An OWL- Based Approach for Integration in Collaborative Feature Modelling

Lamia Abo Zaid[1], Geert-Jan Houben[2], Olga De Troyer[1], and Frederic Kleinermann[1]

[1] Vrije Universiteit Brussel (VUB)
Pleinlaan 2, 1050 Brussel
Belgium
{Lamia.Abo.Zaid, Olga.DeTroyer, Frederic.Kleinermann}@vub.ac.be, http://wise.vub.ac.be/
[2] Delft University of Technology (TU Delft)
Mekelweg 4, 2628 CD Delft
the Netherlands
g.j.p.m.houben@tudelft.nl, http://www.wis.ewi.tudelft.nl

Abstract. Feature models are models that are used to capture differences and commonalities between software features, thus enabling the representation of variability within software. As the number of features grows, along with the increasing number of relations between features, the need rises to have collaboration between designers and have separate feature models together representing one system. Integration of such distributed models becomes an error-prone task. The large number of features and the often complex relations between features calls for the automated support of collaborative feature modelling. In this paper we present an OWL-based approach for the representation of feature models, while adding formal semantics to bring together distributed feature models used in collaborative modelling. We also provide a framework to detect anomalies and conflicting feature relations in a resulting integrated model.

Keywords: Feature model, OWL, knowledge representation, interoperability.

1. Introduction

Today there is an urgent need in the software community for developing variable software. *Variable software* is known under names as software product line or software product family [1]. Variability in software is specified by defining a set of required *variant* tasks, i.e. functionalities that need to be implemented but can be implemented through different variants. This is usually done by summing up all the possible features that products could have. The concept of *feature* commonly represents an increment in program functionality. Feature models (also known as feature diagrams) are used to visually represent features and their relations [2, 3]. Different combinations of features thus make up the variation in products.

Feature models alone are not sufficient for variability. Applying the *divide and conquer* strategy, a software product is divided into components and different teams

or persons are involved in the development of the different components. The main complicating factor is that when dividing a system into a set of components, *dependencies* between their features exist due to constraints such as hardware/software limitations, security policy issues, and others. Typically, there are many relations between the features of a single software component. This complexity even rises with the many interactions, dependencies and conflicts that may exist between the features of different components. Many of these dependencies and relations are not easily captured by feature models, and with the number of features in today's complex systems jumping to a few thousand, feature models become very difficult to manage. For one system, multiple features models could exist to model the variability of the different parts of the system. This makes the *integration* of feature models in a distributed and collaborative system design process a complicated effort.

At the same time, to complicate practical application even further, there is no real agreed upon semantics for feature models [4]. Many variations to the original notation of FODA [2] exist such as FORM [5] and FeatuRSEB [6]; for a detailed study about these variations we refer the reader to [4]. Moreover, there are a number of extensions of FODA, such as to include cardinality [7] and feature constraints [8]. This apparent lack of a common semantics for feature models makes it difficult to exchange and share feature models in practical applications. As a consequence, tool support for feature model has become fragile, making transformations between feature models a problematic issue. We believe that providing a machine-processable feature model ontology will create a common base for generic feature model tool support.

In this paper we provide an OWL-based approach to *represent* and *manage* feature models. We have two main contributions. First, we provide an OWL ontology to *represent* and define feature models. Secondly, we present the semantics for feature model *integration*. We show that employing an ontology-based technique to represent feature models with well-formed semantics allows for better interoperability between different feature models. The integration semantics significantly helps the design process, as it provides the big picture of the overall system which is a vital element in any realistic design process for variable software.

The rest of this paper is organized as follows. In section 2 we give a brief introduction of feature model constructs. Section 3 discusses the need for feature model integration and its semantics. Next, in section 4 we present the semantics of our feature model ontology and show how to use reasoning to infer the relevant model consistency.

2. Feature Model Constructs

Feature models describe hierarchical structures of features. The hierarchies have exactly one root node and the links in the hierarchy show how features are constructed out of other features as their subfeatures. The feature model does not only show the feature composition hierarchy but also shows the nature of the compositions via the *relations* between features. Commonly, there are five types of relations possible in a feature model: *Alternative*, *Or*, *And*, *Mandatory*, and *Optional* [2, 3]. In

addition, additional *dependencies* between features may exist, often used as constraints.

