Automated Analysis and Construction of Feature Models in a Relational Database Using Web Forms

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

The Feature-Oriented Domain Analysis (FODA) approach introduced the feature model abstraction to represent software product lines. However, these 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, we propose a design that uses mainstream Web technologies to enable users to construct syntactically and semantically correct feature models and mainstream relational database technologies to encode the models as directed acyclic graphs. 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.

CCS CONCEPTS

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

KEYWORDS

Software product line, feature model, relational database, feature model construction, automated analysis

ACM Reference Format:
Hazim Shatnawi and H. Conrad Cunningham. 2020. Automated Analysis and Construction of Feature Models in a Relational Database Using Web Forms. In *2020 ACM Southeast Conference (ACMSE 2020), April 2–4, 2020, Tampa, FL, USA*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3374135.3385312

1 INTRODUCTION

The feature modelling technique was first introduced in the Feature-Oriented Domain Analysis (FODA) method [14]. Since its introduction, several extensions have been proposed that enable feature models to represent larger systems with more complex relationships between features than FODA allowed [8, 15, 22]. Several researchers have proposed various ways to represent feature models, such as by using domain specific languages (DSLs) [1], formal methods [26], and propositional formulas for configuration using existing logic-based tools [5].

However, 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. There have been a few efforts to tackle these issues [7, 11, 19], 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.

We address this need by proposing a novel design based on mainstream Web technologies. Our design uses a dynamic Web interface with two parts:

- a Web form enabling the creation, modification, and deletion of features and the definition of relationships and constraints among the features
- a live-preview page showing the constructed feature model represented as a directory structure

This Web-based user interface for product creation and configuration extends our previous work [21], which presents a novel approach to specification of feature models: encoding them in a relational database (RDB). 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 [23]. A significant benefit is the ability to use the well-known database query language (SQL) for reasoning about feature models. In this paper, 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.

The remainder of the paper is structured as follows. Section 2 reviews the feature model concept. Section 3 presents our novel approach to building feature models and configuring products using dynamic Web forms. Section 4 discusses our work in relation to existing work and Section 5 summarizes our contributions and outlines possible future research.

*Also with National Center for Computational Hydroscience and Engineering.

ACMSE 2020, April 2–4, 2020, Tampa, FL, USA

ACM ISBN 978-1-4503-7105-6/20/03.
https://doi.org/10.1145/3374135.3385312

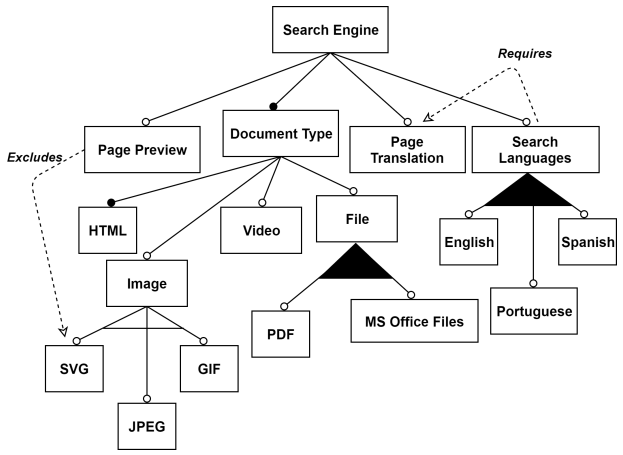

Figure 1: Feature Model for a Search Engine SPL

2 FEATURE MODELS

In the literature, the *feature model* is the most prominent technique for representing a software product line (SPL) or family of related systems [6, 10, 20]. Each choice in how to structure an individual product is called a *feature*. A feature model aggregates all these choices into one high-level descriptive model of the SPL. Some features are common (i.e. those shared among all products) and thus must always be included. Other features are variable and thus can be selectively included or excluded from from a product.

A feature model is a tree-like structure with a single root representing the entire SPL. An edge links a parent feature with a child. The parent represents a high-level design choice while the child represents a more detailed choice. There may also be cross-tree inclusion or exclusion constraints between features.

A feature model can be depicted using a *feature diagram*. Figure 1 shows a feature diagram for a simplified *Search Engine* product line [17, 21]. As described in detail in a previous paper [21], a child has one of four relationships with its parent: *Mandatory* (child must be selected), *Optional* (child can be selected), *Or* group (one or more in the group must be selected), and *Alternative* group (exactly one in the group must be selected). A filled circle at the end of an edge marks that child as *Mandatory* (e.g. *HTML* in Figure 1) and an unfilled circle marks it as *Optional* (e.g. *Video*). A filled arc marks an *Or* group (e.g. under *Search Language*) and an unfilled arc marks an *Alternative* group (e.g. under *Image*). To simplify a model, we restrict a parent to having one group of *Alternative* and *OR* children.

