conducted in Chapter 9. These experiments loaded ten different feature models into the Neo4j database. For each stored feature model, the experiment created a new feature, generated the product configuration form, and then emptied the database. We ensured that each operation behaved as required.

In this chapter, we have thus demonstrated that the answer to Research Question 6 is "Yes". We have designed an approach that can encode an arbitrary "traditional" feature model accurately in a graph-oriented Neo4j database in a manner that is equivalent to the RDB and CSV encodings defined in Chapter 3. We have also designed and implemented operations to load a feature model into a database; empty a database; create, modify, and delete features in an encoded feature model; and generate a product configuration form from the encoded model. Furthermore, we have shown that the approach is practical by using the implementations in the experiments in Chapter 9.

# CHAPTER 9

# COMPARING DATABASE FEATURE MODEL ENCODINGS

This chapter addresses specific Research Question 7 from Section 1.3: ***Which database system is the best for encoding feature models?***

To answer this question, we evaluate the three database encodings for feature models defined in the previous chapters against sets of objective and subjective criteria. In particular, we consider:

- the relational database (MySQL) encoding from Chapter 3
- the document-oriented database (MongoDB) encoding from Chapter 7
- the graph database (Neo4j) encoding from Chapter 8

We select the criteria carefully to help us determine which encodings are "best" from various perspectives.

- For the objective evaluation, we define, conduct, and analyze the results from a set of experiments to determine how well each database encoding performs selected operations as the feature models increase in size. We give the details beginning in the next section.
- For the subjective evaluation, we identify several issues of interest to software developers, evaluate how well each database system handles each issue, and then analyze the results to determine how suitable each system is for the development of feature-modelings applications. We give the details in Section 9.4.

To unify the results of the evaluations and answer the research question, we consider a typical usage scenario for a feature-modeling application in Section 9.5.

## 9.1 Setting Up the Experiments

To evaluate the three database encodings objectively, we define ten different feature models of varying sizes and heights, five different performance tests, and a procedure for conducting the experiments in this section. In Section 9.2, we present the data we collected from conducting the experiments.

### 9.1.1 Feature Models

To test the performance of the database encodings, we define ten feature models. The smallest is the *RasterVectorProcessing* feature model defined in Chapter 6 (shown in Figure 6.5). We also randomly generate nine feature models ranging in size from 500 features to 24,000 features. We select this range based on feature model sizes found in the literature, where feature models vary in size. One example of a large feature model is the Linux kernel variability model, which contains 5426 features [119]. Other large real-world examples include feature models with more than 18,000 features from the automotive industry [64].

Table 9.1 shows the height and number of features, requires, and excludes for the feature models we use in our experiments.

| **Model#** | **#Features** | **Height** | **#Requires** | **#Excludes** |
|---|---|---|---|---|
| 1 | 19 | 3 | 1 | 1 |
| 2 | 500 | 8 | 15 | 8 |
| 3 | 1000 | 6 | 45 | 14 |
| 4 | 2000 | 5 | 0 | 0 |
| 5 | 6500 | 20 | 341 | 377 |
| 6 | 10000 | 8 | 250 | 278 |
| 7 | 13000 | 15 | 370 | 302 |
| 8 | 15500 | 27 | 412 | 305 |
| 9 | 18000 | 45 | 742 | 617 |
| 10 | 24000 | 61 | 1223 | 1167 |

Table 9.1. Feature Models Used in the Experiments

To generate the random feature models, we use a PHP (version 8.0) program running on an Apache Web server (Win64 version 2.4.46). The program encodes the generated models in

CSV files as described in Section 3.3. We also encode the `RasterVectorProcessing` model in a CSV file.

The experimental procedure loads each feature model from its CSV file into each of the database encodings for the performance tests to be run. The feature model encodings and load operations are described in Chapter 3 for MySQL, Chapter 7 for MongoDB, and Chapter 8 for Neo4j.

### 9.1.2 Performance Tests

To test the performance of each database encoding, we also define five performance tests. These include four tests to determine the time required to:

- *Load* a feature model into the database from a CSV file
- *Create* a feature and add it to the feature model stored in the database after performing semantics checks
- *Generate* a product configuration form by traversing the complete feature model
- *Empty* the database

We also define a fifth performance test to determine the *Size* of the database (i.e., the amount of storage space required).

We choose the five performance tests to be representative of the workloads that occur in practice. The *load* and *empty* tests exercise the database operations used in storing a feature model. The *size* test records the space needed to store a feature model in the database. The *create* test exercises the database operations used in checking the validity of a new feature and inserting it into a stored feature model. The *generate* test exercises the database operations used in traversing every feature in the feature model to generate a product configuration Web form. Product configuration is a key aspect of the feature modeling research reported in this dissertation and an important functionality provided by feature modeling tools.

We do not elaborate on the *create* and *generate* tests here because the corresponding feature creation and product configuration operations are discussed sufficiently in the database-specific chapters: relational databases (MySQL) in Chapters 3, 4, and 5; document-oriented databases (MongoDB) in Chapter 7; and graph databases (Neo4j) in Chapter 8. We implement the feature creation and product configuration algorithms in Python 3.8 because:

- MySQL supports Python with the `mysql.connector` driver [7]
- MongoDB supports Python with the `pymongo` driver [77]
- Neo4j supports Python with the `neo4j` driver [89]

We implement the database *load* and *empty* operations using queries specific to the database system. The MySQL, MongoDB, and Neo4j operations are described in Chapters 3, 7, and 8, respectively.

### 9.1.3 Experimental Procedure

The experimental procedure consists of nested loops that iterate over the database encodings, feature models, and performance tests. The procedure repeats each performance test several times and then computes the average and other statistics for the measurements collected for each test. We define the following parameters for the experimental procedure.

- `DB` = the collection of database encodings
- `FM` = the collection of feature models
- `PT` = the collection of time-based feature model performance tests
- `N` = the number of repetitions of the test runs

The following pseudocode, expressed in terms of the above parameters, outlines the experimental procedure:

```
PROCEDURE(DB,FM,PT,N):
For each db in DB:
    For each fm in FM:
        Repeat N times:
            For each pt in PT:
                Perform pt, measuring time to complete
        Compute and record statistics for this run
          including mean, minimum, maximum for each pt
```

As we did for the performance tests, we implement the experimental procedure in Python 3.8 because it is well supported by all three database systems. We measure the time by importing the Python `time` module and using its `time()` function. This function returns the time as a floating point number expressed in seconds since the beginning of the epoch: January 1, 1970, 00:00:00 Coordinated Universal Time (UTC) [110]. For instance, to calculate the elapsed time for a code fragment, we call the `time()` function before and after the fragment, record the two values, and then subtract the first value from the second.

In the procedure description above, the collection `PT` does not include the *size* performance test. It is performed separately because that information cannot be determined programmatically for all three database systems.

- For MySQL, we get the size of the database in megabytes (MB) using the SQL query

  ```
  SELECT table\_schema dbName,
  sum( data_length + index_length )/1024/1024
  'db size in MB' FROM information_schema.TABLES
  GROUP BY table_schema
  ```

  This query can be executed using the MySQL shell, the GUI, or a Python program via the `mysql.connector`.

- For MongoDB, we get the size of the MongoDB collection in bytes by using the MongoDB shell (mongo) function `db.CollectionName.stats().storageSize`.

- For Neo4j, we calculate the size of the database using other information.

The Neo4j query language does not provide a query to determine the size of the database, although the Neo4j Desktop (the database GUI) can determine the size. The recommended method for calculating the size of a Neo4j graph database is to (1) count all nodes, relations, and properties, (2) multiply each count by the size of that entity in bytes, and (3) sum the results [113]. A node takes 15 bytes, a relationship takes 34 bytes, and a property 41 bytes. So, instead of using a Python program, we retrieve the graph information for each feature model using the GUI and calculate the size using an Excel spreadsheet.

## 9.2 Collecting the Experimental Results

As described in Section 9.1, the experiments involve three types of database encodings, ten different feature models, and the performance tests *load*, *empty*, *create*, and *generate*. For each database encoding, the experimental procedure iterates over the feature models and then performs the collection of performance tests ten times. For each test, the procedure measures the time it takes. After all repetitions, the procedure computes the minimum, maximum, and mean statistics for the collection of measured times.

We thus collect the results of these time-based performance tests in 30 tables. Each of the tables records the statistics for the *load*, *empty*, *create*, and *generate* performance tests on one pairing of a database encoding with a feature model. We examine these tables in the next three subsections. We present our analyses of the results, including plots of the data, in Section 9.3.

We perform the *size* performance tests for the database encodings separately from the time-based tests. The results of these tests are given in Section 9.2.4. Table 9.32 records the disk space needed to store each feature model for each of the three database encodings.

We conducted the experiments in the following execution environment:

- Intel® Core$^{TM}$ i7-6650U processor with 4 megabytes (MB) of cache and 16 gigabytes (GB) of RAM

- Samsung MZFLW256HEHP-000MV solid state drive (SSD) 256GB with a sequential write rate of 314.9 megabytes per second (MB/sec), sequential read rate of 723.6 MB/sec, and sequential mixed rate of 303.9 MB/sec.
- Windows 10 Pro operating system version 2004
- MariaDB (a fork of MySQL) version 10.4.17 (in the XAMPP 8.0.0 development environment)
- MongoDB version 4.4.2 (including Compass)
- Neo4j version 4.1.3 (including Desktop)
- Python 3.8 with the drivers `mysql.connector` [100] for MySQL, `pymongo` [48] for MongoDB, and `neo4j` [89] for Neo4j

### 9.2.1 MySQL Results

For the MySQL-based feature model encoding, the experimental procedure ran the performance tests *load*, *empty*, *create*, and *generate* on each of the ten feature models ten times and measured the time needed to complete the test. It then computed the minimum, maximum, and mean values of the times. The statistics for each feature model are given in the following tables:

1. Table 9.2 for the Raster/Vector Image Manipulation SPL defined in Chapter 6, which has 19 features
2. Table 9.3 for the feature model with 500 features
3. Table 9.4 for the feature model with 1000 features
4. Table 9.5 for the feature model with 2000 features
5. Table 9.6 for the feature model with 6500 features

6. Table 9.7 for the feature model with 10,000 features
7. Table 9.8 for the feature model with 13,000 features
8. Table 9.9 for the feature model with 15,500 features
9. Table 9.10 for the feature model with 18,000 features
10. Table 9.11 for the feature model with 24,000 features

Analyses and plots of the data for each performance test are given in Section 9.3.

| Test | Minimum | Maximum | Mean |
|---|---|---|---|
| ***Load*** | 0.001994 | 0.012335 | 0.006917 |
| ***Empty*** | 0.068674 | 0.137555 | 0.087411 |
| ***Create*** | 0.000996 | 0.016715 | 0.004163 |
| ***Generate*** | 0.004979 | 0.006981 | 0.006170 |

Table 9.2. MySQL Test Times in Seconds for Feature Models with 19 Features

| Test | Minimum | Maximum | Mean |
|---|---|---|---|
| ***Load*** | 0.009301 | 0.018071 | 0.011589 |
| ***Empty*** | 0.063812 | 0.121399 | 0.079190 |
| ***Create*** | 0.016438 | 0.020461 | 0.018759 |
| ***Generate*** | 0.272952 | 0.296449 | 0.288086 |

Table 9.3. MySQL Test Times in Seconds for Feature Models with 500 Features

| Test | Minimum | Maximum | Mean |
|---|---|---|---|
| ***Load*** | 0.016954 | 0.060831 | 0.025431 |
| ***Empty*** | 0.065823 | 0.150595 | 0.088861 |
| ***Create*** | 0.001994 | 0.006979 | 0.003789 |
| ***Generate*** | 0.749706 | 0.844134 | 0.780360 |

Table 9.4. MySQL Test Times in Seconds for Feature Models with 1000 Features

| Test | Minimum | Maximum | Mean |
|---|---|---|---|
| ***Load*** | 0.026928 | 0.030917 | 0.028324 |
| ***Empty*** | 0.064826 | 0.175531 | 0.084973 |
| ***Create*** | 0.002991 | 0.004986 | 0.004529 |
| ***Generate*** | 2.229544 | 2.419636 | 2.280450 |

Table 9.5. MySQL Test Times in Seconds for Feature Models with 2000 Features

| Test | Minimum | Maximum | Mean |
|---|---|---|---|
| ***Load*** | 0.084774 | 0.210049 | 0.125500 |
| ***Empty*** | 0.058842 | 0.097710 | 0.071633 |
| ***Create*** | 0.043455 | 0.062836 | 0.052017 |
| ***Generate*** | 9.454666 | 10.275765 | 9.594424 |

Table 9.6. MySQL Test Times in Seconds for Feature Models with 6500 Features

| Test | Minimum | Maximum | Mean |
|---|---|---|---|
| ***Load*** | 0.130561 | 0.268377 | 0.154142 |
| ***Empty*** | 0.067025 | 0.182787 | 0.090511 |
| ***Create*** | 0.046760 | 0.075539 | 0.053453 |
| ***Generate*** | 29.882542 | 43.839573 | 33.115606 |

Table 9.7. MySQL Test Times in Seconds for Feature Models with 10,000 Features

| Test | Minimum | Maximum | Mean |
|---|---|---|---|
| ***Load*** | 0.189491 | 0.364410 | 0.232332 |
| ***Empty*** | 0.060837 | 0.128081 | 0.082952 |
| ***Create*** | 0.013002 | 0.024932 | 0.017807 |
| ***Generate*** | 85.923167 | 149.575551 | 110.274189 |

