# Encoding Feature Models Using Mainstream JSON Technologies

Hazim Shatnawi
University of Mississippi
University, Mississippi, USA
hhshatna@go.olemiss.edu

H. Conrad Cunningham
University of Mississippi
University, Mississippi, USA
hcc@cs.olemiss.edu

## ABSTRACT

Feature modeling is a process for identifying the common and variable parts of a software product line and recording them in a tree-structured feature model. However, feature models can be difficult for mainstream developers to specify and maintain because most tools rely on specialized theories, notations, or technologies. To address this issue, we propose a design that uses mainstream JSON-related technologies to encode and manipulate feature models and then uses the models to generate Web forms for product configuration. This JSON-based design can form part of a comprehensive, interactive environment that enables mainstream developers to specify, store, update, and exchange feature models and use them to configure members of product families.

## CCS CONCEPTS

• **Software and its engineering → Software product lines**.

## KEYWORDS

Software Engineering, Software Product Line, Software Reuse, Feature, Feature Model, Feature Diagram, JSON, MongoDB

**ACM Reference Format:**
Hazim Shatnawi and H. Conrad Cunningham. 2021. Encoding Feature Models Using Mainstream JSON Technologies. In *2021 ACM Southeast Conference (ACMSE 2021), April 15–17, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3409334.3452048

## 1 INTRODUCTION

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* [3, 13, 18].

An SPL should enable and encourage software reuse within its application domain. 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. [18] introduced Feature-Oriented Domain Analysis (FODA). 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 (as described in Section 2).

Various researchers have extended the feature modeling method to better enable it to handle large, complex software families [11, 13, 14, 19]. to support manipulation and analysis using automated tools [6] and to guide developers in constructing valid feature models [1, 9, 10]. However, effective use of these methods and tools often requires specialized knowledge and skills. Mainstream developers thus may find it difficult to systematically build, modify, and reason about feature models and to build valid products from them.

Other researchers leverage the Extensible Markup Language (XML) technologies to specify feature models and configure products from SPLs [12, 15]. XML is "a metalanguage for creating markup languages" [25]. In these approaches, a feature model is an XML document that conforms to an XML Schema. However, a possible problem with XML is that "the XML Schema language has a number of type system constructs which simply do not exist in commonly used . . . programming languages" [20]. There is no standard way to handle these unsupported types, which "leads to interoperability problems" across platforms.

To mitigate the various concerns, we explore a novel approach that encodes feature models using JavaScript Object Notation (JSON) [5, 17]. Why use JSON instead of XML?

- 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 [17, 26]. 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.

The primary contributions of this research include:

- *How to accurately encode feature models in JSON.* Section 3 describes the syntax and semantics of our JSON-encoded feature models.
- *How to translate valid RDB-encoded feature models to and from JSON.* Shatnawi and Cunningham [30, 31] encode feature

*ACMSE 2021, April 15–17, 2021, Virtual Event, USA*
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8068-3/21/04. . . $15.00
https://doi.org/10.1145/3409334.3452048

models in relational database (RDB) tables. Section 4 specifies algorithms to translate their RDB encoding to and from our JSON encoding.
- *How to ensure validity of JSON-encoded feature models while creating, modifying, and deleting features.* Section 5 presents algorithms for creating, modifying, and deleting features. Our approach aims to separate the feature concept from its implementation by using the JSON notation.
- *How to store JSON-encoded feature models in MongoDB databases and preserve their validity while creating, modifying, deleting, and extracting information about features.* Section 6 presents our approach to using document-oriented MongoDB databases [4, 23] to store and manipulate valid JSON-encoded feature models. The section also describes how to generate Web forms (similar to that of Shatnawi and Cunningham [31, 31]) that enable users to specify valid products.

We conclude this paper by examining additional related work in Section 7 and then summarizing our contributions and outlining possible future work in Section 8.