The *Optional*, *Alternative*, and *OR* features are considered variabilities or product choices. By calculating the possible number of each variability's presence, analysts and developers can count the number of different products generated from the SPL.

Cross-tree constraints between features are represented by dotted edges. In Figure 1, feature *Search Language requires* feature *Page Translation*, which is indicated by the arrow pointing to *Page Translation*. Therefore, if feature *Search Language* is selected in the configured product, then feature *Page Translation* must also be included, regardless of whether it is an optional feature. The opposite direction does not hold. The node to which the arrow points is the feature that determines the choice. The *requires* relationship cannot constrain ancestor or descendent features.

Figure 1 also shows that feature *Page Preview excludes* feature *SVG*, which is indicated by the arrow pointing to *SVG*. This means that regardless of the arrow direction, if one feature is present, the other cannot be. Syntactically, our design treats the *exclude* relationship as unidirectional. However, semantically, we treat it as bidirectional. An *exclude* feature cannot constrain ancestor or descendent features. Cross-tree constraints do not follow the hierarchical parent-child tree structure since they can relate features with two different parents, which does not follow the feature model's hierarchy rules.

3 FEATURE MODEL INTERFACE

In our previous work [21], 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 tables. As the tables are populated, care must taken to ensure that the model stays syntactically and semantically correct.

In this paper, we extend the previous work by focusing on how to construct correct feature models systematically and use them effectively. We design a dynamic, Web-based user interface with two phases. The first phase gathers the feature model data interactively, verifies its syntactic and semantic correctness, builds the database, and enables the on-going evolution of the model. The second phase enables the user to interactively configure valid products from the feature model. We want to enable mainstream users to construct, modify, and use feature models without having to learn specialized theories, notations, or technologies.

In the literature, we know of little work with similar goals. The first feature model editor supporting abstract features is FeatureIDE [2]. To use FeatureIDE effectively, developers and users must familiarize themselves with the Feature-Oriented Programming [3] and Aspect-Oriented Programming [16] paradigms.

Our work represents feature models in a generic way. It is not bound to any programming language or mathematical model. It can be used with any general or domain-specific programming language. Our research targets development of SPLs on the Web, where several programming languages can be deployed to generate a product. In our approach, features can be client-side (i.e., JavaScript, HTML, CSS) or server-side (i.e., PHP, Python) software assets.

3.1 Feature Creation

Figure 2 shows the user interface's Feature Creation tab. It consists of a Web form with components requiring entry of the new feature's name, its parent, the type of its relationship with its parent, and its requires and excludes relationships with other features.

The Feature Name component uses an HTML input field to get the information about a 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 be unique within the model. The system ensures the uniqueness 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 previously created features.

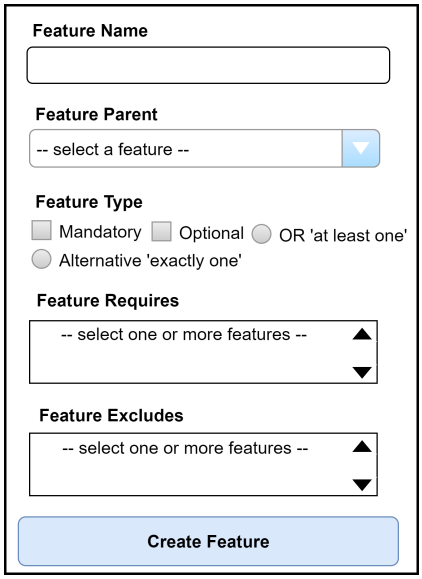

Figure 2: Feature Creation Tab

The Feature Parent component uses an HTML select tag to associate a feature with its parent. This tag displays a drop-down list from which the user selects the parent feature. This part works as a decision-choice that affects the information supported in the Feature Type component's form (described below). The system 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 system identifies 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's component are activated.