Table 9.8. MySQL Test Times in Seconds for Feature Models with 13,000 Features

| Test | Minimum | Maximum | Mean |
|---|---|---|---|
| ***Load*** | 0.193652 | 0.375610 | 0.280743 |
| ***Empty*** | 0.060830 | 0.178522 | 0.089541 |
| ***Create*** | 0.015955 | 0.033907 | 0.024135 |
| ***Generate*** | 113.723715 | 165.636194 | 152.582528 |

Table 9.9. MySQL Test Times in Seconds for Feature Models with 15,500 Features

| Test | Minimum | Maximum | Mean |
|---|---|---|---|
| ***Load*** | 0.196885 | 0.275901 | 0.222032 |
| ***Empty*** | 0.072550 | 0.208823 | 0.096513 |
| ***Create*** | 0.017201 | 0.022192 | 0.018369 |
| ***Generate*** | 142.851873 | 172.387218 | 152.738201 |

Table 9.10. MySQL Test Times in Seconds for Feature Models with 18,000 Features

| **Test** | **Minimum** | **Maximum** | **Mean** |
|---|---|---|---|
| ***Load*** | 0.209529 | 0.379513 | 0.250805 |
| ***Empty*** | 0.067095 | 0.232254 | 0.120001 |
| ***Create*** | 0.021827 | 0.031912 | 0.025261 |
| ***Generate*** | 227.475839 | 260.394062 | 237.876649 |

Table 9.11. MySQL Test Times in Seconds for Feature Models with 24,000 Features

### 9.2.2 MongoDB Results

For the MongoDB-based feature model encoding, the experimental procedure procedure ran the performance tests *load*, *empty*, *create*, and *generate* on each of the ten feature models ten times and measured the time needed to complete the test. It then computed the minimum, maximum, and mean values of the times. The statistics for each feature model are given in the following tables:

1. Table 9.12 for the Raster/Vector Image Manipulation SPL defined in Chapter 6, which has 19 features
2. Table 9.13 for the feature model with 500 features
3. Table 9.14 for the feature model with 1000 features
4. Table 9.15 for the feature model with 2000 features
5. Table 9.16 for the feature model with 6500 features
6. Table 9.17 for the feature model with 10,000 features
7. Table 9.18 for the feature model with 13,000 features
8. Table 9.19 for the feature model with 15,500 features
9. Table 9.20 for the feature model with 18,000 features
10. Table 9.21 for the feature model with 24,000 features

Analyses and plots of the data for each performance test are given in Section 9.3.

| **Test** | **Minimum** | **Maximum** | **Mean** |
|---|---|---|---|
| ***Load*** | 0.025928 | 0.057845 | 0.033585 |
| ***Empty*** | 0.008975 | 0.037898 | 0.014859 |
| ***Create*** | 0.002991 | 0.020943 | 0.007580 |
| ***Generate*** | 0.000995 | 0.000999 | 0.000997 |

Table 9.12. MongoDB Test Times in Seconds for Feature Models with 19 Features

| **Test** | **Minimum** | **Maximum** | **Mean** |
|---|---|---|---|
| ***Load*** | 0.030917 | 0.065200 | 0.042423 |
| ***Empty*** | 0.008975 | 0.014959 | 0.012199 |
| ***Create*** | 0.006982 | 0.012964 | 0.009578 |
| ***Generate*** | 0.309173 | 0.394088 | 0.340311 |

Table 9.13. MongoDB Test Times in Seconds for Feature Models with 500 Features

| **Test** | **Minimum** | **Maximum** | **Mean** |
|---|---|---|---|
| ***Load*** | 0.041278 | 0.055528 | 0.044493 |
| ***Empty*** | 0.008203 | 0.014144 | 0.010425 |
| ***Create*** | 0.009978 | 0.013103 | 0.011318 |
| ***Generate*** | 1.117192 | 1.350086 | 1.230210 |

Table 9.14. MongoDB Test Times in Seconds for Feature Models with 1000 Features

| **Test** | **Minimum** | **Maximum** | **Mean** |
|---|---|---|---|
| ***Load*** | 0.056850 | 0.065632 | 0.059075 |
| ***Empty*** | 0.009248 | 0.019647 | 0.011115 |
| ***Create*** | 0.003766 | 0.006981 | 0.004672 |
| ***Generate*** | 2.724871 | 2.946841 | 2.824609 |

Table 9.15. MongoDB Test Times in Seconds for Feature Models with 2000 Features

| **Test** | **Minimum** | **Maximum** | **Mean** |
|---|---|---|---|
| ***Load*** | 0.133007 | 0.175335 | 0.148040 |
| ***Empty*** | 0.009266 | 0.016313 | 0.011973 |
| ***Create*** | 0.026670 | 0.037415 | 0.029049 |
| ***Generate*** | 12.099744 | 12.778247 | 12.400493 |

Table 9.16. MongoDB Test Times in Seconds for Feature Models with 6500 Features

| **Test** | **Minimum** | **Maximum** | **Mean** |
|---|---|---|---|
| ***Load*** | 0.190235 | 0.245325 | 0.208793 |
| ***Empty*** | 0.009838 | 0.017073 | 0.012191 |
| ***Create*** | 0.040402 | 0.043981 | 0.041824 |
| ***Generate*** | 35.283151 | 48.529585 | 39.553670 |

Table 9.17. MongoDB Test Times in Seconds for Feature Models with 10,000 Features

| Test | Minimum | Maximum | Mean |
|---|---|---|---|
| ***Load*** | 0.265323 | 0.342685 | 0.288418 |
| ***Empty*** | 0.009594 | 0.019411 | 0.014130 |
| ***Create*** | 0.016253 | 0.030916 | 0.019787 |
| ***Generate*** | 91.386352 | 158.888846 | 103.543996 |

Table 9.18. MongoDB Test Times in Seconds for Feature Models with 13,000 Features

| Test | Minimum | Maximum | Mean |
|---|---|---|---|
| ***Load*** | 0.301629 | 0.340774 | 0.316446 |
| ***Empty*** | 0.009513 | 0.021008 | 0.011433 |
| ***Create*** | 0.063110 | 0.080366 | 0.068223 |
| ***Generate*** | 124.309522 | 171.399285 | 129.508539 |

Table 9.19. MongoDB Test Times in Seconds for Feature Models with 15,500 Features

| Test | Minimum | Maximum | Mean |
|---|---|---|---|
| ***Load*** | 0.343081 | 0.409188 | 0.375360 |
| ***Empty*** | 0.009763 | 0.022586 | 0.013922 |
| ***Create*** | 0.019912 | 0.036864 | 0.022385 |
| ***Generate*** | 171.974277 | 220.431957 | 188.721628 |

Table 9.20. MongoDB Test Times in Seconds for Feature Models with 18,000 Features

| Test | Minimum | Maximum | Mean |
|---|---|---|---|
| ***Load*** | 0.426939 | 0.479269 | 0.448579 |
| ***Empty*** | 0.008906 | 0.013278 | 0.010428 |
| ***Create*** | 0.081581 | 0.087570 | 0.084208 |
| ***Generate*** | 261.150933 | 307.459409 | 266.890339 |

Table 9.21. MongoDB Test Times in Seconds for Feature Models with 24,000 Features

### 9.2.3 Neo4j Results

For the Neo4j-based feature model encoding, the experimental procedure procedure ran the performance tests *load*, *empty*, *create*, and *generate* on each of the ten feature models ten times and measured the time needed to complete the test. It then computed the minimum, maximum, and mean values of the times. The statistics for each feature model are given in the following tables:

1. Table 9.22 for the Raster/Vector Image Manipulation SPL defined in Chapter 6, which has 19 features
2. Table 9.23 for the feature model with 500 features
3. Table 9.24 for the feature model with 1000 features
4. Table 9.25 for the feature model with 2000 features
5. Table 9.26 for the feature model with 6500 features
6. Table 9.27 for the feature model with 10,000 features
7. Table 9.28 for the feature model with 13,000 features
8. Table 9.29 for the feature model with 15,500 features
9. Table 9.30 for the feature model with 18,000 features
10. Table 9.31 for the feature model with 24,000 features

Analyses and plots of the data for each performance test are given in Section 9.3.

| Test | Minimum | Maximum | Mean |
|---|---|---|---|
| ***Load*** | 0.029920 | 1.507966 | 0.189593 |
| ***Empty*** | 0.012965 | 0.042885 | 0.020246 |
| ***Create*** | 0.060838 | 0.720073 | 0.137232 |
| ***Generate*** | 0.051862 | 0.169544 | 0.087965 |

Table 9.22. Neo4j Test Times in Seconds for Feature Models with 19 Features

| Test | Minimum | Maximum | Mean |
|---|---|---|---|
| ***Load*** | 0.060532 | 0.093185 | 0.072242 |
| ***Empty*** | 0.012804 | 0.024150 | 0.015956 |
| ***Create*** | 0.008009 | 0.046874 | 0.014309 |
| ***Generate*** | 0.787748 | 0.954446 | 0.872124 |

Table 9.23. Neo4j Test Times in Seconds for Feature Models with 500 Features

| Test | Minimum | Maximum | Mean |
|---|---|---|---|
| ***Load*** | 0.221351 | 0.268822 | 0.231621 |
| ***Empty*** | 0.018796 | 0.024894 | 0.021524 |
| ***Create*** | 0.008863 | 0.020945 | 0.010660 |
| ***Generate*** | 1.916304 | 2.111225 | 1.968274 |

Table 9.24. Neo4j Test Times in Seconds for Feature Models with 1000 Features

| Test | Minimum | Maximum | Mean |
|---|---|---|---|
| ***Load*** | 0.693919 | 0.759823 | 0.725667 |
| ***Empty*** | 0.029720 | 0.036664 | 0.033966 |
| ***Create*** | 0.056267 | 0.196119 | 0.072811 |
| ***Generate*** | 3.670924 | 4.155175 | 3.886160 |

Table 9.25. Neo4j Test Times in Seconds for Feature Models with 2000 Features

| Test | Minimum | Maximum | Mean |
|---|---|---|---|
| ***Load*** | 10.253567 | 11.788021 | 10.919712 |
| ***Empty*** | 0.076301 | 0.097736 | 0.083201 |
| ***Create*** | 0.010961 | 0.037795 | 0.015276 |
| ***Generate*** | 9.923098 | 11.655733 | 10.547466 |

Table 9.26. Neo4j Test Times in Seconds for Feature Models with 6500 Features

| Test | Minimum | Maximum | Mean |
|---|---|---|---|
| ***Load*** | 26.036468 | 28.616515 | 27.324511 |
| ***Empty*** | 0.121134 | 0.183801 | 0.139862 |
| ***Create*** | 0.013711 | 0.036901 | 0.020672 |
| ***Generate*** | 25.613209 | 29.345878 | 27.395161 |

Table 9.27. Neo4j Test Times in Seconds for Feature Models with 10,000 Features

| Test | Minimum | Maximum | Mean |
|---|---|---|---|
| ***Load*** | 52.354977 | 61.173256 | 57.922738 |
| ***Empty*** | 0.228389 | 0.588427 | 0.319346 |
| ***Create*** | 0.134256 | 0.881606 | 0.252533 |
| ***Generate*** | 62.268842 | 76.526353 | 66.337345 |

Table 9.28. Neo4j Test Times in Seconds for Feature Models with 13,000 Features

| Test | Minimum | Maximum | Mean |
|---|---|---|---|
| ***Load*** | 84.520338 | 121.792918 | 99.218970 |
| ***Empty*** | 0.217852 | 0.311166 | 0.264296 |
| ***Create*** | 0.136768 | 0.759222 | 0.274160 |
| ***Generate*** | 83.571676 | 87.650565 | 85.020761 |

Table 9.29. Neo4j Test Times in Seconds for Feature Models with 15,500 Features

| Test | Minimum | Maximum | Mean |
|---|---|---|---|
| ***Load*** | 102.861199 | 118.050459 | 105.977191 |
| ***Empty*** | 0.244262 | 0.346071 | 0.296565 |
| ***Create*** | 0.117670 | 0.269819 | 0.160748 |
| ***Generate*** | 92.617326 | 96.710604 | 94.717800 |

Table 9.30. Neo4j Test Times in Seconds for Feature Models with 18,000 Features

| Test | Minimum | Maximum | Mean |
|---|---|---|---|
| ***Load*** | 167.643698 | 183.148403 | 173.797135 |
| ***Empty*** | 0.329170 | 0.429861 | 0.364161 |
| ***Create*** | 0.015922 | 0.042856 | 0.025865 |
| ***Generate.*** | 156.859509 | 162.810076 | 159.988004 |

Table 9.31. Neo4j Test Times in Seconds for Feature Models with 24,000 Features

### 9.2.4 Database Size Results

The *size* performance test calculates the space required for a feature model. We ran the test on each of the ten feature models and measured the total space required in megabytes. Table 9.32 shows the results after calculating the sizes for the ten feature models for each database encoding.

| **Model#** | **Size** | **MySQL** | **MongoDB** | **Neo4j** |
|---|---|---|---|---|
| 1 | 19 | 0.015625 | 0.019531 | 0.002883 |
| 2 | 500 | 0.046875 | 0.023438 | 0.065370 |
| 3 | 1000 | 0.078125 | 0.035157 | 0.134157 |
| 4 | 2000 | 0.109375 | 0.046875 | 0.250135 |
| 5 | 6500 | 0.296875 | 0.125012 | 0.852258 |
| 6 | 10000 | 0.484375 | 0.195313 | 1.368162 |
| 7 | 13000 | 0.645881 | 0.230468 | 1.711436 |
| 8 | 15500 | 0.801005 | 0.269530 | 2.020330 |
| 9 | 18000 | 0.947225 | 0.324219 | 2.488201 |
| 10 | 24000 | 1.515625 | 0.402344 | 3.083975 |

Table 9.32. Storage in Megabytes for Each Database Encoding and Feature Model

An analysis and plot of the data for this performance test are given in Section 9.3.