To illustrate this, figure 1 shows the Order Process example, the running example in this paper. It shows three feature models representing the three segments of the order process problem: Order Process, Order Fulfilment, and Order Payment. The feature model shows for each feature its name and type. A feature that contributes to variability is called a *variable feature*. Accompanied with additional feature dependency constraints (see figure 1), a feature model gives information about the features that should be part of a valid software product. A valid composition of features is called a *configuration* [1, 3]: a valid composition of features results in a valid product, which is a product that meets all the type restrictions and feature dependencies.

The *segmentation* of the information about features and their relations could cause unjustified or contradictory decisions when constructing a product. A *feasible* feature model is one that is consistent, i.e. holds no contradictions. A model containing contradictions makes it difficult to find feasible feature compositions, thus reducing the number of valid products. Furthermore segmentation of functionality across different feature models may result in conflicts between the constraints of the different segments. Thus there is a need for feature model conflict detection in the integration of segments; we will discuss this in more details in section 3.

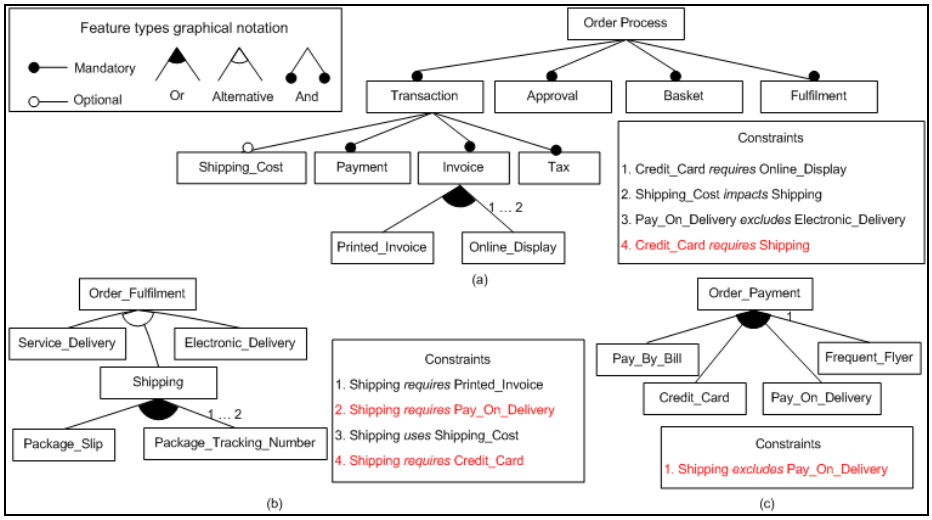

Fig. 1. Order Process Problem, modified after [9], with a) Order Process Segment, b) Order Fulfilment Segment, and c) Order Payment Segment

Current research in feature models is oriented towards finding feasible feature compositions that adhere to all of the relations and constraints defined. In [3] the authors attempt to use a Logic based Truth Maintainance System (LTMS) and Boolean Satisfiability Problem Solver (SAT solver) to propagate constraints. LTMS also provides automatic selections for a possible configuration, and provides justification for automatically selected/deselected features. In [10] feature models are

transformed into a Constraint Satisfaction Problem where a constraint solver is used to determine the feasible configurations of a feature model. In [11], the authors use Higher Order Logic (HOL) to formulate feature models: Prototype Verification System (PVS), a HOL solver, is then used to find feasible configurations. Although in these techniques configurations are automatically found, debugging in case of a design error is a hard task. Neglecting the fact that a contradiction in the model may be blocking feasible or expected feature combinations is a major drawback for such feature analysis techniques [12].

A different approach is presented in [13], where an OWL-based approach was used to represent and verify feature models. OWL constraints are used to model feature relations and constraints defined by the feature model. Given a certain feature configuration their approach can detect whether it is valid or not.

3. Feature Model Integration

Division of (the design of) large systems in terms of functionality comes quite natural. Often different decentralized teams are involved: this makes agreement on all features and their relations not an easy task. Thus the need to *integrate* separated features models is crucial for obtaining a correct global understanding of the system.