The Feature Type (Relationships) component uses an HTML list with two radio buttons and two checkboxes to assign the relationship between the newly inserted feature and its parent. 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), the system identifies 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* one 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 selecting-multiple-options list to define *Requires* and *Excludes* between features. The user can select 0-N features in both fields (requires and excludes). *Requires* and *Excludes* relationships must obey the following rules:

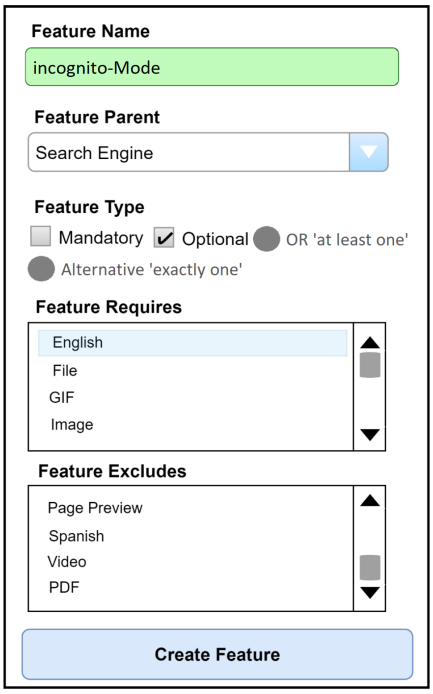

Figure 3: Creating a New Feature *incognito-Mode*

- 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 include an ancestor. The system checks this by recursively constructing the path from a feature to the root (i.e. concept feature) of the feature model.
- A feature cannot require or exclude a sibling. The system performs this check in order to remove the siblings from the lists of possible components 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.

Figures 2 and 3 illustrate how a user can add feature *Incognito-Mode* to the feature model using the Web interface. The system guides the user to construct a syntactically and semantically correct (i.e. valid) feature model, going through the following steps:

- The user enters the new feature's name *Incognito-Mode*. The system checks to ensure that name is not already defined.
- The system lists all available parent features including the root. The user selects the new feature's parent from the list.

- Once the user selects the feature's parent, the system checks that parent's relationships with its children. If the relationship is *Mandatory* or *Optional*, 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 3, since the user selects *Search Engine* as the parent feature, its relationship with children is either *Mandatory* or *Optional*. Therefore, the system deactivates the *Alternative* and *OR* relationships and activates the *Mandatory* and *Optional*. Although they are represented as checkboxes, the system ensures that the user does not select both *Mandatory* and *Optional*. If one is selected, the other is deactivated.
- After the user identifies the new feature's parent, the system 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. The system 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.
- Once the user identifies which features are *required* by the new feature, the system 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. The system 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 shows an algorithm to validate these cross-tree constraints.

Although the system checks for mutual exclusivity, it does not prevent the user from making incorrect decisions in some cases. An example would be a feature which is required by one feature and excluded by a different feature. This issue can arise during the product configuration phase, when the user selects the features from the feature model to include in a particular product. When this issue occurs, the system notifies the user in order to modify the relations again during the feature model's construction phase.

3.2 Feature Modification and Deletion

The user interface's Feature Modification tab enables a previously defined feature to be changed. It displays a radio button for each feature defined in the SPL. When the user selects a radio button, the system fetches the information about the corresponding feature and populates the Feature Creation form accordingly. The user can then modify the feature's name, parent, type, and cross-constraints relationships as needed. The same validation rules applied to these values during feature creation apply during feature modification.

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 displaying a radio button for each feature in the SPL. The system allows any feature to be deleted, even a mandatory feature.

Algorithm: 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 components* in the web form and handle their selections

1 Define arrays used in the algorithm (allFeaturesArr, ..etc) **if** *parent exists in the feature model* **then**
2 **if** *parent is select 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: Requires/Excludes Algorithm

If the deleted feature has no children, the system just deletes the feature and updates the feature model to reflect the change.

If the deleted feature has children, the system 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 system asks the user whether to delete the entire SPL.

3.3 Product Configuration

The second phase of the user interface consists of the Product Configuration interface. It enables the user to configure a product from the family of possible products defined by the feature model. This phase is a dynamic enhancement of a phase in our previous work [21]. As the feature model is being defined, the system interprets the syntax and semantics of the stored model and displays a hierarchical list. Whenever the user changes the feature model, the system dynamically updates this "live preview" of the model.

The Product Configuration interface enables the user to select a set of features for a complete product that satisfies the feature model's constraints [21]. It indicates *Optional* features and features within an *Alternative* group using checkboxes and indicates features within an *OR* group with radio buttons. *Mandatory* features are represented as radio buttons which are selected by default and cannot be modified. The system indicates 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 system updates the display to show what features have been selected and which are still available to be selected according to the semantics of the feature model. When a feature involved in a *Requires* or *Excludes* relationship is selected, then the related feature is selected or deselected as required by the semantics. When product configuration is complete, the user clicks the *Submit* button. If no feature has been selected, the system displays a warning message.

As future work, we plan a third phase of system that will generate a product from the selected configuration of a feature model. This process will not be tied to any specific programming language. As much as feasible, the system design will enable use of a wide range of programming languages, general-purpose or domain-specific.