## 2 FEATURE MODELS

A *feature model* is a compact representation of an SPL that captures all the design choices in one high-level description [18]. Each feature corresponds to a single design choice. A *common feature* is shared by all products in the SPL. A *variable feature* only appears in specific products. When configuring a product in the SPL, a common feature must be included; a variable feature may or may not be included based on the user's choice. The feature model should enable these variabilities to be managed effectively [2].

A feature model is a tree-like structure with a single root (called the *concept*) that represents the entire SPL [13, 18]. A node represents a feature and an edge represents the relationship between two features. There are two kinds of relationships [2, 3, 13, 18]:

- Parent-child relationships, each of which represents the relationship between a high-level (parent) design choice and a more detailed (child) design choice
- Cross-tree inclusion and exclusion constraints

Figure 1 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 [36] in Python 3.8.

The relationships between a parent and its children include:

- **Mandatory** features, which are child design choices that must appear in the final product whenever their parent features also appear. A mandatory feature is labeled by a black circle on top of the node in the feature model [13, 18]. In Figure 1, *Library* and *GDAL* are mandatory features.
- **Optional** features, which are child design choices that may or may not appear in the final product based on the user selections. An optional feature is labeled by a white circle on top of the node in the feature model [13, 18]. In Figure 1, *OGR*, *Polygonize*, and *PNG* are optional features.
- **OR** (one-to-many) features, which are sets of child features from which one or more members are selected. The selected features appear in the final product if their parent node and all other ancestor features up to the root are selected. An OR group is labeled by a black-filled arc or line that joins the OR features' edges. It is an extension to FODA's basic feature models proposed by Czarnecki [13]. In Figure 1, *CustomBlock* and *NativeBlock* form an OR group.
- **Alternative** (exactly one) features, which are sets of child features from which exactly one must be selected. This feature appears in the final product if its parent and all other ancestor features up to the root are selected. An alternative group is labeled by an arc or a line that joins the alternative features' edges [13]. In Figure 1, *DeleteEmptyPolygons*, *AreaCalc*, *Dissolve*, and *AddFields* form an alternative group.

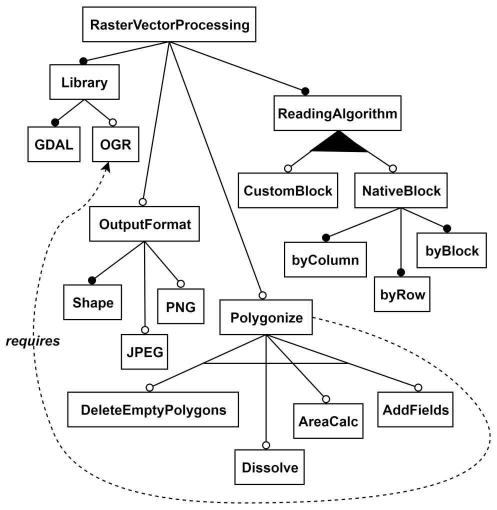


**Figure 1: Feature Model for Raster/Vector Manipulation SPL**

The cross-tree constraints enable features to **require** or **exclude** other features in the feature model [13, 18]. These constraints are represented by dotted edges. In Figure 1, *Polygonize* **requires** *OGR*, so the edge points to *OGR*. Therefore, if *Polygonize* is selected, then *OGR* must also be included. The converse does not hold. In **excludes** relationships, however, the converse does hold.

Figure 1 depicts an SPL with the product line concept *RasterVectorProcessing*. This feature indicates the purpose of the SPL. The figure shows a *Library* feature with two children. (1) The *GDAL* library, shown as a mandatory feature, is selected by default. (2) The *OGR* library, shown as an optional feature, can either be selected or not selected to be in a product.