If we see the separate feature models as parts of the global puzzle, then for each part separately we could (assume to) guarantee the correctness or the consistency of the model. As an informal example, suppose we have four features *A*, *B*, *C*, *D*; in *model 1, A* is dependent on *B,* and *B* is dependent on *C*. In *model 2*, *C* excludes *A,* and *D* requires *B*. On their own, both *model 1* and *model 2* are consistent. While combining them in a global model, the model becomes inconsistent. This interaction of features in terms of dependencies can influence the selection of other features within a valid composition. We define these interactions as *constraints* between features. We have done a literature study in the field of feature modelling to identify *possible constraints* between features. We also investigated the current *limitations* of feature models and the current need to define constraints in languages like Object Constraint Language (OCL) or even simple English sentences. Furthermore we have looked at work that *extends* feature models by adding more terms and notations [7, 14]. From these studies, we have composed a list of constraints defining semantic relations between features. We call these constraints *feature to feature constraints (FTFC)*: Table 1 gives our feature to feature constraints and their meaning.

Table 1. Feature to Feature Constraints (dependencies)

FTFC name	**Meaning**
Excludes	Feature A excludes feature B means that A and B cannot occur together (XOR). *Ex.* “Maximum graphics” *excludes* “Maximum performance”.
Extends	Feature B extends feature A if B adds to the functionality of A. *Ex.* “Full registration” *extends* “Simple registration”.
Impacts	If feature A has an impact on feature B, it means that the existence of A affects the existence of B. This is typically used as a less rigid relation than the Requires relation. *Ex*. “Air conditioning” *impacts* “Horse power”.
Implies	If feature A implies feature B, it means that the existence of A indicates that B should also exist due to a functional need (use relation) or a logical need

	(ex. auxiliary features). Ex. “Advanced graphics” *implies* “High memory”
Includes	Indicates that feature A has feature B inside of it. *Ex.* “Add username" *includes* "Check user name exists”.
Incompatible	If feature A is incompatible with feature B, then A and B are mutual exclusive due to a conflict. It adds more semantics to the cause of exclusion than excludes, and is usually used for hardware/software dependencies. *Ex*. “Advanced graphics” *incompatible with* “Basic graphic controller”
Requires	Feature A requires feature B if A is functionally dependent on B. Ex. “Advanced editor” *requires* “Spelling checker”.
Uses	Feature A uses feature B then there is a dependency relation, so logically if A is required then B should also be required. *Ex*. “Search” *uses* “Provide hints”.
Same	Constraint used to indicate that two features are the same. *Ex*. “Advanced graphics” *same* “AG”

Back to our order process example of figure 1: on its own, each segment is consistent, but putting together the three segments there is a clear inconsistency between the features (marked in red in figure 1). Furthermore, constraints between the features represent semantic links for the integration, such as the *uses* relation between *shipping* and *shipping_cost*. Naturally, in connecting the segments there is also a need to indicate that features are semantically the same. As an example *fulfilment* in Figure 1.a is semantically the same feature as *fulfilment* as a root feature in Figure 1.b. By explicitly defining such links as part of the model it becomes possible to track features that depend on or influence other features in the overall integrated model.

4. Feature Models Represented in OWL

This section describes our OWL [15] (Web Ontology Language) based ontology for representing feature models. By definition, an ontology is a conceptualization of (a part of) the world. In this section we describe our conceptualization of feature models with extended semantics for *integration*. We chose OWL to represent our ontology. First, because it allows exchanging different feature models, driven by the standardized common, agreed upon semantics of the feature model representation. Second, OWL has formal semantics making it machine-processable which enhances feature modelling tool support, as it will remove the ambiguity in representations and provide a formal understanding of the underlying model. Finally, OWL (DL) was designed to support DL reasoning on top of the ontology model, which enables using DL reasoners to infer knowledge. Next, we will discuss the ontology in more details.

4.1 Feature Model Ontology Constructs

An ontology expresses knowledge of the world in terms of classes, properties and restrictions. Classes represent the real-world concepts or objects. We have chosen the iterative engineering approach described in [16] to model our Feature Model Ontology (FMO). In our ontology representation we model the feature model *constructs* as classes. Our intension is to express the feature model(s) *including* integration support: we represent the information of feature model constructs by

providing the vocabulary and structure to represent feature models in a descriptive way. Following a top-down approach to define the key constructs within the feature model representation, (figure 2 shows the ontology class hierarchy):