## 9.3 Analyzing the Performance Test Results

Section 9.2 presents the results of our experiments. We conducted experiments on three different database encodings (MySQL, MongoDB, and Neo4j) using ten different feature models of varying sizes. For each database encoding and feature model, we performed four different time-based performance tests (*load* the database, *empty* the database, *create* and insert a new feature, and *generate* the product configuration form). We repeated each test ten times and computed the mean time for the test to complete. For each database encoding and feature model, we also performed a space-based performance test to determine the database size.

In this section, we analyze the results of our experiments. We seek to determine how the various database encodings perform on each performance test as the size of the feature model increases. We show five plots, one for each of the performance tests. Each plot shows the ten feature models in increasing size along the x-axis and time (or space) in increasing value along the y-axis.

### 9.3.1 Load Performance Tests

Figure 9.1 shows a plot of the results of conducting the *Load* performance tests for the three database encodings and ten feature models. For each database encoding and feature model, it shows the mean time (in seconds) taken to *load* a feature model that is encoded in a CSV file.

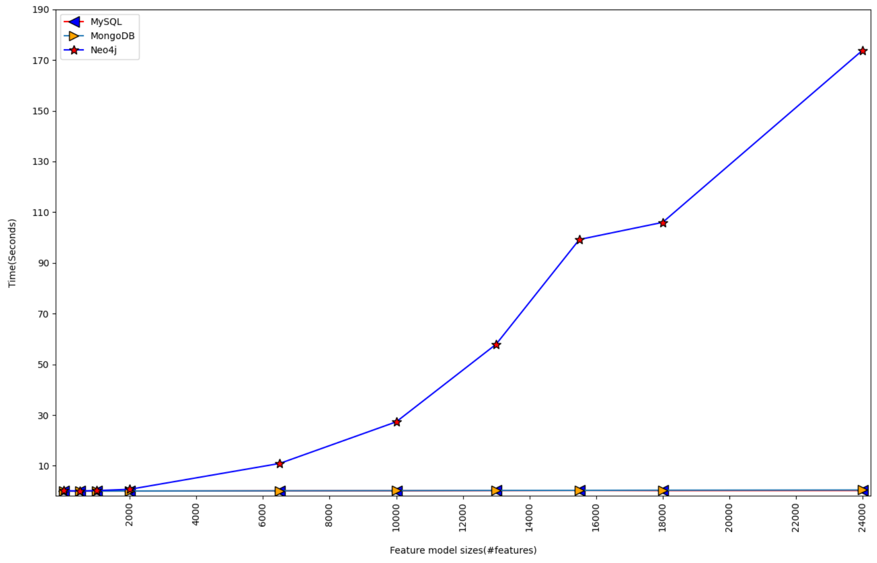


Figure 9.1. Combined Results for the *Load* Performance Tests

From the tables in Section 9.2 and the plot, we observe that all tested feature models *load* into both MySQL and MongoDB in less than 0.5 seconds, with only a slight increase as the feature model size increases. A feature model loads into MySQL slightly faster than the corresponding model loads into MongoDB.

Neo4j performs differently. In tests of feature models up to 2000 features, it loads the models in less than 0.75 seconds, slower than the other two database systems but not prohibitively so. However, the time begins to grow explosively beginning with the feature model of size 6500.

Why does Neo4j perform so poorly as the feature models increase in size? As discussed in Chapter 8, the poor performance seems to result from how we must construct the Neo4j graph database from the set of feature-to-feature (i.e., node-to-node) relationships in the feature model's CSV encoding. If we naively use the fast `CREATE` statement in Neo4j's Cypher query language to create the Neo4j nodes, a feature with M relationships with other features (e.g., with several child or required/excluded features) would be created as M separate nodes in the graph. Instead we must use the `MERGE` clause to create the nodes without duplication.

In Neo4j, query evaluation is usually *lazy.* That is, "most operators pipe their output rows to their parent operators as soon as they are produced" [94]. Thus, "a child operator may not be fully exhausted before the parent operator starts consuming the input rows produced by the child" [94]. The parent and child operators can execute concurrently and the child operators can be stopped as soon as no further output is needed.

However, some query evaluation must be done in an *eager* fashion. If an operator needs "to complete execution in its entirety" [94] before its result can be used by its parent, then it must be executed eagerly. Such operations may result in high memory usage [94] as well as losing the advantages of laziness. Examples of eager operators include sorting and aggregation.

If a `CREATE` statement has a `MERGE` clause, then the `MERGE` must complete before the `CREATE` can be executed. The operation to *load* a Neo4j database from a CSV file involves three MERGE clauses as defined in Section 8.3). So, as the size of the feature model gets large, the *load* operation becomes slow.

If being able to quickly load a feature model from an external CSV file is a significant factor in the choice of feature model storage, then both MySQL and MongoDB seem to be good choices regardless of the size of the feature model. Neo4j is also acceptable for a modest size feature model of a few thousand nodes, but its performance deteriorates for larger feature models. However, we do not expect that the load time for a feature model is typically a significant factor in most practical situations. We expect feature models to be loaded once and then used repeatedly for other operations such as generating product configuration forms. Alternatively, a feature model may be constructed and modified incrementally rather than loaded all at once.

### 9.3.2 Empty Performance Tests

Figure 9.2 shows a plot of the results of conducting the *Empty* performance tests for the three database encodings and ten feature models. For each database encoding and feature model, it shows the mean time (in seconds) taken to execute the test.

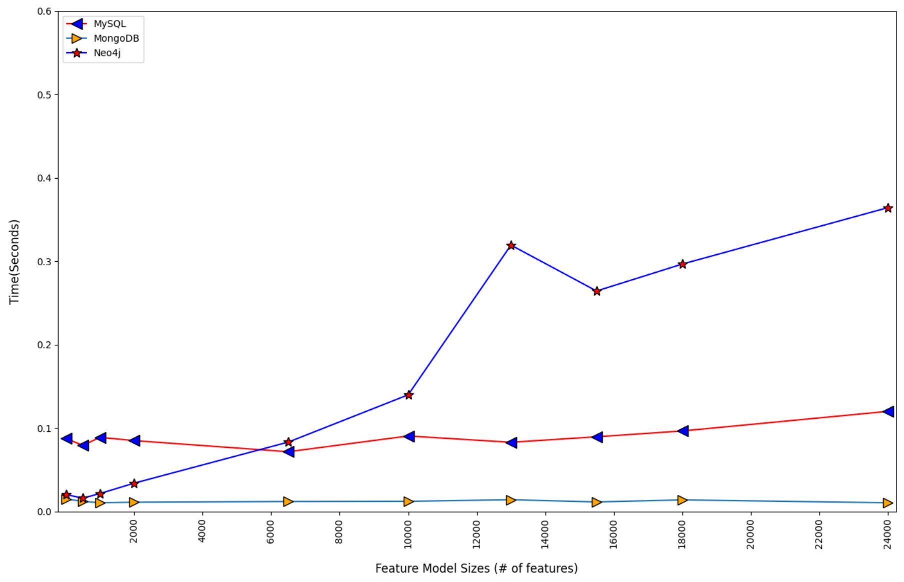


Figure 9.2. Combined Results for the *Empty* Performance Tests

From the tables in Section 9.2 and the plot, we observe that MongoDB required less than 0.02 seconds to empty a database for all the feature models we tested, regardless of size. Emptying a feature model in MySQL was slower, taking approximately 0.1 seconds, with perhaps a slow increase beginning with the feature model of size 13,000.

Again, Neo4j performs differently. In tests of feature models up to 2000 features, it takes more time than MongoDB but less than MySQL. However, with a feature model size of 6500, Neo4j's time exceeds the time for MySQL and continues to grow erratically as the size of the feature model increases. It has an odd spike in the time requirement for the model of size 13,000 before decreasing and beginning to grow again. However, it is important to note that even with the model of size 24,000, the empty time is still less than 0.4 seconds.

Emptying a Neo4j database often exhibits irregular behaviors. For instance, sometimes a larger feature model (e.g., the size 15,500 model in our experiments) can take less time than a smaller one (e.g., the size 13,000 model).

In Neo4j research discussions, the recommended way to empty a database is "to stop the database, delete the graph store (`data/graph.db` (pre v3.x) or `data/databases/graph.db` (3.x forward) or similar) directory, and start the database" [52]. These steps cause Neo4j to build a fresh and empty database.

Another process recommended for emptying databases with 100,000 or more nodes is to use the Awesome Procedures for Neo4j (APOC) library [57], which provides many useful support procedures for Neo4j database querying. We installed APOC as a Neo4j Desktop plugin and edited the Neo4j configuration file to cause it to be activated and loaded. We loaded and deleted a database using APOC. However, this procedure did not yield a significant difference in the time because our largest feature models are considerably smaller than 100,000 nodes.

If being able to quickly empty a database is a significant factor in the choice of feature model storage, then both MongoDB and MySQL seem to be good choices regardless of the size of the feature model. Neo4j behaves more erratically, but even with a feature model of 24,000 features, its performance is likely not prohibitive. However, we do not expect that the empty time for a feature model is typically a significant factor in most practical situations.

### 9.3.3 Create Performance Tests

Figure 9.3 shows a plot of the results of conducting the *Create* performance tests for the three database encodings and ten feature models. For each database encoding and feature model, it shows the mean time (in seconds) taken to execute the test.

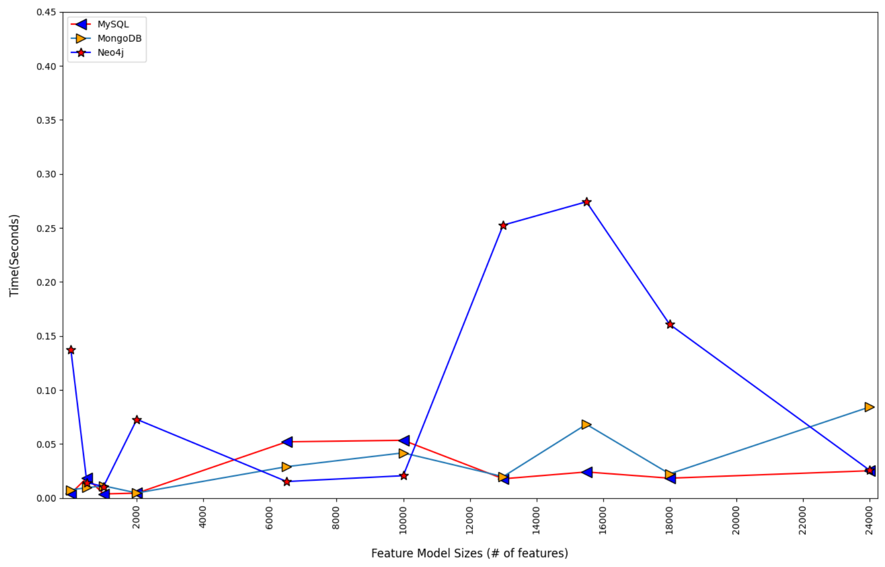


Figure 9.3. Combined Results for the *Create* Performance Tests

From the tables in Section 9.2 and the plot, we observe that the time to *create* a feature and add it to the feature model stored in a database is generally smaller in MySQL and MongoDB than in Neo4j. The times increase slightly as the feature models get larger. All three database systems have somewhat erratic behavior, but Neo4j seems much more erratic than the others. However, all of the create times are quite small regardless of feature model size—no more than 0.3 seconds in the worst cases.

As is evident from a few discussions archived on the Neo4j community websites and `Stackoverflow.com`, the minor timing differences observed for Neo4j are expected for requests to the Neo4j server from the Neo4j Desktop. Neo4j uses caching and thus the first few queries have irregular times because of the need "to warm up the cache" [123]. The Cypher compiler caches the execution plan for the Cypher queries (i.e., caches queries it has processed before). However, when a query is seen again, the Neo4j-Shell gives better timing results.

Also, as we noted in the discussion of the load performance test, the use of the `CREATE` statement with `MERGE` clauses can also slow down the creation and insertion of new features.

In future work, it may be useful to experiment with sequences of creation, modification, and deletion operations drawn from a more diverse workload. This may yield additional useful data about the performance, especially of Neo4j.

If being able to quickly create and insert a feature is a significant factor in the choice of feature model storage, then both MongoDB and MySQL seem to be good choices regardless of the size of the feature model. Neo4j behaves more erratically, but even with the larger feature models its performance is still quite acceptable and may improve as the "cache warms up". Although the Web interface does involve feature creation, modifications, and deletion, it does its work incrementally. Thus, it only needs a few such operations at a time.

### 9.3.4 Generate Performance Tests

Figure 9.4 shows a plot of the results of conducting the *Generate* performance tests for the three database encodings and ten feature models. For each database encoding and feature model, it shows the mean time (in seconds) taken to execute the test.

From the tables in Section 9.2 and the plot, we observe that for feature models up to size 6500 all database systems perform similarly, with Neo4j usually taking more time than MySQL and MongoDB. However, the time for the generate test increases significantly in the range between 6500 and 10,000 features. Beyond 10,000, all seem to increase more or less linearly, but MySQL and MongodB increase with steeper slopes than Neo4j. (The performance of MySQL seems a bit erratic, but still is more or less linear overall.) Neo4j generates the form more quickly for larger feature models.