The *RasterVectorProcessing* 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 [27] array syntax. For larger rasters, the SPL offers the *ReadingAlgorithm* feature for reading raster files. This feature offers two mechanisms in an OR relationship, enabling the user to select one or both. (1) The *CustomBlock* feature is 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. (2) The *NativeBlock* feature uses whatever the raster's reading

```
{
    "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 2: Example of a JSON-encoded Feature Model**

mechanism to read column-by-column, row-by-row, or using the native block size retrieved from the raster band.

The SPL offers a *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 add to the shape file such as deleting empty polygons, dissolve polygons to be merged into one shape, calculate the polygons' area, and add fields to the outputted raster files (e.g. id, description). These operation under the *Polygonize* feature require the OGR library for accessing and manipulating vector shape files.

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

## 3 ENCODING FEATURE MODELS IN JSON

This section presents this paper's first contribution: how to accurately encode feature models in JSON. Figure 2 shows part of the feature model presented in Figure 1.

This JSON-based language [17] 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.

We represent a feature as a JSON object [17] 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
- `relation`, which is either the string `OR` or `alternative`
- `requires`, which is an array of feature names
- `excludes`, which is an array of feature names
- `children`, which is an array of feature objects

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.

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.

## 4 TRANSLATING FEATURE MODELS

Shatnawi and Cunningham [30, 31] describe an approach to feature modeling with similar goals to the work we present here. It uses a mainstream relational database (RDB) to represent a feature model as a directed acyclic graph. For the purposes of this paper, the design consists of three tables. (1) The **feature** table defines the set of features, representing each feature by a unique `id`. (2) 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. (3) The **relationships** table lists the static set of possible relationships between features. Shatnawi and Cunningham's approach also generates a dynamic, Web-based user interface that enables users to construct and modify valid models and to configure valid products from them.

Our research seeks to provide a JSON encoding for feature models that can also serve as an exchange and archival mechanism for Shatnawi and Cunningham's RDB-encoded feature models. This section presents this paper's second contribution: how to translate valid RDB-encoded feature 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 3 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 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.

```
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 3: RDB-to-JSON Feature Model Translator**

```
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 4: JSON-to-RDB Feature Model Translator**

If we assume that a JSON document correctly encodes a valid feature model (e.g., is an output of the RDB-to-JSON translator above), 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 4, the algorithm traverses the JSON-encoded tree and gathers information about the tree's nodes and edges that it subsequently uses to populate the **feature** and **featuresRelations** tables [30]. The **relationships** table is a static table that is the same for all feature models.

```
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 5: Operation to Create a Feature**

If we call the DECODE procedure 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.

## 5 MANIPULATING JSON MODELS

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

Figure 5 shows the operation to *create* a new feature and add it to the JSON-encoded feature model. The operation's inputs are the properties of a feature object as described in Section 3. The operation first verifies that the feature's name is unique among

```
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: Algorithm to Get a Feature's Descendants**

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 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 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 5) merges the arrays returned by the *getAscendants* and *getDescendants* algorithms and then passes the result to a third algorithm (shown in Figure 7) 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

```
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 7: Algorithm to Enforce Cross-tree Constraints**

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. We also created a Web form similar to the one proposed in [31] for enabling the creation, modification, and deletion of features through the user interface.

## 6 USING MONGODB DATABASES

So far, we have defined how to encode feature models as JSON documents and discussed how to manipulate them without giving much attention to how they are stored. In this section, we present this paper's fourth contribution: how to store JSON-encoded feature models in MongoDB databases and preserve their validity while creating, modifying, deleting, and extracting information about features. We also generate Web forms that enable users to specify valid products.

Since the 1970s, relational databases have been the most prominent approach to organizing large data collections [34]. As we have noted, Shatnawi and Cunningham [30] 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 [21].

In this section, we explore the use of a NoSQL database to store feature models. In particular, we investigate the document-oriented MongoDB databases [4, 23] to store the JSON-encoded models and its query language to manipulate the models. We choose a MongoDB database because it is organized around JSON-like documents. To support this investigation, we install both the *MongoDB server* and the *Compass Community* GUI. We use PHP's *MongoDB Driver Manager class* [33] to connect to a MongoDB database.