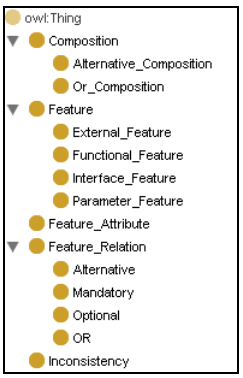

Fig. 2. FM Ontology Class Hierarchy

a) Feature Model Ontology Classes

- *Feature:* is the main ontology construct. Features could be of type: *external*, *functional*, *interface* or *parameter*.
- *Composition:* represents *Alternative/Or* relations in a feature model. *And* relations are normalized to *mandatory* relations and thus are omitted.
- *Feature Attribute:* defines a variable associated with the feature; the value of the variable is specified during the composition of the product.
- *Feature Relation:* represents the *Mandatory*, *OR, Optional*, or *Alternative* types for a feature..
- *Inconsistency:* is a class that captures inconsistent features: features belonging to the Inconsistency class will be assigned during the reasoning phase.

b) Feature Model Ontology Properties: We represent the integration semantics defined in section 3 (Table 1) as sub-properties of the *Feature_to_Feature_Constraint* property, which has Feature class as both domain and range. Furthermore, *Incompatible* and *Excludes* are defined as symmetric properties. *Extends*, *Requires* and *Includes* are defined as transitive properties. Furthermore, for the sake of logical consistency of the model some properties are mutual exclusive. In addition, we define properties that help to model the hierarchal structure of feature models.

4.2 Ontology Implementation Issues

We implemented our Feature Model Ontology using Protégé OWL [17], Pellet [18] as a DL reasoner, SWRL [19] to represent rules, and Jess [20] as a rule engine.

a) Specifying the Feature Model Ontology Consistency

Ontology consistency is often used to refer to concept satisfiability. We used Pellet for checking the ontology consistency. In our case we also seek for *Variability Model Consistency* (i.e. logical consistency of the feature model), which can be enforced by defining rules that capture such inconsistencies (conflicts). When bringing together (integrating) fragmented feature models the aim is to obtain one model in which we could easily identify such inconsistencies. To support the detection of inconsistencies in our ontology, we have defined a class named *Inconsistency*; all instances causing a logical inconsistency will be given membership to this class. The cases which cause a logical inconsistency are represented by a set of SWRL rules: a rule has an antecedent defining an inconsistent situation and a consequent that marks the individuals causing this inconsistent situation. Marking is done by asserting them to have a *problem* relation between them. *Problem* is a property of the *Inconsistency* class. We specify the set of rules that assign inconsistent

individuals to the Inconsistency class via the problem property. We capture two types of inconsistency problems: first, those that emerge from using two properties that are mutually exclusive for the same features (ex. b.2, c.1 in figure 1), and second, those that detect a two-way direction of using a certain property which is defined to be asymmetric (ex. a.4, b.4 in figure 1).

b) Reasoning on the Integrated Model

Coming back to our running example (figure 1); we populate the Feature Model Ontology with instances representing the Order Process example. Each feature in the problem is represented as an instance of the Feature class. Relations are represented as instances of the Composition class.

We use Jess to run the SWRL rules: the rules are transferred to Jess along with the ontology and the rule engine evaluates these rules against the ontology population; we refer the reader to [21] for more information. As a result, Jess will associate the features that have inconsistencies with the *problem* property, namely *Shipping*, *Credit_Card*, and *Pay_on_Delivery*. We then run Pellet to check the ontology consistency and compute the inferred types (new assertions were made by firing the rules in Jess). Pellet infers that features having a *problem* relation are members of the *Inconsistency* class. In the example the Inconsistency class has 3 inferred individuals.

This example shows how our approach allows integrating distributed feature models by means of specifying the points of integration and using rules to check the variability model consistency. The combination of the rule engine's ability to run conflict detection rules with the reasoner's ability to infer new types enables detecting inconsistencies that follow from *implicit* (hidden) relations between the features.

5. Conclusion

Although OWL was initially proposed for the semantic web, its expressive power and formal semantics made it usable in many other domains. This paper demonstrates the use of OWL for creating an ontology for feature models, adding feature-based integration semantics to the integration of segmented feature models.

As opposed to the work in [13], our target was to enable creating one model from collaboratively obtained segmented feature models. For doing this, there was a need for introducing an ontology for feature model representation, that will formally represent feature model *semantics*. For the purpose of bringing together collaborative feature models we enriched our ontology with formal semantics to specify the *integration* between features. When bringing together fragmented feature models there is a need for conflict detection between different feature models. We applied a rule-based approach, to capture conflicts between features of the integrated model.