The test to generate the product configuration does not modify the feature model. Its functioning relies heavily on traversals of the graph. As we expected, the graph-based

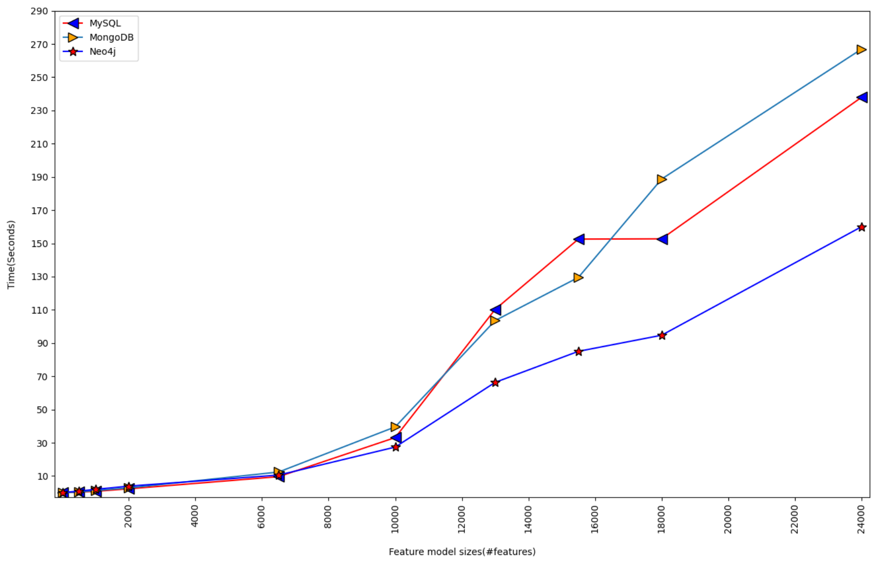


Figure 9.4. Combined Results for the *Generate* Performance Tests

database system performs significantly better in these kinds of operations than in operations that modify the feature model.

If being able to quickly generate a product configuration Web form is a significant factor in the choice of feature model storage, then any of the models perform reasonably for models up to about size 6500. However, Neo4j performs significantly better as the models grow beyond size 10,000. Thus, Neo4j is a good choice for feature models of any size.

### 9.3.5 Size Performance Tests

Figure 9.5 shows a plot of the results of conducting the *Size* performance tests for the three database encodings and ten feature models. For each database and feature model, it shows the size of the database's storage requirements in megabytes.

From Table 9.32 and the plot, we observe that the storage requirements (size) of the three database encodings grow approximately linearly with the size of the feature model. This is what we would expect. The MongoDB line has the shallowest slope and, hence, grows more slowly than the others. The MySQL line has a steeper slope, and the Neo4j line has the steepest slope of the three.

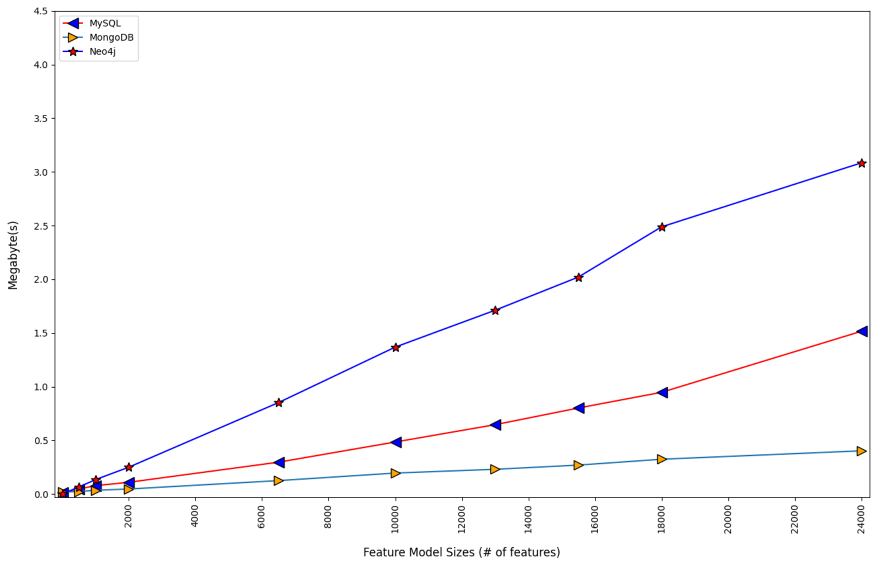


Figure 9.5. Combined Results for the *Size* Performance Tests

For the feature model with 24,000 nodes, MongoDB requires about 0.4 MB, MySQL about 1.5 MB, and Neo4j about 3.1 MB. MongoDB requires just 13 percent of the space that Neo4j does. We note, however, that the Neo4j database's size is a calculated value while the other two are measured, so this may not be a totally fair comparison. We also note that these storage requirements just include the essential aspects of the feature model; they do not include the full amount of storage needed for code, build scripts, and other metadata in a practical system.

If keeping the database storage requirement small is a significant factor in the choice of feature model storage, then MongoDB seems to be the best choice by far. However, the storage requirements for MySQL and Neo4j do not seem prohibitive when using contemporary systems with large, fast disks or solid state drives (such as the 256 gigabyte drive we used for our testing).

### 9.3.6 Performance Test Analysis Summary

We summarize the analysis of the performance test results on the three different database encodings as follows:

- MySQL and MongoDB both perform well on the feature model *load* test, regardless of the size of the feature model. Neo4j performs poorly as the feature model gets larger than 6500 features, likely because of the use of the `MERGE` clauses to merge new features and relationships into the feature model's graph.

- Regardless of the size of the feature model, MongoDB performs the best of the three on the feature model *empty* test, with MySQL also performing acceptably. However, Neo4j's performance deteriorates for feature models larger than 10,000 features.

- MySQL and MongoDB both perform well on the *create* test, regardless of the size of the feature model. Neo4j's performance is more erratic, likely because of the effects of query caching.

- For feature models larger than 10,000 features, Neo4j performs the best of the three on the test to *generate* a product configuration Web form. For smaller feature models, all three perform similarly.

- As shown in the *size* test, MongoDB is by far the most efficient in its use of storage space. It is followed by the less efficient MySQL, which, in turn, is followed by the even less efficient Neo4j.

In practical situations, we do not expect the small performance differences among the database encodings on the *empty* and *create* operations to be significant. We do not expect the emptying of a database to be a frequently occurring operation. For a Web interface such as the one presented in Chapter 4, operations like the *create* test do occur, but execution times of no more than 0.3 seconds should not degrade the Web interface's response time noticeably.

MySQL and MongoDB have excellent performance on the *load* performance test. What about Neo4j as the size of the feature model gets large?

One large feature model discussed in the scientific literature is the Linux kernel variability model [119], which has 5426 features. Consider the "large" feature model with 6500 features used in this experiment, which is close in size to the Linux kernel variability model. It has a height of 20, irregular branches where some paths are longer and contain more features than others, 341 features *required* by other features, and 377 features *excluded* by other features. Neo4j takes approximately 12 seconds to load this feature model from a CSV file. Given that we expect the loading of a feature model from an external file to be an infrequent operation, a 12-second load time is not a substantial issue for most applications.

Other large feature models discussed in the scientific literature have 18,000 or more features [64]. Consider the "huge" feature model with 18,000 features used in this experiment. Neo4j takes approximately 106 seconds to load this feature model from a CSV file. If the load operation must be performed frequently, Neo4j's performance becomes problematic. However, if this operation is seldom performed, then Neo4j's performance for other operations may lessen the effect of the long load times.

Neo4j is built around the concept of a graph, directly storing the relationships between entities. As a graph database, Neo4j is optimized for operations that navigate through the entities and relationships of a graph. Feature models are essentially directed acyclic graphs, so some of our operations on feature models can exploit these capabilities of Neo4j (e.g., for finding ancestors or finding paths to leaves). As we expect, Neo4j excels in the performance test to *generate* a product configuration form, which requires traverals of the feature model's graph. However, the superior performance of Neo4j over MySQL and MongoDB only becomes evident for feature models with more than 10,000 features.

The *size* performance test reveals that the Neo4j encoding has a much larger storage requirement than MongoDB and MySQL. For example, the feature model with 18,000 features requires about 1 MB of storage in MySQL and about 2.5 MB in Neo4j. If Neo4j's excellent performance on the *generate* and similar operations is important, then its higher storage costs may be worthwhile.

For feature models with 10,000 or fewer features, any of the database encodings seem to perform sufficiently well. However, for larger models, Neo4j's superior performance on the *generate* test indicates that it may be a better approach.

## 9.4 Analyzing the Subjective Criteria

In addition to the objective performance criteria considered in the previous sections, we also consider subjective criteria in our evaluation of the feature model encodings in the MySQL, MongoDB, and Neo4j database systems. To define the subjective criteria, we first identify key issues of interest to developers of feature-modeling applications. We then evaluate how well each database system handles each issue and analyze the results to determine how suitable each system is for the development of feature-modelings applications. We define the following subjective criteria for this research:

- Maturity and level of support
- Ease of installation
- Ease of programming

In this section, we analyze each criterion in turn.

### 9.4.1 Maturity and Level of Support

To analyze the subjective criterion on system *maturity and level of support*, we pose the following questions:

- How long has the database system existed and how widely is it used?
- How well is the database system supported?
- How well does the database system support popular programming languages?
- How well are users of the database system supported (e.g., by documentation, tutorials, and mechanisms for getting questions answered)?

We then evaluate each database system using these questions.

#### 9.4.1.1 Relational Database Systems and MySQL (MariaDB)

Relational database systems date back to the 1970s. ACM Turing Award winner Edgar F. Codd proposed the revolutionary concepts underlying relational database systems while working for International Business Machines (IBM) Research Laboratory in the early 1970s. Codd's seminal research paper "A Relational Model of Data for Large Shared Data Banks" [32] applied "elementary relation theory to systems which provide shared access to large banks of formatted data". Although initially slow to catch on, many different relational database systems have been released as proprietary products by companies such as IBM, Oracle, and Microsoft and as widely used open-source projects such as PostgreSQL and MySQL.

MySQL is one of the most widely used relational database systems. It was developed by the Swedish company MySQL AB in the mid-1990s as a free and open-source software project. In 2008, Sun Microsystems acquired MySQL AB. Then, in 2010, Oracle acquired Sun Microsystems. Because of concerns about Oracle's commitment to maintaining MySQL as a viable, free and open-source product, Michael Widenius, one of the founders of MySQL AB, created an open-source fork of MySQL called MariaDB in 2010 [40].

Although MariaDB [71] is just over a decade old, it is a continuation of a quarter century of MySQL development and a half century of experience with relational database systems. MariaDB is managed and supported by the non-profit MariaDB Foundation [72] as an open-source project with a diverse team of sponsors and contributors. In addition, enhanced services are offered by the separate MariaDB corporation. The first version of MariaDB released in 2011 was 5.1, following MySQL's version numbering. As of March 2021, the most recent stable release is 10.5.9. Sponsors and/or large users of MariaDB include technology companies such as Google, Microsoft, Alibaba, Mozilla, and Wikipedia.

In our research, we use the XAMPP software platform [9], which bundles MariaDB with the Apache HTTP server and the PHP and Perl programming languages. The bundle includes phpMyAdmin, a GUI for accessing MariaDB databases. MariaDB also supports connectors from other languages such as Python, JavaScript, Ruby, C/C++, Java, and C#. A full list of supported programming languages is given on the MariaDB company website [70].

MariaDB users are supported by a wealth of information sources. The MariaDB website [71] provides a complete set of documentation and other websites provide tutorials. Given that MariaDB remains mostly compatible with the popular RDB system MySQL, there are also a number of technical books and articles relevant to MariaDB [39, 40, 128]. In addition, MariaDB users can seek help on various question-and-answer forums for professional programmers (e.g., `stackoverflow.com`) and use various collaborative software project hosting platforms (e.g., `GitHub.com`).

#### 9.4.1.2 Document-oriented Database Systems and MongoDB

In the past two decades, a number of alternatives to relational database systems have emerged. These are often grouped under the broad term NoSQL [74]. One type of NoSQL databases is the document-oriented database. This type is useful for storing semistructured data sets such as JSON and XML documents. Well-known examples include MongoDB, Couchbase, CouchDB, and Firestore. In this research, we use MongoDB, partly because of its compatibility with JSON documents.

In 2007, the company 10gen—now named MongoDB, Inc. [79]—created the document-oriented database system MongoDB. It released version 1.0 of MongoDB as an open-source product in 2009. MongoDB, Inc. continues to support the community server as an open-source product and an expanded enterprise server and cloud services as commercial products. As of March 2021, the most recent stable version is 4.4.4. According to the MongoDB website [79], the software is used by over 50,000 companies. Other database products or services also support MongoDB-compatible application programming interfaces (APIs) for their offerings (e.g., Amazon DocumentDB and Microsoft's Azure Cosmos DB).

In our research, we use the MongoDB Compass [78], a GUI which includes a MongoDB server. MongoDB supports drivers for languages such as PHP, Perl, Python, JavaScript, Ruby, C/C++, Java, and C#. A full list of supported programming languages is given on the MongoDB website [81].

The user-level support for MongoDB is similar in scope to that of MariaDB, but, as a newer system, it has fewer overall information sources. MongoDB, Inc. provides a complete set of documentation [80] and other websites provide tutorials. There are also a number of technical books and articles relevant to NoSQL in general and MongoDB in particular [15]. In addition, MongoDB users can seek help on various question-and-answer forums for professional programmers (e.g., `stackoverflow.com`) and use various collaborative software project hosting platforms (e.g., `GitHub.com`).