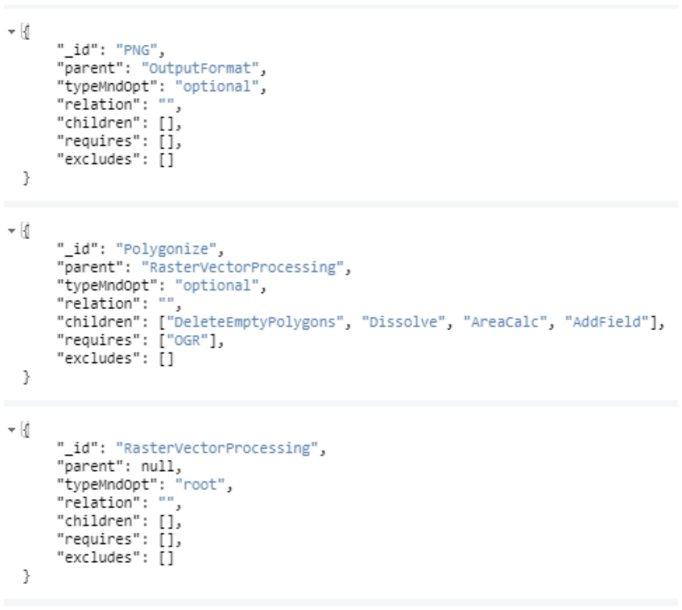
In our JSON-encoded feature model (e.g., Figure 3), a feature is a JSON object with zero or more other distinct JSON objects embedded within its `children` array. The representation in a MongoDB database separates each object representing a feature and stores it in a separate document. Therefore, the MongoDB representation of the feature model depicted in Figure 1 consists of 19 documents linked to each other via the defined relationships (`parent`, `children`, `requires`, or `excludes`). Using the MongoDB Query Language (MQL) enables us to create new features, delete or modify existing features, and extract information from the model.

If we construct a valid JSON-encoded feature model (e.g., indirectly using the RDB-to-JSON translator or directly using the creation algorithm), then we can store it in a MongoDB database using the Model Tree Structures with Parent References (MTSPR) pattern [23]. An MTSPR is a data model that organizes documents in a tree-like structure by storing references to the *parent* nodes in the *child* nodes [22].

Figure 8 shows the resulting feature model stored in a MongoDB database. The top feature in the feature model (i.e., the concept node) is `RasterVectorProcessing`. Like the other features, it has `_id`, `parent`, `relation`, `requires`, `excludes`, and `children` similar to the properties in the JSON-encoded models presented in Section 5. The primary difference is that the `type` property (which is a reserved word in MongoDB) is renamed `typeMndOpt`.

For the MongoDB-encoded feature models, we developed operations for creating new features and modifying and deleting existing features. These are similar to the ones described in Section 5 except that they create and manipulate the MTSPR representation used in the MongoDB database. In addition, we developed an algorithm that traverses the MongoDB-encoded feature model, determines the relationships and constraints between features, and generates a dynamic Web form (similar to that of Shatnawi and Cunningham [30, 31]) that enables a user to configure valid products from the SPL.

The algorithm first connects to the MongoDB server with the default `hostname` and `port` provided by the MongoDB Compass application and then determines which database name and collection holds the feature model. Next, it uses the MQL query `{parent:` `"root"}` to fetch the unique feature whose `parent` property has the value `"root"`. The root feature's `_id` property is the root's name.