4 DISCUSSION

In the literature, we find considerable work on the representation of feature models. Van Deursen and Klint [24] propose the Feature Description Language (FDL), a textual language to describe features that can be mapped to UML diagrams. Cechticky et al. [9] and Ge and Whitehead [12] propose XML-based approaches to feature modeling and configuration. White et al. [25] 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 non-developers. Users also need to be familiar with complex topics such as propositional formulas and constraint satisfaction or need to learn a new programming language in order to work with feature models. Our system 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 [13] 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 [19], staged configuration [11], FAMA [7], and FeatureIDE [2] 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. Our system checks the validity of the feature model at every step of its creation, modification, and use.

5 CONCLUSION AND FUTURE WORK

In 1990, Kang et al. [14] introduced feature modeling as a part of the Feature-Oriented Domain Analysis (FODA) approach. For the past three decades, feature models have been regarded as useful abstractions for representing the common and variable parts of software product lines. Various researchers have put forward a number of extensions, specification languages, and tools for working with feature models. However, feature models are often difficult for mainstream developers to specify and maintain because most tools rely on specialized theories, notations, or technologies.

In our previous work [21], we addressed this issue by proposing a design that uses mainstream relational database technologies to encode feature models as directed acyclic graphs. This paper extends that work and proposes a design that uses mainstream Web technologies to construct feature models and store them in the database. To demonstrate our design, we developed a proof of concept system. The first phase of our system leverages the dynamic capabilities of Web forms to collect the needed information from the users and ensure the resulting feature model is both syntactically and semantically correct. The system interactively guides the user to construct a valid feature model. Using a similar Web form, the second phase of our system interactively guides the user to configure valid members of the product family represented by a stored feature model.

Thus, our Web interface design, combined with the relational database design, can form parts of a comprehensive environment that enables mainstream developers to specify, store, and update feature models and use them to configure and generate members of product families. Figure 5 shows our overall system workflow from user registration through feature model creation, product configuration, and final product generation. We are currently designing the product generation phase of the system.

We are exploring ways to make our system more modular, flexible, and extensible, while still relying on technologies commonly used by mainstream developers. One novel approach we are investigating is the addition of a specification language based on JavaScript Object Notation (JSON) [4] to provide an alternative encoding of feature models. We propose to define a JSON dialect and use a JSON Schema [18] to specify its syntax and (as much as possible) its semantics. This potentially allows the evolving JSON validation tools to handle most of the rules enforcement we described in Section 3. We also propose to design a programmatic interface to handle other aspects of the creation and manipulation of valid feature models encoded in our specification language.

This JSON-based specification language can serve as a precise medium for communication among loosely coupled modules. This language can allow parts of a system to work in isolation from other parts 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

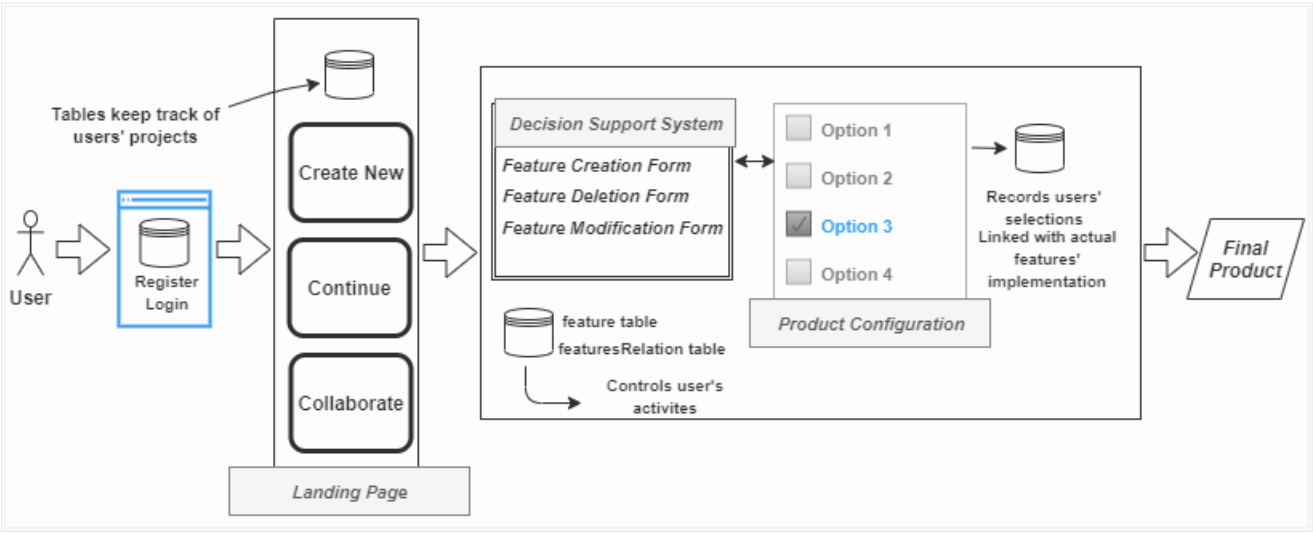

Figure 5: Overall System Workflow

format for archiving feature models. Although a realistic feature model description will be verbose, it can be read by human users as well as machines. We do not expect the specification language to supplant the use of the database, but the language potentially adds novel capabilities to our evolving feature modeling environment.

