

Towards Context-Aware Feature Modelling using Ontologies

Dennis Wagelaar*

Vrije Universiteit Brussel, Pleinlaan 2, 1050 Brussels, Belgium
`dennis.wagelaar@vub.ac.be`

Abstract. Feature modelling is a well-known technique for modelling the commonalities and variabilities in a software product line. A feature model can describe feature interactions (dependencies and exclusions), such that one can derive which are valid feature configurations. There are cases in which feature interaction is caused by an external factor, e.g. the software libraries that are required by particular features cannot be used at the same time. If this external factor (or context) changes in the future, e.g. one library replaces both conflicting libraries, the features requiring the library functionality no longer exclude each other. The feature model needs to be updated each time such an external factor changes. We propose to separate feature interactions caused by external factors by explicitly modelling external dependencies. Separate context models are used to describe the relevant external factors that currently apply. When these models are put together, one can derive which features can be combined. If an external factor changes, only the context model describing that factor needs to be updated. The feature model remains the same. We propose to use ontologies and OWL-DL in particular as an underlying framework to express both the feature model and the context model. Automatic reasoners for description logics can be used to determine validity of feature configurations.

1 Introduction

In software product line development, the possible product configurations can be modelled using feature models [1]. Feature models can express simple constraints on what feature combinations are valid. One can specify the inclusion of a feature as mandatory or optional and one can specify sets of alternative features. In addition, one can specify simple feature dependencies, since the feature model is organised as a tree where sub-features require the inclusion of their super-features. Several feature model extensions [2] add expressiveness by allowing the modelling of multiple feature instances and their cardinality constraints, feature groups and group cardinalities, feature attributes and additional relationships, such as *consists-of* and *is-generalisation-of*. In [3], a feature modelling tool that

* The author's work is part of the CoDAMoS project, which is funded by the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT-Flanders)

uses XPath for describing advanced feature interactions is discussed and in [4] constraint programming is used to express advanced feature interactions.

While the expressiveness of feature models has been substantially improved, feature models still suffer from fragility caused by external factors (or context). If an external factor causes a particular feature interaction (dependency or exclusion), a change in this external factor may require a revision of the feature model. Consider a software family with a voice output feature and an audio cue feature, both intended for notifying the user. The voice output feature requires a text-to-speech engine to be installed on the computer (i.e. the text-to-speech engine is not considered a part of the software family). This text-to-speech engine currently accesses the sound hardware directly. The audio cue feature requires an audio mixing engine (again not part of the software family) that also accesses the audio hardware directly. As such, the two features cannot be used simultaneously. This is reflected in the feature model by modelling these features as mutually exclusive. If, later on, we replace the text-to-speech engine by another one that uses the audio mixing engine, text-to-speech can suddenly be used together with other audio. The voice output feature and the audio cue feature are no longer mutually exclusive, even though our software family remained the same.

We propose to separate the feature interactions caused by external factors from the internal feature interactions by explicitly modelling these external factors and dependencies. The external factors themselves are described in separate context models, while the feature model itself only describes the dependencies on these external factors. Ontologies [5], and OWL-DL [6] in particular, are used as an underlying framework to express both context model and feature model. Automatic reasoners for description logics, such as RACER [7], can be used to determine the validity of feature configurations for a given context.

2 A Context Ontology

In order to reason about context and context constraints, an ontology for describing context is used. Ontologies can serve as a common vocabulary for a domain. By using a shared model of context, we can reason about the relationship between a context description and a context constraint, even if the two do not have a direct relationship. An example context constraint is that the Java Swing framework needs to be present. An example of a context description includes a Java JDK 1.5. Since both the context constraint and the context description refer to the context ontology to explain what the Java Swing framework resp. the Java JDK 1.5 is, one can derive whether the Java JDK 1.5 satisfies the Java Swing framework constraint.

In this paper, the context ontology described in [8] is used as a common vocabulary¹. This ontology is in turn inspired by the User Agent Profile specification (UAPProf) [9] and Composite Capability/Preference Profiles (CC/PP) [10],

¹ Other ontologies can be used, but it is necessary to use the same ontology for describing context and constraints on that context.

both of which are standards intended to describe target platforms. The ontology is expressed in OWL-DL, a variant of OWL that corresponds to description logics [11], allowing for automated reasoning about the ontology. The part of the ontology that models platforms is shown in Fig. 1.