#### 9.4.1.3 Graph Database Systems and Neo4j

Another type of NoSQL databases is the graph database. This type of database uses a graph structure to represent and store data. It defines nodes to store data entities and edges to store relationships between entities. Many of the concepts of graph databases go back to the hierarchical and network database systems of the 1960s [8, 120]. Unlike relational database systems, contemporary graph database systems differ considerably from those of a half century ago.

Neo4j is an open-source graph database that was created by Neo4j, Inc. Version 1.0 was released in 2010. The company continues to support the community edition as an open-source product and an expanded enterprise edition and cloud services as commercial products. As of March 2021, the most recent stable version is 4.2.3. Neo4j is used by many companies including eBay, Airbnb, Lyft, and Caterpillar.

In our research, we use the Neo4j Desktop [91], a GUI which includes a Neo4j server. Neo4j supports drivers for languages such as PHP, Perl, Python, JavaScript, Ruby, C/C++, Java, and C#. A full list of supported programming languages is given on the Neo4j website [93].

The user-level support for Neo4j is similar in scope to that of MariaDB and MongoDB, but, as a newer system not deployed as widely, it has fewer overall information sources.

Neo4j, Inc. provides excellent documentation, manuals, free courses, and a community forum [88]. In addition, there are several published technical articles [128] and books about Neo4j [129] and its query languages Cypher [87] and Gremlin [20]. As a newer product, Neo4j has less support than MySQL and MongoDB on the general question-and-answer forums for professional programmers such as `stackoverflow.com`.

#### 9.4.1.4 Most Mature and Best Supported?

MariaDB, MongoDB, and Neo4j all have positive aspects. All three have at least a decade of development and use. All three seem to be stable software systems with adequate support for repairing flaws and evolving to meet the needs of their users. All three have drivers for most major general-purpose programming languages. All three have good documentation and reference materials. All three are open-source projects, but MariaDB seems to have the most diverse team of contributors. MongoDB and Neo4j seems to be primarily supported by the like-named companies.

However, MySQL (and, hence, MariaDB) is clearly the most mature and has the best support among the three database systems we examined. The popular MongoDB system also seems to be more mature and better supported than the younger and less widely used Neo4j. This ranking is backed up by Stack Overflow's 2020 Annual Developer Survey [122], which had 49,537 responses from professional developers. In the survey question on databases, MySQL tops the list as the "most discussed", MongoDB comes in fourth, and Neo4j does not appear, but the software review site G2 ranks Neo4j as the highest rated among the graph database systems reviewed [42].

### 9.4.2 Ease of Installation

To analyze the subjective criterion on *ease of installation*, we pose the following questions:

- How convenient is the database system's installation process?
- How much disk space does the installed system require?

- How easy is to install drivers to enable programming languages to access the databases?

We then evaluate each database system using these questions. In this evaluation, we focus primarily on installation of the database system software on the Windows platform for use from Python 3.8 programs.

#### 9.4.2.1 MySQL Installation

To support our use of MySQL, we installed the XAMPP "stack" [9], a free and open-source, cross-platform Web server distribution developed by Apache Friends. XAMPP bundles the Apache HTTP server, the MariaDB database system [71, 111] (which is a fork of MySQL), the phpMyAdmin GUI for MariaDB, the PHP and Perl application programming languages, and related tools. The Windows version downloads as a Windows installer program (i.e., `.exe` file). The full installation of XAMPP requires 716 megabytes of file space. Our use of the XAMPP installer was convenient and trouble-free.

To enable our Python 3.8 programs to connect to a MySQL database, we installed the `sql.connector` driver [100] using the single command:

```
pip install mysql-connector-python
```

This `pip` program invocation selects the current version of the Python package `mysql-connector-python` from the Python Package Index (PyPi) and installs it, making this driver available for import by Python programs on the computer system. The use of `pip` was thus also convenient and trouble-free.

We also installed the MySQL driver (`mysql`) for NodeJS through the ExpressJS server-side framework. This effort was successful. PHP is already installed for MariaDB (MySQL) through the XAMPP software.

#### 9.4.2.2 MongoDB Installation

To support our use of MongoDB, we installed MongoDB Compass, a free interactive GUI [80]. The Windows version downloads as a Windows installer program (i.e., `.exe` file). The full installation requires 113 megabytes of file space for the MongoDB Compass GUI and 911 megabytes for the MongoDB Community Server version 4.4.2. Our use of the MongoDB Compass installer was convenient and trouble-free.

To enable our Python 3.8 programs to connect to a MongoDB database, we installed the `pymongo` driver using the single command:

```
pip install pymongo
```

As with MySQL, our use of `pip` was also convenient and trouble-free.

In addition, we installed the MongoDB driver `mongodb` for NodeJS through the ExpressJS server-side framework. This effort was successful. For PHP, we also installed the `mongodb` driver.

#### 9.4.2.3 Neo4j Installation

To support our use of Neo4j, we installed the Neo4j Desktop [88], a free integrated development environment (IDE) for Neo4j development. The Windows version downloads as a Windows installer program (i.e., `.exe` file). Because Neo4j is written in Java, the software requires that a Java Virtual Machine (JVM) runtime be available on the machine. If the JVM is not already installed, then it must also be downloaded and installed separately from the Neo4j Desktop. The full installation of Neo4j requires 648 megabyes for the Neo4j Dekstop version 1.4.1. The JVM requires an additional 126 megabytes. Our use of both the Neo4j Desktop and JVM installers were convenient and trouble-free.

To enable our Python 3.8 programs to connect to a Neo4j database, we installed the `neo4j` driver using the single command:

```
pip install neo4j
```

As with MySQL and MongoDB, our use of `pip` was also convenient and trouble-free.

We also installed the Neo4j driver (`neo4j-driver`) for NodeJS through the ExpressJS server-side framework. This effort was successful. Installing a PHP driver for Neo4j required testing several drivers to find a working version called NeoClient [46], a driver that has subsequently been upgraded and renamed the GraphAware Neo4j PHP Client [47].

#### 9.4.2.4 Easiest to Install?

In general, the installation of all three database systems and their GUIs went smoothly. They were packaged by their developers as convenient Windows installers, which made their installation a simple one-step process. Of course, if a JVM runtime was not available on the system already, the installation of Neo4j also required a separate step to install it. The disk space required for the database system installation is reasonable for complex, modern software systems—ranging from about 650 MB for Neo4j (not counting the JVM) to 1025 MB for MongoDB. In addition, installing the Python drivers using `pip` was also a convenient one-step process that we carried out without any glitches. Installation of the NodeJS and PHP drivers also went smoothly, except that we did encounter some difficulty in finding and installing a functional PHP driver for Neo4j.

### 9.4.3 Ease of Programming

To analyze the subjective criterion on *ease of programming*, we pose the following questions:

- How convenient is it to precisely encode feature models in the database systems?
- How convenient is it to precisely manipulate feature models in the database systems?

We then evaluate each database system using these questions.

#### 9.4.3.1 MySQL Programming

To encode a feature model in a MySQL database, we use a design consisting of three tables as explained in Chapter 3. We adopt the adjacency matrix as our method for representing a model. The three-column **featuresRelations** table has a parent feature in the first column, a child feature in the second column, and the type of relationship from the parent to the child in the third column. We include a row in this table if and only if there is a corresponding edge in the feature model. The **Features** table also stores all features in the feature model and the **Relations** table stores the relationship codes and their names. By using SQL or phpMyAdmin, it is easy and straightforward for us to implement this design.

As we also discuss in Chapter 3, a feature model can also be encoded in a CSV file with the same structure as the **featuresRelations** table. MySQL can readily export the **featuesRelations** table to a CSV file and import (or load) it from that format.

We use both MySQL's query language SQL [40] and a programming language (e.g., PHP or Python) to access and manipulate a feature model stored in a MySQL database as described above. SQL is the well-known standard language for storing, manipulating, and retrieving data in any relational database. It provides an excellent set of powerful statements and clauses for manipulating stored data. We use SQL to fetch a feature model's information and a programming language driver to send queries and receive information from the database. In Chapter 4, we show how to use these capabilities to create and insert a new feature, modify an existing feature, and delete an existing feature from a feature model. In Chapter 5, we also show how to use these capabilities to generate a Web form for configuring valid products by traversing the stored feature model.

We find the use of both the relational model and SQL queries to be simple and familiar. That is a strength of the MySQL encoding of feature models. However, a graph is not a native concept in relational databases (as it is in graph databases). So, a complex algorithm is needed to perform any operation that requires traversing the feature model's graph.

#### 9.4.3.2 MongoDB Programming

To encode a feature model in a MongoDB database, we use a design that stores a feature model in a collection of MongoDB documents as explained in Chapter 7. Each document within the collection consists of the three properties `fromFeature`, `toFeature`, and `relationType`, similar to the **featuresRelations** table discussed above in the MySQL design. A document defines the parent feature `fromFeature` to have a relationship of type `relationType` with the child feature `toFeature`. Within the collection, any two features have at most one such relationship defined. Each feature consists of one document. Thus, a feature model with 500 features would consist of 500 documents stored in a collection. By using MongoDB Compass with MQL or a programming language with an appropriate driver, it is easy and straightforward for us to implement this design.

The MongoDB encoding is thus quite similar to the **featuresRelations** table in the MySQL design and, hence, to the CSV encoding. Using either Compass or a programming language with an appropriate driver, we can readily export a MongoDB collection to a CSV file and import (or load) it from that format.

In Chapter 7, we chose a particular data model (pattern) for encoding a feature model that made some of our operations straightforward. However, there are other data models we could have chosen [80]. These models might be useful in circumstances where we would need to emphasize different operations.

MQL is a rich query language for fetching and manipulating documents in a MongoDB database. It includes the usual CRUD (Create, Read, Update, and Delete) operations plus text search, geospatial, and other useful queries [80]. The query language is simple and easy to use and interpret.

We find MongoDB documents as a convenient and intuitive way to encode feature models and MongoDB's query language a powerful way to manipulate the feature models. However, much like MySQL, a graph is not a native concept in MongoDB. So, a complex algorithm is needed to perform any operation that requires traversing the feature model's graph.

#### 9.4.3.3 Neo4j Programming

To encode a feature model in a Neo4j database, we use a design for a graph database to store an arbitrary feature model. This design stores the feature model's directed acyclic graph straightforwardly as a Neo4j graph. Each node of the Neo4j graph corresponds to a feature and each edge from one node to another corresponds to a relationship of the specified type between the corresponding features. By using the Neo4j Desktop with the Cypher query language [87] or a programming language with an appropriate driver, it is easy and straightforward for us to implement this design.

The Neo4j encoding is thus quite similar to the **featuresRelations** table in the MySQL design and, hence, to the CSV encoding. Using either the Neo4j Desktop or a programming language with an appropriate driver, we can readily export a Neo4j database to a CSV file and import (or load) it from that format.

Cypher [87] is a rich query language for fetching and manipulating graph structures in a Neo4j database. It is simple and easy to use and interpret. Although developed by Neo4j, Inc. for use with the Neo4j graph database, Cypher (i.e., openCypher [95]) has been been implemented for a number of other graph database systems and is part of an effort to specify a new Graph Query Language (GQL) international standard [58].

Neo4j is a graph database. Thus, it is straightforward to store a tree-like structure such as a feature model in a Neo4j database. In Neo4j, a traversal (e.g., to get all ancestors of a node) can be stated directly as a single Cypher query. Neo4j provides several builtin path-finding algorithms such as Breadth-First Search and Depth-First Search. These are quite helpful for traversing the hierarchical structure of a feature model. (More about the path-finding algorithms provided by Neo4j through Cypher can be found in the Neo4j Website's documentation for path-finding algorithms [92].) The Neo4j Desktop can also display a graphical view of the entire stored feature model. It also provides tools that enable a user to manipulate the feature model directly by manipulating its visual representation [91].

#### 9.4.3.4 Easiest to Program?

All three database systems can encode feature models precisely and conveniently, although each encodes the feature model's graph structure in different ways. The relational database system MySQL uses its familiar table structure to store the graph's nodes and edges and the powerful standard query language SQL to access the feature model. The document-oriented database system MongoDB uses its JSON-like documents to store the graph's nodes and edges and the powerful MongoDB-specific query language MQL to access the feature model. The graph database system Neo4j uses its native graph concepts of entities and relationships between them to store the graph's nodes and edges and the powerful query language Cypher to access the feature model.

Given that Neo4j was designed to store and manipulate graph-like structures, its representation is the most natural for feature models. Neo4j's visualization and manipulation tools also enhance Neo4j's support for a feature model as a graph.

However, MySQL's support of the standard query language SQL means it might be easier to port the MySQL application to other relational database systems. MongoDB and Neo4j currently use query languages developed primarily for those systems and that are not yet standardized.

All three database systems enable programs to access and manipulate the stored feature models. For all three, we designed algorithms to create, modify, and delete a feature. For all three, we also designed algorithms to generate a product configuration form. The implementations of these algorithms use the database systems' query languages, particularly the ability to embed query language statements in programming languages.

Neo4j's built-in support for graph traversals in its query language gives Neo4j an advantage in expressing algorithms such as those for generating the product configuration form. As we saw in the *Generate* performance test in Section 9.3.4, this can give Neo4J a performance advantage in some situations as well.