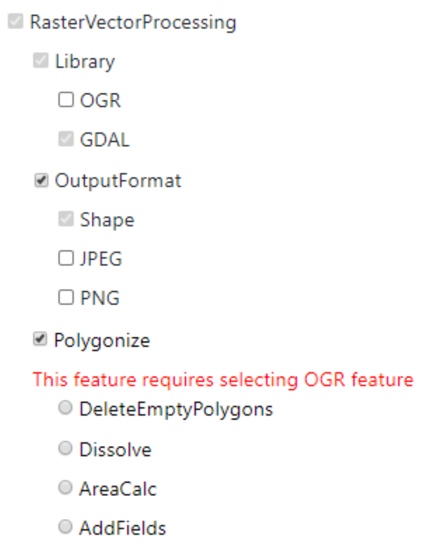


```
{
    "_id": "PNG",
    "parent": "OutputFormat",
    "typeMndOpt": "optional",
    "relation": "",
    "children": [],
    "requires": [],
    "excludes": []
}

{
    "_id": "Polygonize",
    "parent": "RasterVectorProcessing",
    "typeMndOpt": "optional",
    "relation": "",
    "children": ["DeleteEmptyPolygons", "Dissolve", "AreaCalc", "AddField"],
    "requires": ["OGR"],
    "excludes": []
}

{
    "_id": "RasterVectorProcessing",
    "parent": null,
    "typeMndOpt": "root",
    "relation": "",
    "children": [],
    "requires": [],
    "excludes": []
}
```

**Figure 8: Feature Model for Raster/Vector Manipulation SPL Stored in a MongoDB Database**



☑ RasterVectorProcessing
  ☑ Library
    ☐ OGR
    ☑ GDAL
  ☑ OutputFormat
    ☑ Shape
    ☐ JPEG
    ☐ PNG
  ☑ Polygonize
  This feature requires selecting OGR feature
    ○ DeleteEmptyPolygons
    ○ Dissolve
    ○ AreaCalc
    ○ AddFields

**Figure 9: Part of the Product Configuration Form**

In the next step, the algorithm generates an HTML form like the one shown in Figure 9. The algorithm represents the feature model's hierarchical structure as an HTML directory list structure. The algorithm builds the form by calling a recursive display function that takes the current feature's `name` (the root by default), its `parent`, its `relationship` with its parent, and the current `indentation` level. The display function fetches the value of the feature's `children` property and then recursively calls itself for each child. It stops the recursion when a feature has no children. The HTML form represents `optional` features and features within an `alternative` group as check boxes and features within an `OR` group as radio buttons. It represents `mandatory` features as radio buttons which are selected by default and cannot be modified. The form shows the cross-tree `requires` and `excludes` relationships, which cannot be shown directly in the hierarchy, as warning messages under the determined features. When the user selects a feature, the form updates the display to show which features have been selected and which are still available to be selected according to the semantics of the feature model.

Figure 9 shows part of the Web form generated from the SPL depicted in Figure 1. The directory list displays check boxes for the optional feature `OutputFormat` and the OR relationship among `ReadingAlgorithm`'s children and radio buttons for the alternative relationship among `Polygonize`'s children. The mandatory feature `ReadingAlgorithm` shows as a radio button locked into the selected state. The form indicates the cross-tree constraint that `Polygonize` requires `OGR` as a warning message under the determined feature.

The proof-of-concept implementation stores a JSON-encoded feature model in a MongoDB database. To do so, the implementation must transform the JSON model by breaking it into a set of JSON feature objects structured according to the MTSPR pattern. It connects to the MongoDB database using Python 3.8 and the package `pymongo` [24]. The implementation generates the Web form by printing the needed HTML statements into a text file.

## 7 DISCUSSION

The work reported in this paper focuses on how to use mainstream technologies to construct and maintain syntactically and semantically correct feature models for SPLs. In particular, we explore how to use the ubiquitous JSON notation and technologies to encode feature models. In this section, we compare our approach to others found in the literature.