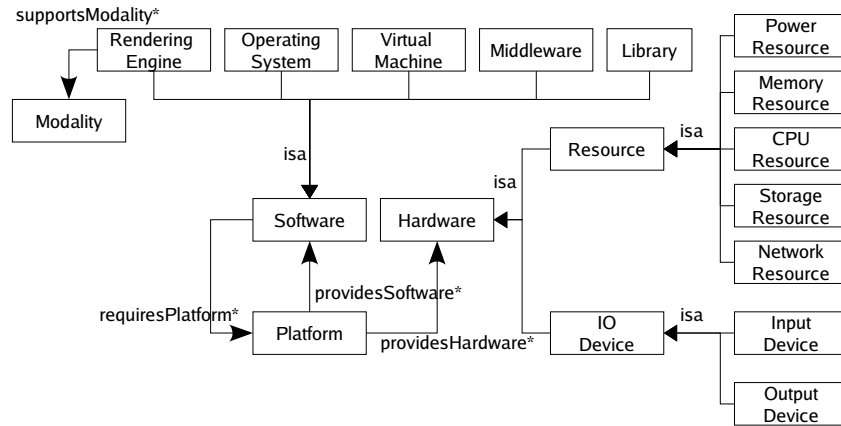


Fig. 1. Part of the context ontology for describing platforms

The platform concept in this ontology can provide software and hardware. A '*' next to a relationship denotes a one-to-many relationship. Software and hardware are broken down into different sub-concepts. This is denoted by the special "isa" subsumption relationship, e.g. the set of operating systems subsumes the set of software in general. The software can impose requirements on the platform, e.g. the need for a network resource, a particular virtual machine or a user interface rendering engine that supports voice communication. This is denoted by the "requiresPlatform" relationship, which points to a description of the required platform.

The ontology can be extended for particular domains of platforms, such as Java virtual machines. Fig. 2 shows such an ontology. The "VirtualMachine" concept starts with "context:" to indicate it refers to the "VirtualMachine" concept from the main context ontology. The "JavaVM" virtual machine can be subdivided in many different configurations. "J2SE" refers to the virtual machines that run Java 1.2 or up. "JavaVM" is split up into "J2ME", "JDK" and "PersonalJava". Next to virtual machines, other concepts are introduced as well, e.g. "AWT" and "Swing". These refer to the Java Abstract Window Toolkit (AWT) resp. Swing rendering engines. Since some Java virtual machine configurations already include these, instances of such virtual machines also serve as instances of the AWT and/or Swing rendering engine. This is represented in the ontology by defining additional "isa" relationships to the AWT or Swing rendering engine from the virtual machines.

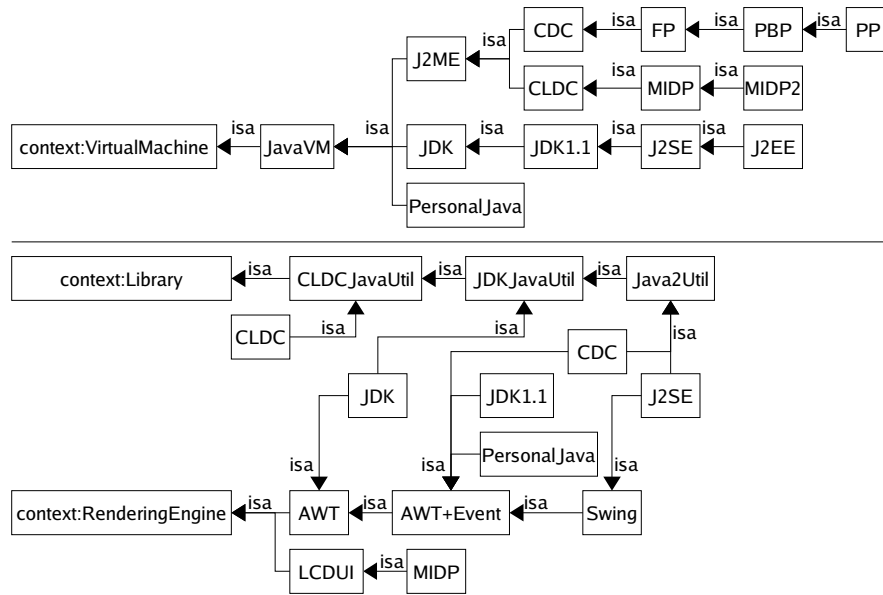


Fig. 2. An ontology describing Java virtual machines

More information on these ontologies and the description of a concrete context can be found in [12].

3 A Feature Ontology

If we want to use the given context ontology in a feature model, we need to map feature models to the domain of ontologies as well. An initial ontology has been developed to express feature models. This ontology is shown in Fig. 3.

There are two relationships defined for Feature: *impliesFeature* and *transitivelyImpliesFeature*, both of which point to other Features. The first relationship is used to model direct dependencies of a particular feature. The *impliesFeature* relationship is a sub-relationship of *transitivelyImpliesFeature*, i.e. if feature A implies feature B, it also transitively implies feature B. The *transitivelyImpliesFeature* relationship is a transitive relationship that is used to model all dependencies of a feature (direct and indirect).