For our future work towards a complete framework to model and manage feature models, we aim to further enrich our ontology by considering even more use cases than the ones done until now. We also need to provide explanations to users on why a certain inference is made by the reasoner. Currently Pellet provides some support for such debugging possibilities, but in a very non-user-friendly format. As a second stage, an innovative user interface to query the features model ontology is required to allow users to query about features and their relations within the ontology.

References

1. Asikainen, T.: Modelling Methods for Managing Variability of Configurable Software Product Families. Licentiate thesis. Helsinki University of Technology, Department of Computer Science and Engineering (2004)
2. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, A.: Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie-Mellon University (1990)
3. Batory, D.: Feature models, grammars, and propositional formulas. In: Obbink, H., Pohl, K. (eds.) SPLC 2005. LNCS, vol. 3714 (2005)
4. Bontemps, Y., Heymans, P., Schobbens, P.-Y., Trigaux, J.-C.: Semantics of Feature Diagrams. In: Workshop on Software Variability Management for Product Derivation Towards Tool Support (2004)
5. Kang, K., Kim, S., Lee, J., Kim, K., Shin, E., Huh, M.: FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures. In: J. Annals of Software Engineering. vol. 5, pp. 143-168 (1998)
6. Griss, M., Favaro, J., d'Alessandro, M.: Integrating Feature Modeling with the RSEB. In: Fifth International Conference on Software Reuse, pages 76–85 (1998)
7. Czarnecki, K., Kim, C. H. P.: Cardinality-Based Feature Modeling and Constraints: A Progress Report. In: OOPSLA'05 International Workshop on Software Factories (2005)
8. Lopez-Herrejon, R.E., Batory, D.: A Standard Problem for Evaluating Product-Line Methodologies. In: Bosch, J. (ed.) GCSE 2001. LNCS, vol. 2186, pp. 9–13. (2001)
9. Ye, H.; Liu, H.: Approach to modelling feature variability and dependencies in software product lines. In: Software, IEE Proceedings -Volume 152, Issue 3, Page(s): 101 – 109, (2005)
10. Benavides, D., Trinidad, P., Ruiz-Cortés, A.: Automated Reasoning on Feature Models. In: 17th Conference on Advanced Information Systems Engineering (CAiSE'05)
11. Mikoláš, J., Kiniry, J.: Reasoning about Feature Models in Higher-Order Logic. In: 11th International Software Product Lines Conference (SPLC 2007).
12. Batory, D., Benavides, D. , Ruiz-Cortés, A.: Automated Analyses of Feature Models: Challenges Ahead. In: Communications of the ACM (Special Section on Software Product Lines) (2006)
13. Wang, H., Li, Y., Sun, J., Zhang, H., Pan, J.: A semantic web approach to feature modeling and verification. In: Workshop on Semantic Web Enabled Software Engineering (SWESE'05) (2005)
14. Sinnema, M., Deelstra, S., Nijhuis, J., Bosch, J.: Managing Variability in Software Product Families. In: Proceedings of the 2nd Groningen Workshop on Software Variability Management (SVMG 2004)
15. OWL Web Ontology Language Overview, http://www.w3.org/TR/owl-features/
16. Noy, N. F., McGuinness, D. L.: Ontology Development 101: A Guide to Creating Your First Ontology. Stanford Knowledge Systems Laboratory Technical Report KSL-01-05 and Stanford Medical Informatics Technical Report SMI-2001-0880 (2001)
17. Stanford Protégé OWL, http://protege.stanford.edu/overview/protege-owl.html
18. Pellet DL Reasoner, http://pellet.owldl.com/
19. Horrocks, I., Patel-Schneider, P. F., Boley, H., Tabet, S., Grosof, B., Dean, M.: SWRL: A Semantic Web Rule Language Combining OWL and RuleML, http://www.w3.org/Submission/SWRL
20. Jess Rule Engine, http://herzberg.ca.sandia.gov/
21. O'Connor, M. J., Knublauch, H., Tu, S. W., Musen, M. A.: Writing Rules for the Semantic Web Using SWRL and Jess. In: 8th International Protege Conference, Protege with Rules Workshop, Madrid, Spain (2005)