Since Kang et al. [18] introduced the FODA method in 1990, researchers and practitioners have used the feature modelling method extensively to model SPLs. Over the years, researchers have proposed several variations to the method [11, 13, 14, 19]. However, to use these methods effectively, mainstream developers may need to acquire specialized knowledge and skills. They may find it difficult to use these methods in building, modifying, and reasoning about feature models. In our work, we seek to use mainstream technologies familiar to most developers.

To support manipulation and analysis using automated tools, some researchers express feature models using formal specifications. For example, Batory et al. [7] 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 [6] 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 discussed in Section 2. Checking the validity of a product configuration is thus a matter of evaluating the corresponding propositional formula. If the resulting value is *true*, the configuration is valid; otherwise, it is not.

We agree that the use of grammars and propositional formulas can help users understand feature models. Mainstream 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. In comparison, our approach uses everyday programming technologies such as databases, Web forms, and JSON. In addition, our interactive approach helps users discover problems during model construction, albeit at the cost of running frequent validity checks.

Other feature modeling research (e.g., pure::variants [10], FAMA [9], and FeatureIDE [1]) seeks to guide developers in constructing valid feature models. However, these require considerable expertise to use effectively, and they do not help users to configure products.

Cechticky et al. [12] and Ge and Whitehead [15] propose approaches to feature modeling and SPL configuration that use the Extensible Markup Language (XML) technologies. XML is "a metalanguage for creating markup languages" [25]. To design a language in the XML family, a designer must choose a specific set of names for the language's elements and attributes. In these approaches, a feature model is thus an XML document that conforms to an XML Schema.

According to Lanthaler [20], a problem with XML is that "the XML Schema language has a number of type system constructs which simply do not exist in commonly used . . . programming languages". There is no standard way to handle these unsupported types, which "leads to interoperability problems" across platforms. In comparison, our work seeks to leverage the easier-to-understand JSON technologies.

The research reported in this paper is similar in approach to that of Shatnawi and Cunningham [30, 31], which also seeks to use mainstream technologies such as RDBs and Web forms. The current research uses mainstream JSON technologies and seeks to be compatible with their RDB and user interface designs.

## 8 CONCLUSION AND FUTURE WORK

Feature modelling provides a simple representation for software product lines. This paper presents the following novel approaches:

- *How to accurately encode feature models in JSON.* Section 3 defines the syntax and semantics of a custom JSON language to represent "traditional" feature models. A JSON-encoded feature model is equivalent to the conceptual feature model depicted by a feature diagram.
- *How to translate valid RDB-encoded feature models to and from JSON.* Section 4 specifies algorithms to translate valid feature models in Shatnawi and Cunningham's [30, 31] RDB encoding to and from our JSON-encoded models. The translations preserve the validity of the models.
- *How to ensure validity of JSON-encoded feature models while creating, modifying, and deleting features.* Section 5 presents algorithms for creating, modifying, and deleting features in JSON-encoded feature models. These algorithms preserve the validity of the feature model as it is updated.

- *How to store JSON-encoded feature models in MongoDB databases and preserve their validity while creating, modifying, deleting, and extracting information about features.* Section 6 presents our approach to using document-oriented MongoDB databases [4, 23] to store and manipulate valid JSON-encoded feature models. These algorithms preserve the validity of the feature model as it is updated. The section also describes how to generate Web forms (similar to that of Shatnawi and Cunningham [31, 31]) that enable users to specify valid products.

To better understand the semantic issues that underlie this work, we are designing memory-based data structures and algorithms for feature models. We plan two prototype implementations that will emphasize safety, generality, and efficiency. We plan for one implementation to use the functional language Haskell and the other to use object-oriented Python 3.8+ with *mypy* type checking [32]. We plan for both to be able to read and write valid JSON-encoded feature models.

In the JSON-to-RDB translator, we assume that the input JSON-encoded feature model is syntactically and semantically correct. We plan to relax this assumption in future work. We plan to define an appropriate JSON Schema [16, 28] to be able to validate much of the model using standard JSON validators (e.g., Ajv [29]). 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 design a custom semantic validator.