The SoftwareFeature and HardwareFeature concepts are defined equivalent to the Software and Hardware concepts from the context ontology. This way, Software and Hardware instances from the context model can be treated as Features.

Finally, the Configuration concept is used to model product configurations. Two sub-concepts, ValidConfiguration and InvalidConfiguration are introduced

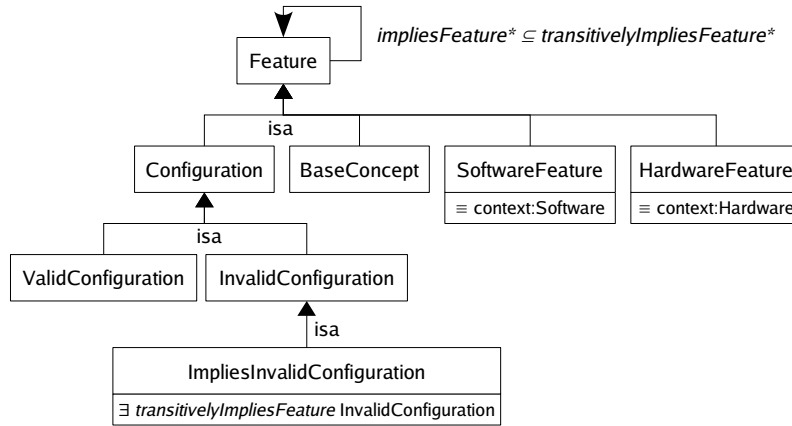


Fig. 3. An initial ontology for describing feature models

as a basis for the definition of conditions for validity and invalidity of configurations. An extra concept, *ImpliesInvalidConfiguration*, is used to classify each configuration that (transitively) implies an *InvalidConfiguration* as an *InvalidConfiguration* itself.

Each concrete feature is modelled as a concept with exactly one instance. Each abstract feature or feature group is modelled as only a concept. An example feature model is shown in Fig. 4.

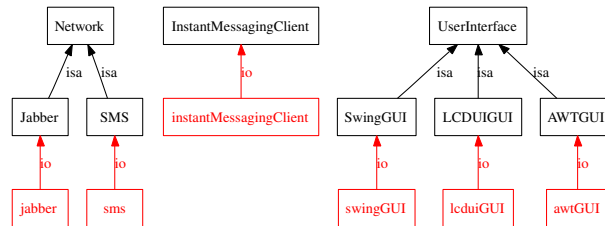


Fig. 4. An example feature model ontology

Only the concrete features have instances (their names starting in lowercase). The “*impliesFeature*” relationship can now be used to specify dependencies between the concrete features. One can then create sub-concepts for *ValidConfiguration* and *InvalidConfiguration* which define what the conditions are for a valid configuration and what makes a configuration invalid.

When defining the feature dependencies through the “*impliesFeature*” relationship, the conditions defined under *InvalidConfiguration* can be used by the reasoner to determine whether the feature model is still consistent (i.e. valid

configurations are possible). For example, the “lcduiGUI” feature should never be selected together with the “awtGUI” feature. This can be expressed in a sub-concept of InvalidConfiguration. If “awtGUI impliesFeature lcduiGUI”, then “awtGUI” can be classified as an InvalidConfiguration. Hence, any configuration that includes “awtGUI” is classified as an InvalidConfiguration through Implies-InvalidConfiguration. This is made possible by requiring each feature to imply itself (see the reflexiveness requirement in the previous section).

The given example presents features that can be included exactly once. One can also model more generative features that are applied multiple times (e.g. once for every class attribute). These features may even be applicable beyond the current software family. An example of such a feature model is shown in Fig. 5.

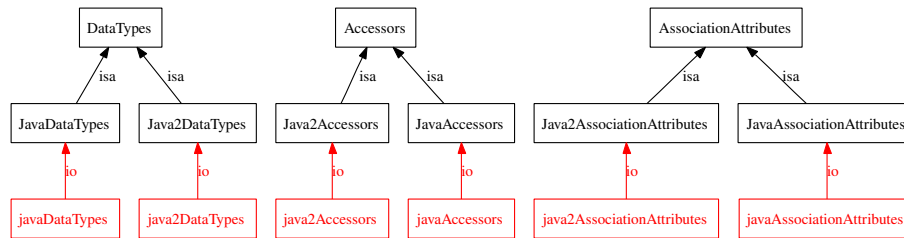


Fig. 5. An example feature model ontology of generative features

A ValidConfiguration for this feature model includes at least one feature from each of the root feature groups (DataTypes, Accessors and AssociationAttributes). The InvalidConfigurations include either both a JavaDataTypes and a Java2DataTypes or both a JavaAccessors and a Java2Accessors or both a JavaAssociationAttributes or a Java2AssociationAttributes.