ACKNOWLEDGMENTS

We thank the anonymous referees for their helpful comments. We improved the paper in several ways as a result.

REFERENCES

[1] M. Acher, P. Collet, P. Lahire, and R. France. 2011. A Domain-Specific Language for Managing Feature Models. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*. ACM, Taichung, Taiwan, 1333—-1340.

[2] 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.

[3] S. Apel, D. Batory, C. Kästner, and G. Saake. 2013. *Feature-Oriented Software Product Lines*. Springer, Berlin.

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

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

[6] G. Bécan, M. Acher, B. Baudry, and S. Ben Nasr. 2016. Breathing Ontological Knowledge Into Feature Model Synthesis: An Empirical Study. *Empirical Software Engineering* 21, 4 (2016), 1794–1841.

[7] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-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. http://www.vamos-workshop.net/proceedings/VaMoS_2007_Proceedings.pdf.

[8] J. Bürdek, S. Lity, M. Lochau, M. Berens, U. Goltz, and A. Schürr. 2014. Staged Configuration of Dynamic Software Product Lines with Complex Binding Time Constraints. In *Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems*. ACM, Magdeburg, Germany, 16.

[9] 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.

[10] K. Czarnecki and U. Eisenecker. 2000. *Generative Programming: Methods, Techniques, and Applications*. Addison-Wesley Professional, Boston.

[11] K. Czarnecki, S. Helsen, and U. Eisenecker. 2005. Staged Configuration through Specialization and Multilevel Configuration of Feature Models. *Software Process: Improvement and Practice* 10 (04 2005), 143–169.

[12] 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.

[13] S. Günther and S. Sunkle. 2012. rbFeatures: Feature-Oriented Programming with Ruby. *Science of Computer Programming* 77, 3 (2012), 152–173.

[14] 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.

[15] 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.

[16] H. Kurdi. 2013. Review on Aspect Oriented Programming. *International Journal of Advanced Computer Science and Applications* 4, 9 (2013), 22–27.

[17] M. Mendonça. 2009. *Efficient Reasoning Techniques for Large Scale Feature Models*. Ph.D. Dissertation. University of Waterloo.

[18] 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, Canada, 263–273.

[19] pure-systems GmbH 2019. *pure::variants User's Guide*. pure-systems GmbH, Magdeburg, Germany. https://www.pure-systems.com/fileadmin/downloads/pure-variants/doc/pv-user-manual.pdf.

[20] S. Sepúlveda, A. Cravero, and C. Cachero. 2016. Requirements Modelling Languages for Software Product Lines: A Systematic Literature Review. *Information and Software Technology* 69 (2016), 16–36.

[21] 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.

[22] A. Sree-Kumar, E. Planas, and R. Clarisó. 2016. Analysis of Feature Models Using Alloy: A Survey. In *Proceedings of the 7th International Workshop on Formal Methods and Analysis in Software Product Line Engineering (FMSPLE'16)*. EPTCS, Eindhoven, The Netherlands, 45–60.

[23] T. Thüm, C. Kästner, S. Erdweg, and N. Siegmund. 2011. Abstract Features in Feature Modeling. In *Proceedings of the 15th International Software Product Line Conference SPLC 2011*. IEEE, Munich, Germany, 191–200.

[24] A. Van Deursen and P. Klint. 2002. Domain-Specific Language Design Requires Feature Descriptions. *Journal of Computing and Information Technology* 10, 1 (2002), 1–17.

[25] J. White, D. Benavides, D. Schmidt, P. Trinidad, B Dougherty, and A. Ruiz-Cortés. 2010. Automated Diagnosis of Feature Model Configurations. *Journal of Systems and Software* 83, 7 (2010), 1094–1107.

[26] J. White, J. Galindo, T. Saxena, B. Dougherty, D. Benavides, and D. Schmidt. 2014. Evolving Feature Model Configurations in Software Product Lines. *Journal of Systems and Software* 87 (2014), 119–136.