In this work, we investigate the use of MongoDB databases [4, 23] to store feature models. In future work, we plan to investigate the use of graph-based Neo4j databases [8, 35] as a storage mechanism. This will give us the opportunity to systematically compare and evaluate the performance of several different feature model encodings based on various objective and subjective criteria [34].

## ACKNOWLEDGMENTS

We thank the anonymous referees and editors for their corrections and helpful suggestions. We also thank the Department of Computer and Information Science and (the first author's employer) the National Center for Computational Hydroscience and Engineering (NCCHE).

## REFERENCES

[1] M. Alam, A. Khan, and A. Zafar. 2018. Implementing Variability in SPL Using FeatureIDE: A Case Study. In *Proceedings of the International Conference on Electrical, Electronics, Computers, Communication, Mechanical and Computing (EECCMC).* IEEE Madras Section, Tamil Nadu, India, 584–593.

[2] S. Apel, D. Batory, C. Kästner, and G. Saake. 2013. *Feature-oriented Software Product Lines: Concepts and Implementation.* Springer, Berlin, Germany.

[3] P. Arcaini, A. Gargantini, and M. Radavelli. 2019. Achieving Change Requirements of Feature Models by an Evolutionary Approach. *Journal of Systems and Software* 150 (2019), 64–76.

[4] K. Banker, P. Bakkum, S. Verch, and D. Garrett. 2016. *MongoDB in Action* (second ed.). Manning, Shelter Island, NY, USA.

[5] L. Bassett. 2015. *Introduction to JavaScript Object Notation: A To-the-point Guide to JSON.* O'Reilly Media, Sebastopol, CA, USA.

[6] D. Batory. 2005. Feature Models, Grammars, and Propositional Formulas. In *International Conference on Software Product Lines.* Springer, Rennes, France, 7–20.

[7] D. Batory, R. Lopez-Herrejon, and J. Martin. 2002. Generating Product-lines of Product-families. In *17th IEEE International Conference on Automated Software Engineering.* IEEE, Edinburgh, UK, 81–92.

[8] D. Bechberger and J. Perryman. 2020. *Graph Databases in Action.* Manning, Shelter Island, NY, USA.

[9] D. Benavides, S. Segura, P. Trinidad, and A. Cortés. 2007. 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).* VaMoS, Limerick, Ireland, 129–134.

[10] D. Beuche. 2016. Using pure::variants across the Product Line Lifecycle. In *Proceedings of the 20th International Systems and Software Product Line Conference.* ACM, Montreal, QC, Canada, 333–336.

[11] J. Carbonnel, D. Delahaye, M. Huchard, and N. Clémentine. 2019. Graph-based Variability Modelling: Towards a Classification of Existing Formalisms. In *Proceedings of the International Conference on Conceptual Structures (ICCS).* Springer, Marburg, Germany, 27–41.

[12] V. Cechticky, A. Pasetti, O. Rohlik, and W. Schaufelberger. 2004. XML-based Feature Modelling. In *International Conference on Software Reuse*, Vol. 3107. Springer, Madrid, Spain, 101–114.

[13] K. Czarnecki and U. Eisenecker. 1999. *Generative Programming: Methods, Techniques, and Applications.* Addison-Wesley, Boston, MA, USA.

[14] K. Czarnecki, S. Helsen, and U. Eisenecker. 2005. Staged Configuration Through Specialization and Multilevel Configuration of Feature Models. *Software Process: Improvement and Practice* 10, 2 (April 2005), 143–169.

[15] G. Ge and E. Whitehead. 2008. Rhizome: A Feature Modeling and Generation Platform. In *Proceedings of the International Conference on Automated Software Engineering.* IEEE, L'Aquila, Italy, 375–378.