### 9.4.4 Subjective Criteria Analysis Summary

We summarize our analysis of the subjective criteria (maturity and level of support, ease of installation, and ease of programming) for the three database database systems (MySQL, MongoDB, and Neo4j) as follows:

- With a quarter century of development, MySQL is the most mature and best supported of the three database systems. Its 2010 fork MariaDB (that we use) continues to build on that legacy.

  However, all three database systems are reasonably stable, open-source, cross-platform products with a sufficient level of support and maturity for the purposes of our project.

- The Windows installer programs make the installation of each of the database systems a trouble-free, one-step process on Windows 10. Similarly, the `pip` tool also makes the installation of an individual Python driver a trouble-free, one-step process. The result is similar for installing NodeJS and PHP drivers in most cases.

  However, we did encounter difficulty in finding and installing a functional PHP driver for Neo4j, the newer and least widely used of the three database systems.

- Neo4j supports graphs as its native structure, so it provides the most straightforward support for encoding and manipulating feature models. The support for graph traversal built into Neo4j and its powerful query language also make many of our algorithms easier to express and more efficient to execute in that environment. MySQL and MongoDB do not directly support graph structures and their traversals.

  However, all three database systems can encode feature models precisely and conveniently, and all enable programs to access and manipulate the stored feature model as needed by our feature modeling application. So, the choice is partly a matter of the personal experiences and preferences of the software developers. MySQL's support for the international standard query language SQL and MongoDB's use of JSON-like documents are advantages for some.

Neo4j, the newest product with the smallest user base, probably entails the most risk, but it has also stimulated significant interest in several growing application areas and it is currently considered the leading graph database system. Neo4j's support of the openCypher [95] and Graph Query Language (GQL) [58] international standardization efforts may mitigate some of the long-term risk in using Neo4j and graph databases in general.

## 9.5 Answering the Research Question

Now, let us return to the chapter's research question. Which database system is the best for encoding feature models?

The answer is, as often is the case for such a question, that the “best” choice depends on the circumstances. The summary of the performance test results in Section 9.3.6 and of the subjective criteria in Section 9.4.4 articulate some issues to consider, or rules of thumb, for various circumstances.

In terms of the subjective criteria, the “best” choice is partly a matter of the experience and preference of the software development organization. MySQL is the most mature and better supported, Neo4j is the least mature and least well supported, and MongoDB is in between, but all three are sufficiently mature and well supported for our feature modeling project. The installation of the database systems and drivers for all three are straightforward. Some of the Neo4j drivers for some languages may be works in progress, but we experienced no problems with the Python drivers. The graph database Neo4j provides the most direct encoding of the feature model’s graph and a query language specialized for efficient graph operations. Although MySQL and MongoDB do not directly support graph concepts or operations, their organizational concepts and powerful query languages enable feature models to be encoded and manipulated effectively.

In terms of the performance tests, the “best” choice likely depends upon how the feature-modeling application is used. A reasonable usage scenario seems to be that a feature model is loaded once from a CSV file, evolved incrementally using the Web interface from Chapter 4, and used frequently to configure products using the live-preview Web form from Chapter 5. We assume that the feature-modeling application is executed on the same kind of platform on which we conducted the experiments (i.e., on a typical personal workstation in 2021). In this scenario, the *empty* performance test is of low relevance, the *load*, *create*, and *size* tests are of medium relevance, and the *generate* test of high relevance.

Given our experimental results, the performance depends upon how many features are in a feature model. For the purposes of our discussion, let us use the two sizes we used in Section 9.3.6—a “large” model with 6500 features and a “huge” model with 18,000 features.

For the “large” model in our usage scenario, any of the database encoding performs reasonably. All three encodings perform similarly on the performance tests except that Neo4j has a longer load time and uses more file space than the other two. However, Neo4j's 12 second load time and 0.85 megabyte space requirement are not a significant issue for these medium relevance tests.

For the "huge" model in our usage scenario, the Neo4j encoding is likely the "best". It has excellent performance on the high relevance test to *generate* a product configuration form. Unfortunately, it has poor performance on loading the feature model from the CSV file (106 seconds) and a larger file space requirement (2.5 MB) than the other two systems. However, we assume the feature model is only loaded once and that the file space usage is not an issue, even though 2.5 times what MySQL requires.

Given this usage scenario, Neo4j seems to be the "best" from a performance perspective, allowing us to accommodate huge models. Thus, the choice becomes one of the preferences of the software developers. We find Neo4j very interesting because of its specialization for storing and manipulating graphs.

However, we need to investigate Neo4j more broadly and deeply than we have done in this study. We need to study how we can more fully exploit its graph encoding and manipulation capabilities and how to optimize its performance (especially for loading feature models). We should also study how its query language is evolving toward an international standard. We need to develop a more complete pragmatic understanding of how to use Neo4j for the feature modeling application. We leave these to future work.

## 9.6 Conclusion

This chapter addresses specific Research Question 7 from Section 1.3: ***Which database system is the best for encoding feature models?***

To answer this question, we evaluate the three database-based feature model encodings defined in the previous chapters with respect to a set of objective and subjective criteria. In particular, we consider:

- the relational database (MySQL) encoding from Chapter 3
- the document-oriented database (MongoDB) encoding from Chapter 7
- the graph database (Neo4j) encoding from Chapter 8

To evaluate the encodings objectively, we define a set of experiments in Section 9.1 to determine how each encoding performs for several different feature models using several different performance tests.

We define ten feature models. The smallest is the `RasterVectorProcessing` feature model defined in Chapter 6 (shown in Figure 6.5). We also define nine feature models ranging in size from 500 features to 24,000 features, a size range that is representative of feature models in the real world. Table 9.1 shows the height and number of features, requires, and excludes for each model. For the experiments, we encode each feature model in a CSV file that will be loaded into a database system during the experiments.

To test the performance of each database-based feature model encoding, we also define five performance tests. We select these tests to be representative of the workloads that occur in practice. These include four tests to determine the time required to:

- *Load* a feature model into the database from a CSV file
- *Empty* the database
- *Create* a feature and add it to the feature model stored in the database after performing semantics checks
- *Generate* a product configuration form by traversing the complete feature model

We repeat each time-based performance test ten times for each pairing of a database encoding with a feature model, collect the times to complete the test, and compute the average time. We also define a fifth performance test to determine the *Size* of the database (i.e., the amount of storage space required), which we do once for each pairing of a database encoding with a feature model.

We show the results of the experiments in Section 9.2 and their analysis in Section 9.3. Section 9.3.6 summarizes this objective analysis.

To evaluate the encoding subjectively, in Section 9.4 we analyze the following criteria for each database system:

- Its maturity and level of support
- Its ease of installation
- Its ease of programming

For each criterion, we examine each database system to determine its suitability with respect to that criterion. Section 9.4.4 summarizes this subjective analysis.

Finally, in Section 9.5, we seek to answer the research question: Which database system is the best for encoding feature models? As we might expect, the answer is "It depends!"

# CHAPTER 10

# CONCLUSION

The research proposed in this dissertation seeks to answer the following general research question: ***Can mainstream Web and database technologies (relational, document-oriented, and graph) be used effectively to construct syntactically and semantically correct feature models and to configure products from these models? And, if so, which database system is best for encoding feature models?***

To achieve this, our research aims to answer the following specific research questions:

1. ***Can relational database tables be used to accurately encode feature models?***

   In Chapter 3, we demonstrated that the answer to this question is “Yes”. We encoded an arbitrary “traditional” feature model (as defined in Section 2.2) accurately as a directed acyclic graph in three relational database tables [116]. We also defined an equivalent feature model encoding in a CSV file. Furthermore, we showed that the design is practical by providing a proof-of-concept implementation and applying it to an example. Section 3.4 argues that our encoding is correct and gives a more detailed evaluation of our contributions related to this research question.

   We published a preliminary version of this work in 2017 [116].

2. ***Can mainstream Web and relational database technologies be used to construct correct feature models interactively and incrementally?***

   In Chapter 4, we demonstrated that the answer to this question is “Yes”. We proposed a novel design based on mainstream Web and relational database concepts. Our design uses three dynamic Web forms that incrementally construct feature models by interactively gathering information about the features and their relationships from the user [117]. These feature models are stored in a relational database designed according to our approach [116] described in Chapter 3. We implemented the Web interface design using mainstream relational database and Web technologies. The implementation validates its inputs and ensures that the feature model stored in the database is syntactically and semantically correct at any time. Section 4.3 argues that our design is correct and gives a more detailed evaluation of our contributions related to this research question.

We published a preliminary version of this research in 2020 [116].

3. ***Can mainstream Web and relational database technologies be used to configure correct products corresponding to a feature model?***

   In Chapter 5, we demonstrated that the answer to this question is "Yes". We designed and implemented a Web form that, given a syntactically and semantically correct feature model stored in the relational database [116, 117], enables the user to select any set of features from the feature model that corresponds to a correct configuration of a product. Section 4.3 argues that our design is correct and gives a more detailed evaluation of our contributions related to this research question.

   We published preliminary versions of this work in 2017 [116] and 2020 [117].

4. ***Can JSON technologies be used to represent feature models correctly and enable them to be exchanged in textual form?***

   In Chapter 6, we demonstrated that the answer to this question is "Yes". We designed an approach that can encode an arbitrary "traditional" feature model accurately in a JSON document in a manner that is equivalent to the RDB encoding defined in Chapter 3. We also designed and implemented programs that can translate a valid RDB encoding of a feature model to an equivalent JSON encoding and vice versa. In addition, we have operations to create, modify, and delete features in a JSON-encoded feature model. Section 6.6 argues that our design is correct and gives a more detailed evaluation of our contributions related to this research question.

We published preliminary version of this work in 2021 [118].

5. ***Can a document-oriented NoSQL database be used to accurately encode feature models?***

   In Chapter 7, we demonstrated that the answer to this question is "Yes". We designed an approach that can encode an arbitrary "traditional" feature model accurately in a document-oriented MongoDB database in a manner that is equivalent to the RDB and CSV encodings defined in Chapter 3. We also designed and implemented operations to load a feature model into a database; empty a database; create, modify, and delete features in an encoded feature model; and generate a product configuration form from the encoded model. Furthermore, we showed that the approach is practical by using the implementations in the experiments in Chapter 9. Section 7.6 argues that our encoding is correct and gives a more detailed evaluation of our contributions related to this research question.

6. ***Can a graph-oriented NoSQL database be used to accurately encode feature models?***

   In Chapter 8, we demonstrated that the answer to this question is "Yes". We designed an approach that can encode an arbitrary "traditional" feature model accurately in a document-oriented Neo4j database in a manner that is equivalent to the RDB and CSV encodings defined in Chapter 3. We also designed and implemented operations to load a feature model into a database; empty a database; create, modify, and delete features in an encoded feature model; and generate a product configuration form from the encoded model. Furthermore, we showed that the approach is practical by using the implementations in the experiments in Chapter 9. Section 8.6 argues that our encoding is correct and gives a more detailed evaluation of our contributions related to this research question.

7. ***Which database system is the best for encoding feature models?***

   To answer this question in Chapter 9, we evaluated the three database encodings (defined in Chapters 3, 7, and 8) against sets of objective and subjective criteria. We selected the criteria carefully to help us determine which encodings are "best" from various perspectives.

   To evaluate the database encodings objectively, we defined a set of experiments to determine how each performs for ten different feature models using five different performance tests. We selected feature models ranging in size from 19 features to 24,000 features, a range that should include most real-world models. We selected performance tests that are representative of the workloads that occur in practice. We conducted the experiments and collected measurements, and then we analyzed the results with a focus on determining how well each database encoding performs on the tests as the feature models increase in size. We then stated a few rules of thumb to guide decisions about which is best under what circumstances.

   To evaluate the database systems subjectively, we identified three different issues of interest to software developers and elaborated each by defining a set of two-to-four related questions. We evaluated each database system against the questions. We then analyzed the results with a focus on determining how suitable each database system is for the development of feature-modelings applications. We then again stated a few rules of thumb to guide decisions about which is best under what circumstances.

   To unify the results of the analyses, we considered a typical usage scenario for a feature-modeling application and two representative feature models—one "large" model and one "huge" model. Within this scenario, we can suggest which encoding may be "best" for our feature-modeling application. In general there is no one best choice, but, by applying the rules of thumb, software developers can make good choices.

# CHAPTER 11

# FUTURE WORK

In this dissertation research project, we have investigated and answered seven research questions. However, like most research efforts, new questions arise from the process of answering the old ones. In this chapter we identify several questions that we, or others, can explore in future research.

1. How can we extend our feature-modeling approach to enable concrete software products to be generated from a feature model?

   We plan to extend the database design to include information about the internal design and implementation of features. Our objective is to be able to build products corresponding to a selected product configuration.

2. How can we extend our feature-modeling approach to incorporate complex cross-tree constraints?

   In this dissertation, we consider normal cross-tree constraints that most researches use in their work (e.g., feature A requires feature B or feature A excludes feature B). Some recent work on large feature models suggest the use of complex cross-tree constraints, which are sets of propositional formulas that cannot be expressed using simple constraints (i.e., requires and excludes) [64]. We plan to investigate whether complex cross-tree constraints are a useful extension to our approach and, if so, how to incorporate them effectively.

3. How can we validate the syntax and semantics of feature models encoded in JSON?

One promising approach is to use the emerging JSON Schema technologies, as defined in the draft standard [59] and implemented by prototype tools such as the JavaScript (Node.js) package Ajv [105]. We plan to define a schema that specifies and enables validation of most aspects of the syntax of JSON-encoded feature models. We plan to use the JSON Schema types and its properties such as `type`, `pattern`, `properties`, `required`, `additionalProperties`, `items`, and `enum` to make the syntactic rules precise. We also plan to use the `$defs` and `$ref` properties to define and document simple abstractions that are used within a schema, such as for the "feature name", "feature tree", "feature list", "type", and "relationship" validators.

4. How can we improve the performance of the Neo4j encoding, particularly in operations to create new nodes and load databases from external files?

   In Chapter 9 we noted in the discussion of the *load* performance test that the use of the `CREATE` statement with `MERGE` clauses slows down the creation and insertion of new features. Also, in the discussion of the *create* performance test, we noted that Neo4j's caching of queries sometimes results in erratic behaviors. In future work, it may be useful to experiment with sequences of creation, modification, and deletion operations drawn from a more diverse workload. This may yield additional useful data that can enable us to better exploit Neo4j's use of lazy and eager evaluation and query caching.

5. How can we improve the performance of our feature-modeling system by changing the implementation languages and/or drivers?

   In our experimental study, we used the interpreted application languages Python, PHP, and JavaScript and their available drivers. We may be able to improve performance by switching to a compiled language such as Java and its drivers. We may also be able to improve performance by experimenting with alternative drivers for our current languages.

6. Which Neo4j query language best supports encoding and manipulating feature models?

For the Neo4j-related research in Chapters 8 and 9, we used Cypher, a declarative language for querying graph databases [95]. It is an easy to use graph query language that provides powerful statements and clauses for manipulating and traversing graphs. Neo4j also supports Gremlin [20], a graph traversal query language that is both declarative and imperative. A strength of Gremlin is that it allows users to define and use their own custom algorithms for graph traversals. They can potentially optimize the traversals to fit the specific natures of their applications. Cypher, by contrast, does not allow customization. Instead, it seeks to choose the best algorithm for a particular circumstance from among its builtin traversal algorithms. We plan to compare the two languages using objective and subjective criteria similar to the ones we used in Chapter 9.

7. Which document-oriented database is more suitable to encode and manipulate feature models?

   For the research in Chapters 7 and 9, we used the document-oriented database system MongoDB. We plan to investigate other document-oriented database systems such as CouchDB [6] and ArangoDB [11]. Both are free and open-source systems with positive reviews. We plan to compare either CouchDB or ArangoDB against MongoDB using objective and subjective criteria similar to the ones we used in Chapter 9.

8. Which graph-based database is more suitable to encode and manipulate feature models?

   For the research in Chapters 8 and 9, we used the graph database system Neo4j. We plan to investigate other graph database systems such as Dgraph [37] and ArangoDB [11]. Both are free and open-source systems with positive reviews. We plan to compare either Dgraph or ArangoDB against Neo4j using objective and subjective criteria similar to the ones we used in Chapter 9.

# BIBLIOGRAPHY

# BIBLIOGRAPHY

[1] Serge Abiteboul. Querying semi-structured data. In *International Conference on Database Theory*, ICDT '97, pages 1–18, Delphi, Greece, 1997. Springer.

[2] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert France. FAMILIAR: A domain-specific language for large scale management of feature models. *Science of Computer Programming*, 78:657–681, 2013.

[3] Md Mottahir Alam, Asif Khan, and Aasim Zafar. Implementing variability in SPL using FeatureIDE: A case study. In *Proceedings of the International Conference on Electrical, Electronics, Computers, Communication, Mechanical and Computing (EECCMC)*, pages 584–593, Tamil Nadu, India, January 2018. IEEE Madras Section.

[4] Mohamed Alloghani, Dhiya Al-Jumeily, Abir Hussain, Ahmed Aljaaf, Jamila Mustafina, Mohamed Khalaf, and Sin Ying Tan. The XML and semantic web: A systematic review on technologies. In *International Conference on Big Data Analytics, Data Mining and Computational Intelligence*, pages 92–102, Los Angeles, CA, USA, 2019. IEEE.

[5] Nele Andersen, Krzysztof Czarnecki, Steven She, and Andrzej Wąsowski. Efficient synthesis of feature models. In *Proceedings of the 16th International Software Product Line Conference (SPLC), Volume 1*, pages 106–115, Salvador, Brazil, 2012. ACM.

[6] J. Chris Anderson, Jan Lehnardt, and Noah Slater. *CouchDB: The Definitive Guide*. O'Reilly Media, Sebastopol, CA, USA, February 2010.

[7] Oracle Corporation and/or its affiliates. *MySQL Community Downloads – Connector/Python*, 2021. URL `https://dev.mysql.com/downloads/connector/python/`. Retrieved March 11, 2021.

[8] Renzo Angles and Claudio Gutierrez. Survey of graph database models. *ACM Computing Surveys*, 40(1):1–39, 2008.

[9] Apache Friends. *XAMPP Apache + MariaDB + PHP + Perl*, 2021. URL `https://www.apachefriends.org/index.html`. Retrieved March 14, 2021.

[10] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines*. Springer, 2013.

[11] ArangoDB Inc. *ArangoDB Documentation*, 2021. URL `https://www.arangodb.com/documentation/`. Retrieved March 20, 2021.

[12] Paolo Arcaini, Angelo Gargantini, and Marco Radavelli. Achieving change requirements of feature models by an evolutionary approach. *Journal of Systems and Software*, 150:64–76, 2019.

[13] Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. Schemas and types for JSON data: From theory to practice. In *Proceedings of the 2019 International Conference on the Management of Data*, Amsterdam, The Netherlands, 2019. ACM.

[14] Ebrahim Bagheri, Faezeh Ensan, and Dragan Gasevic. Decision support for the software product line domain engineering lifecycle. *Automated Software Engineering*, 19: 335–377, 2011.

[15] K. Banker, P. Bakkum, S. Verch, and D. Garrett. *MongoDB in Action*. Manning, Shelter Island, NY, USA, second edition, 2016.

[16] Lindsay Bassett. *Introduction to JavaScript Object Notation: A To-the-point Guide to JSON*. O'Reilly Media, Sebastopol, CA, USA, 2015.

[17] Don Batory. The road to Utopia: A future for generative programming. In *Domain-Specific Program Generation*, number 3016 in LNCS, pages 1–18. Springer, 2004.

[18] Don Batory. Feature models, grammars, and propositional formulas. In *International Conference on Software Product Lines*, pages 7–20, Rennes, France, 2005. Springer.

[19] Don Batory, Roberto E. Lopez-Herrejon, and J.-P. Martin. Generating product-lines of product-families. In *17th IEEE International Conference on Automated Software Engineering*, pages 81–92, Edinburgh, UK, 2002. IEEE.

[20] Dave Bechberger and Josh Perryman. *Graph Databases in Action*. Manning, Shelter Island, NY, USA, November 2020.

[21] David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz Cortés. FAMA: Tooling a framework for the automated analysis of feature models. In *Proceeding of the First International Workshop on Variability Modelling of Software-Intensive Systems*, VaMoS, pages 129–134, Limerick, Ireland, January 2007. VaMoS.

[22] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wąsowski. A survey of variability modeling in industrial practice. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*, pages 1–8, Pisa, Italy, 2013. ACM.

[23] Ted J. Biggerstaff. A perspective of generative reuse. *Annals of Software Engineering*, 5(169), 1998.

[24] Tim Bray, Jean Paoli, C Michael Sperberg-McQueen, Eve Maler, and Franois Yergeau. Extensible Markup Language (XML) 1.0 (fifth edition), November 2008. URL `https://www.w3.org/TR/2008/REC-xml-20081126`. Retrieved March 23, 2021.

[25] Anca-Raluca Breje, Robert Gyorodi, Cornelia Győrödi, Doina Zmaranda, and George Pecherle. Comparative study of data sending methods for XML and JSON models. *International Journal of Advanced Computer Science and Applications*, 9, 2018.

[26] Peter Buneman. Semistructured data. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS'97, pages 117–121, Tucson, AZ, USA, 1997. ACM.

[27] Johannes Bürdek, Sascha Lity, Malte Lochau, Markus Berens, Ursula Goltz, and Andy Schürr. 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 (VaMoS)*, page 16, Sophia Antipolis, France, 2014. ACM.

[28] Jessie Carbonnel, David Delahaye, Marianne Huchard, and Nebut Clémentine. Graph-based variability modelling: Towards a classification of existing formalisms. In *Proceedings of the International Conference on Conceptual Structures*, ICCS, pages 27–41, Marburg, Germany, 2019. Springer.

[29] Vaclav Cechticky, Alessandro Pasetti, Ondrej Rohlik, and Walter Schaufelberger. XML-based feature modelling. In *International Conference on Software Reuse*, volume 3107, pages 101–114, Madrid, Spain, July 2004. Springer.

[30] Lianping Chen, Muhammad Ali Babar, and Nour Ali. Variability management in software product lines: A systematic review. In *Proceedings of the 13th International Software Product Line Conference*, pages 81–90, San Francisco, CA, USA, 2009. ACM.

[31] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns.* Addison-Wesley Professional, Boston, MA, USA, 2002.

[32] Edgar F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377—-387, June 1970. URL `https://doi.org/10.1145/362384.362685`.

[33] Doughlas Crockford. *Introducing JSON*, 2021. URL `http:://json.org`. Retrieved March 23, 2021.

[34] Krysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Techniques, and Applications.* Addison-Wesley, Boston, MA, USA, 1999.

[35] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005.

[36] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Staged configuration through specialization and multilevel configuration of feature models. *Software Process: Improvement and Practice*, 10(2):143–169, April 2005.

[37] Dgraph Labs, Inc. *Get Started with Dgraph*, 2021. URL `https://dgraph.io/docs`. Retrieved March 20, 2021.

[38] Michael Droettboom. Understanding JSON Schema. Technical report, Space Telescope Institute, 2016 (updated 12 April 2020). URL `https://json-schema.org/understanding-json-schema`. Retrieved March 11, 2021.

[39] Paul DuBois. *MySQL*. Addison-Wesley Professional, Boston, MA, USA, 2013.

[40] Russell J.T. Dyer. *Learning MySQL and MariaDB*. O'Reilly Media, Sebastopol, CA, USA, 2015.

[41] Stefan Ferber, Jürgen Haag, and Juha Savolainen. Feature interaction and dependencies: Modeling features for reengineering a legacy product line. In *International Conference on Software Product Lines*, pages 235–256, San Diego, CA, USA, 2002. Springer.

[42] G2.com. *Best Graph Databases*, 2021. URL `https://www.g2.com/categories/graph-databases/`. Retrieved March 15, 2021.

[43] GDAL/OGR Contributors. *GDAL/OGR Geospatial Data Abstraction Software Library*. Open Source Geospatial Foundation, 2021. URL `https://gdal.org/`. Retrieved March 15, 2021.

[44] Guozheng Ge and E. James Whitehead. Rhizome: A feature modeling and generation platform. In *Proceedings of the International Conference on Automated Software Engineering*, pages 375–378, L'Aquila, Italy, 2008. IEEE.

[45] Mark L. Gillensonm. *Fundamentals of Database Management Systems*. Wiley, 2011.

[46] GitHub, Inc. *Neoxygen's NeoClient*, 2021. URL `https://github.com/neoxygen/neo4j-neoclient`. Retrieved March 18, 2021.

[47] GitHub, Inc. *GraphAware Neo4j PHP Client*, 2021. URL `https://github.com/graphaware/neo4j-php-client`. Retrieved March 18, 2021.

[48] GitHub, Inc. *PyMongo: Mongo Python Driver*, 2021. URL `https://github.com/mongodb/mongo-python-driver/`. Retrieved March 11, 2021.

[49] Mark Goodyear. *Enterprise System Architectures: Building Client Server and Web Based Systems*. CRC Press, 2017.

[50] Google, Inc. *V8: A High-performance JavaScript Engine*, 2021. URL `http://v8.dev`. Retrieved March 23, 2021.

[51] Dan Gookin. *Guide to XML and JSON Programming*. independently published (Amazon.com), 2019.

[52] Dave Gordon. *Large Delete Transaction Best Practices in Neo4j*, 2021. URL `https://neo4j.com/developer/kb/large-delete-transaction-best-practices-in-neo4j/`. Retrieved March 12, 2021.

[53] Martin L. Griss, John Favaro, and Massimo d'Alessandro. Integrating feature modeling with the RSEB. In *Proceedings of the Fifth International Conference on Software Reuse*, pages 76–85, Victoria, BC, Canada, 1998. IEEE.

[54] Sebastian Günther and Sagar Sunkle. rbFeatures: Feature-oriented programming with Ruby. *Science of Computer Programming*, 77(3):152–173, 2012.

[55] Johannes I. M. Halman, Adrian P. Hofer, and Wim van Vuuren. Platform-driven development of product families: Linking theory with practice. *Journal of Product Innovation Management*, 20(2):149–162, 2003.

[56] Maarit Harsu. A survey on domain engineering. Technical Report 12, Tampere University of Technology, Tampere, Finland, 2002.

[57] Michael Hunger, Andrea Santurbano, et al. *Awesome Procedures for Neo4j (APOC) 4.2.x*, 2021. URL `https://neo4j.com/developer/neo4j-apoc/`. Retrieved March 12, 2021.

[58] ISO Graph Query Language Proponents. *Graph Query Language GQL: Standards*, 2020. URL `https://www.gqlstandards.org/`. Retrieved March 19, 2021.

[59] JSON Schema Organisation. *JSON Schema Specification 2019-09*, 2019. URL `http://json-schema.org/specification.html`. Retrieved March 11, 2021.

[60] JSON Schema Organisation. *JSON Schema*, 2020. URL `http://json-schema.org`. Retrieved March 11, 2021.