[16] JSON Schema Organisation 2021. *JSON Schema.* JSON Schema Organisation. http://json-schema.org

[17] JSON.org 2021. *Introducing JSON.* JSON.org. https://www.json.org/json-en.html

[18] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study.* Technical Report CMU/SEI-90-TR-21. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA.

[19] K. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh. 1998. FORM: A Feature-Oriented Reuse Method with Domain-specific Reference Architectures. *Annals of Software Engineering* 5, 1 (1998), 143–168.

[20] M. Lanthaler and C. Gütl. 2012. On Using JSON-LD to Create Evolvable RESTful Services. In *WS-REST '12: Proceedings of the Third International Workshop on RESTful Design.* ACM, Lyon, France, 25–32.

[21] A. Meier and M. Kaufmann. 2019. *SQL & NoSQL Databases: Models, Languages, Consistency Options and Architectures for Big Data Management.* Springer, Berlin.

[22] MongoDB, Inc. 2021. *Model Tree Structures with Parent References.* MongoDB, Inc. https://docs.mongodb.com/manual/tutorial/model-tree-structures-with-parent-references/

[23] MongoDB, Inc. 2021. *The MongoDB 4.4 Manual.* MongoDB, Inc. https://docs.mongodb.com/manual/

[24] MongoDB, Inc. 2021. *PyMongo 3.11.2 Documentation.* MongoDB, Inc. https://pymongo.readthedocs.io/en/stable/

[25] M. Murata, D. Lee, M. Mani, and K. Kawaguchi. 2005. Taxonomy of XML Schema Languages Using Formal Language Theory. *ACM Transactions on Internet Technologies* 5, 4 (November 2005), 660–704.

[26] N. Nurseitov, M. Paulson, R. Reynolds, and C. Izurieta. 2009. Comparison of JSON and XML Data Interchange Formats: A Case Study. In *22nd International Conference on Computer Applications in Industry and Engineering (CAINE 2009).* International Society for Computers and Their Applications, San Francisco, CA, USA, 157–162.

[27] T. Oliphant. 2006. *A Guide to NumPy.* Vol. 1. Trelgol Publishing, USA.

[28] F. Pezoa, J. Reutter, F. Suarez, M. Ugarte, and D. Vrgoč. 2016. Foundations of JSON Schema. In *Proceedings of the 25th International Conference on World Wide Web.* International World Wide Web Conferences Steering Committee, Montreal, QC, Canada, 263–273.

[29] E. Poberezkin. 2021. *Ajv: Another JSON Schema Validator.* https://ajv.js.org.

[30] H. Shatnawi and H. Cunningham. 2017. Mapping SPL Feature Models to a Relational Database. In *Proceedings of the ACM SouthEast Conference.* ACM, Kennesaw, GA, USA, 42–49.

[31] H. Shatnawi and H. Cunningham. 2020. Automated Analysis and Construction of Feature Models in a Relational Database Using Web Forms. In *Proceedings of the ACM SouthEast Conference.* ACM, Tampa, FL, USA, 323–338.

[32] The Mypy Project 2021. *mypy: Optional Static Typing for Python.* The Mypy Project. http://mypy-lang.org/index.html

[33] The PHP Group 2021. *The MongoDB Driver Manager class.* The PHP Group. https://www.php.net/manual/en/class.mongodb-driver-manager.php

[34] C. Vicknair, M. Macias, Z. Zhao, X. Nan, Y. Chen, and D. Wilkins. 2010. A Comparison of a Graph Database and a Relational Database: A Data Provenance Perspective. In *Proceedings of the ACM SouthEast Conference.* ACM, Oxford, MS, USA, 1–6.

[35] A. Vukotic, N. Watt, T. Abedrabbo, D. Fox, and J. Partner. 2014. *Neo4j in Action.* Manning, Shelter Island, NY, USA.

[36] F. Warmerdam, E. Rouault, et al. 2021. *GDAL/OGR Geospatial Data Abstraction Software Library.* Open Source Geospatial Foundation. https://gdal.org/