[61] JSON.org. *Introducing JSON*, 2021. URL `https://www.json.org/json-en.html`. Retrieved March 11, 2021.

[62] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, 1990.

[63] Kyo C. Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin, and Moonhang Huh. FORM: A feature-oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering*, 5(1):143–168, 1998.

[64] Alexander Knüppel, Thomas Thüm, Stephan Mennicke, Jens Meinicke, and Ina Schaefer. Is there a mismatch between real-world feature models and product-line research? In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*, pages 291–302, Paderborn, Germany, 2017. ACM. doi: 10.1145/3106237.3106252.

[65] Heba A. Kurdi. Review on aspect oriented programming. *International Journal of Advanced Computer Science and Applications*, 4(9):22–27, 2013.

[66] Thomas Leich, Sven Apel, Laura Marnitz, and Gunter Saake. Tool support for feature-oriented software development: FeatureIDE: An Eclipse-based approach. In *Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology eXchange, Eclipse'05*, pages 55–59, San Diego, CA, USA, 2005. ACM.

[67] Daniela Lettner, Klaus Eder, Paul Grünbacher, and Herbert Prähofer. Feature modeling of two large-scale industrial software systems: Experiences and lessons learned. In *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 386—-395, Ottawa, ON, Canada, 2015. IEEE.

[68] Stephen Ludin and Javier Garza. *Learning HTTP/2: A Practical Guide for Beginners.* O'Reilly Media, Sebastopol, CA, USA, 2017.

[69] Mike Mannion. Using first-order logic for product line model validation. In *International Conference on Software Product Lines*, pages 176–187, San Diego, CA, USA, 08 2002. Springer.

[70] MariaDB. *Application Programming Interfaces*, 2021. URL `https://mariadb.com/kb/en/connectors/`. Retrieved March 15, 2021.

[71] MariaDB. *MariaDB: Knowlesge Base*, 2021. URL `https://mariadb.com/`. Retrieved March 14, 2021.

[72] MariaDB Foundation. *MariaDB Foundation*, 2021. URL `https://mariadb.org/`. Retrieved March 14, 2021.

[73] Tom Marrs. *JSON at Work.* O'Reilly Media, Sebastopol, CA, USA, 2017.

[74] A. Meier and M. Kaufmann. *SQL & NoSQL Databases: Models, Languages, Consistency Options and Architectures for Big Data Management.* Springer, Berlin, 2019.

[75] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. *Mastering Software Variability with FeatureIDE.* Springer, 2017.

[76] Marcılio Mendonça. *Efficient Reasoning Techniques for Large Scale Feature Models.* PhD thesis, University of Waterloo, Waterloo, ON, Canada, 2009.

[77] MongoDB, Inc. *PyMongo 3.11.3 Documentation*, 2021. URL `https://pymongo.readthedocs.io/en/stable/`. Retrieved March 11, 2021.

[78] MongoDB, Inc. *MongoDB Compass*, 2021. URL `https://www.mongodb.com/products/compass/`. Retrieved March 15, 2021.

[79] MongoDB, Inc. *MongoDB: A Complete Data Framework*, 2021. URL `https://www.mongodb.com/`. Retrieved March 15, 2021.

[80] MongoDB, Inc. *The MongoDB 4.4 Manual*, 2021. URL `https://docs.mongodb.com/manual/`. Retrieved March 11, 2021.

[81] MongoDB, Inc. *Community Library*, 2021. URL `https://docs.mongodb.com/drivers/community-supported-drivers/`. Retrieved March 15, 2021.

[82] Mukelabai Mukelabai, Damir Nešić, Salome Maro, Thorsten Berger, and Jan-Philipp Steghöfer. Tackling combinatorial explosion: A study of industrial needs and practices for analyzing highly configurable systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 155–166, Montpellier, France, 2018.

[83] Chiranjib Mukherjee and Gyan Mukherjee. Role of adjacency matrix in graph theory. *IOSR Journal of Computer Engineering (IOSR-JCE)*, 16:58–63, 2014.

[84] Makoto Murata, Dongwon Lee, Murali Mani, and Kohsuke Kawaguchi. Taxonomy of XML schema languages using formal language theory. *ACM Transactions on Internet Technologies*, 5(4):660–704, November 2005.

[85] James M. Neighbors. *Software Construction using Components*. PhD thesis, University of California, Irvine, 1980.

[86] James M. Neighbors. Draco: A method for engineering reusable software systems. In *Software Reusability: Concepts and Models*, volume 1, pages 295–319. ACM, 1989.

[87] Neo4j, Inc. *Cypher Query Language*, 2021. URL `https://neo4j.com/developer/cypher/`. Retrieved March 11, 2021.

[88] Neo4j, Inc. *Neo4j – The Leader in Graph Databases*, 2021. URL `https://neo4j.com/company/`. Retrieved March 11, 2021.

[89] Neo4j, Inc. *Using Neo4j from Python*, 2021. URL `https://neo4j.com/developer/python/`. Retrieved March 11, 2021.

[90] Neo4j, Inc. *Neo4j Patterns*, 2021. URL `https://neo4j.com/docs/cypher-manual/current/syntax/patterns/`. Retrieved March 11, 2021.

[91] Neo4j, Inc. *Neo4j Desktop*, 2021. URL `https://neo4j.com/product/`. Retrieved March 15, 2021.

[92] Neo4j, Inc. *Path Finding Algorithms*, 2021. URL `https://neo4j.com/docs/graph-data-science/current/algorithms/pathfinding/`. Retrieved March 18, 2021.

[93] Neo4j, Inc. *Drivers & Language Guides*, 2021. URL `https://neo4j.com/developer/language-guides/`. Retrieved March 15, 2021.

[94] Neo4j, Inc. *Neo4j Graph Platform*, 2021. URL `https://neo4j.com`. Retrieved March 11, 2021.

[95] Neo4j, Inc. *openCypher*, 2021. URL `http://www.opencypher.org/`. Retrieved March 19, 2021.

[96] Linda Northrop. Software product lines essentials. Technical report, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, 2008.

[97] Linda Northrop, Paul Clements, Felix Bachmann, John Bergey, Gary Chastek, Sholom Cohen, Patrick Donohoe, Lawrence Jones, Robert Krut, Reed Little, John McGregor, and Liam O'Brien. A framework for software product line practice, version 5.0. Technical report, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, 2007.

[98] Regina O. Obe and Leo S. Hsu. *PostgreSQL: Up and Running: A Practical Introduction to the Advanced Open Source Database*. O'Reilly Media, Sebastopol, CA, USA, 2014.

[99] Travis E. Oliphant. *A Guide to NumPy*, volume 1. Trelgol Publishing, USA, 2006.

[100] Oracle Corporation. *MySQL Connectors*, 2021. URL `https://www.mysql.com/products/connector/`. Retrieved March 11, 2021.

[101] David Lorge Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, 2(1):1–9, 1976.

[102] David Lorge Parnas. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering*, 5(2):128–138, 1979.

[103] Pablo Parra, Óscar R. Polo, Segundo Esteban, Agustín Martínez, and Sebastián Sánchez. A component-based approach to feature modelling. In *Proceedings of the 23rd International Systems and Software Product Line Conference – Volume B*, SPLC, pages 137–142, Paris, France, 2019. ACM.

[104] Felipe Pezoa, Juan Reutter, Fernando Suarez, Martín Ugarte, and Domagoj Vrgoč. Foundations of JSON Schema. In *Proceedings of the 25th International Conference on World Wide Web*, pages 263–273, Montreal, QC, Canada, April 2016. International World Wide Web Conferences Steering Committee.

[105] Evgeny Poberezkin. *Ajv: Another JSON Schema Validator*, 2021. URL `https://ajv.js.org`. Retrieved March 15, 2021.

[106] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.

[107] Shelley Powers. *Learning Node*. O'Reilly Media, Sebastopol, CA, USA, second edition, 2016.

[108] Rubén Prieto-Díaz. Domain analysis: An introduction. *SIGSOFT Software Engineering Notes*, 15(2):47–54, 1990.

[109] pure-systems GmbH. *pure::variants User's Guide.* Magdeburg, Germany, 2019. URL `https://www.pure-systems.com`. Retrieved March 11, 2021.

[110] Python Software Foundation. *Python's `time` Module: Time Access and Conversions*, 2021. URL `https://docs.python.org/3/library/time.html`. Retrieved March 11, 2021.

[111] Federico Razzoli. *Mastering MariaDB.* Packt Publishing, 2014.

[112] Matthias Riebisch, Kai Böllert, Detlef Streitferdt, and Ilka Philippow. Extending feature diagrams with UML multiplicities. In *Proceedings of the 6th World Conference on Integrated Design and Process Technology (IDPT)*, pages 1–7, Pasadena, CA, USA, 2002. Society for Design and Process Science.

[113] José Rocha. *Understanding Neo4j's Data on Disk.* Neo4j, Inc., 2021. URL `https://neo4j.com/developer/kb/understanding-data-on-disk/`. Retrieved March 11, 2021.

[114] Jean-Claude Royer and Hugo Arboleda. *Model-Driven and Software Product Line Engineering.* Wiley, 2013.

[115] Charles Severance. Discovering JavaScript Object Notation. *IEEE Computer*, 45(4): 6–8, 2012.

[116] Hazim Shatnawi and H. Conrad Cunningham. Mapping SPL feature models to a relational database. In *Proceedings of the ACM SouthEast Conference*, pages 42–49, Kennesaw, GA, USA, April 2017. ACM.

[117] Hazim Shatnawi and H. Conrad Cunningham. Automated analysis and construction of feature models in a relational database using web forms. In *Proceedings of the ACM SouthEast Conference*, Tampa, FL, USA, April 2020. ACM.

[118] Hazim Shatnawi and H. Conrad Cunningham. Encoding feature models using mainstream JSON technologies. In *Proceedings of the ACM SouthEast Conference*, Virtual Event, USA, accepted for publication, April 2021. ACM.

[119] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wasowski, and Krzysztof Czarnecki. The variability model of the linux kernel. In *Proceedings of the Fourth International Workshop on Variability Modelling of Software-Intensive Systems*, VaMoS, pages 45–51, Linz, Austria, 01 2010. Institute for Computer Science and Business Informatics (ICB).

[120] Avi Silberschatz, Michael Stonebraker, and Jeff Ullman. Database systems: Achievements and opportunities. *Communications of the ACM*, 34(10):110–120, 1991.

[121] Anjali Sree-Kumar, Elena Planas, and Robert Clarisó. 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)*, volume 206 of

*Electronic Proceedings in Theoretical Computer Science*, pages 45–60, Eindhoven, The Netherlands, 2016. Open Publishing Association.

[122] Stack Exchange, Inc. *Stack Overflow 2020 Developer Survey*, 2021. URL `https://insights.stackoverflow.com/survey/2020#technology-databases-all-respondents4`. Retrieved March 15, 2021.

[123] Stackoverflow.com. *Cypher Execution Time When Running Same Query Multiple Times*, 2021. URL `https://stackoverflow.com/questions/60680417/neo4j-query-execution-time-when-executing-the-same-query-multiple-times-only-t`. Retrieved March 14, 2021.

[124] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. A classification and survey of analysis strategies for software product lines. *ACM Computing Surveys*, 47(1):1–45, 2014.

[125] Zia ul Haq, Gul Faraz Khan, and Tazar Hussain. A comprehensive analysis of XML and JSON web technologies. *New Developments in Circuits, Systems, Signal Processing, Communications and Computers*, pages 102–109, 2015.

[126] Arie van Deursen and Paul Klint. Domain-specific language design requires feature descriptions. *CIT. Journal of Computing and Information Technology*, 10(1):1–17, 2002.

[127] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, 2000.

[128] Chad Vicknair, Michael Macias, Zhendong Zhao, Xiaofei Nan, Yixin Chen, and Dawn Wilkins. A comparison of a graph database and a relational database: A data provenance perspective. In *Proceedings of the ACM SouthEast Conference*, pages 1–6, Oxford, MS, USA, April 2010. ACM.

[129] Aleksa Vukotic, Nicki Watt, Tareq Abedrabbo, Dominic Fox, and Jonas Partner. *Neo4j in Action.* Manning, Shelter Island, NY, December 2014.

[130] Priscilla Walmsley. *Definitive XML Schema, 2nd Edition.* Prentice Hall, 2012.

[131] Jules White, David Benavides, Douglas C. Schmidt, Pablo Trinidad, Brian Dougherty, and Antonio Ruiz-Cortés. Automated diagnosis of feature model configurations. *Journal of Systems and Software*, 83(7):1094–1107, 2010.

[132] Jules White, Jose A. Galindo, Tripti Saxena, Brian Dougherty, David Benavides, and Douglas C. Schmidt. Evolving feature model configurations in software product lines. *Journal of Systems and Software*, 87:119–136, 2014.

[133] Saurabh Zunke and Veronica D'Souza. JSON vs. XML: A comparative performance analysis of data exchange formats. *International Journal of Computer Science and Network*, 3:257–261, 2014.

# VITA

Hazim Husain Shatnawi is a Ph.D. candidate in computer science at the University of Mississippi. He completed his Bachelor of Computer Science and Information Systems at Jordan University of Science and Technology in May 2006, being named to the Honor Roll during the 2005-2006 academic year. He finished his M.S. in computer science at the University of Mississippi in May 2013. He joined the National Center for Computational Hydroscience and Engineering (NCCHE) in September 2015 and is currently employed full-time as a Senior System Analyst.

Shatnawi's research interests are in software architecture and engineering, software product lines, feature modeling, relational and NoSQL database design, and Web development for scientific, engineering, and geospatial applications.