As one may have noticed, each of these features has a Java-variant and a Java2-variant. If one chooses to include the Java-variant of a feature, one has to choose the Java-variants of the other features too, or the resulting software product will be inconsistent (e.g. having Java2 accessor methods while having Java attributes). Since ontologies support multiple classification, one can simply add an extra classification that models which features are a Java-variant and which are a Java2-variant. Such a classification is shown in Fig. 6.

One can now add an additional InvalidConfiguration definition that applies when both a CLDCJavaUtilFeature (i.e. the Java-variant) and a Java2UtilFeature (i.e. the Java2-variant) is chosen.

A similar multiple classification approach is also used by Batory et al. in Step-Wise Refinement [13]. They use an algebraic approach of multiple dimensions to classify their features. One can choose one value from each of the classifying dimensions, which will then reduce the amount of applicable features to a consistent sequence (features must be applied in-order).

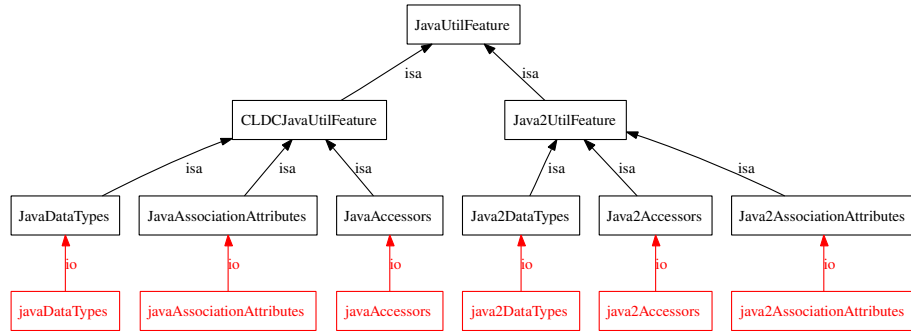


Fig. 6. An alternative classification of the example generative features

After a consistent feature model has been developed, concrete product configurations can be defined. These can be checked in the same way as the feature model consistency was checked: configurations are classified as `ValidConfiguration` or `InvalidConfiguration`. Only the configurations that can be proved valid, cannot be proved invalid and do not introduce inconsistencies in the model as a whole are the actual valid configurations.

4 Expressing External Feature Interactions

Now that both feature models and context models can be expressed using ontologies, we can define external feature interactions. Consider the example of “`awtGUI`” excluding the “`lcduiGUI`” feature. This has been modelled by defining a new `InvalidConfiguration` sub-concept that applies when both “`awtGUI`” and “`lcduiGUI`” are selected. This isn’t really correct, since it is perfectly possible to include multiple user interfaces in our application. The “`awtGUI`” and “`swingGUI`” can be combined, for example, and one of them will be chosen at run-time. The real reason that “`lcduiGUI`” cannot be combined is that there is no build environment that can compile a code base including both “`lcduiGUI`” and “`awtGUI`” at the same time. This is an example of an external factor, that has nothing to do with our application domain and may change in the future.

Using the ontologies discussed previously, we can define a context model that describes the available build environments. Two instances, “`jdk1.5.0`” and “`wirelessToolkit2.2`” are defined to describe the concrete build environments. “`jdk1.5.0`” is an instance of `JDK` and “`wirelessToolkit2.2`” is an instance of `MIDP2` (see Fig. 2). In addition, a new sub-concept of `InvalidConfiguration` (see Fig. 3) is defined:

$$\begin{aligned}
 \text{JDKandCLDC} &\sqsubseteq \text{feature} : \text{InvalidConfiguration} \\
 &\equiv \exists \text{feature} : \text{transitivelyImpliesFeature java} : \text{JDK} \\
 &\equiv \exists \text{feature} : \text{transitivelyImpliesFeature java} : \text{CLDC}
 \end{aligned}$$

Now consider the example feature model shown in Fig. 4. We can add “`jdk1.5.0`” to the “`impliesFeature`” property of “`awtGUI`”. Similarly, we add “`wirelessToolkit2.2`”

to “impliesFeature” for “lcduiGUI”. When building a Configuration that implies “awtGUI” or “lcduiGUI”, then “jdk1.5.0” or “wirelessToolkit2.2” are automatically included as well. When both are included at the same time, the configuration will automatically be classified as an InvalidConfiguration, since both “jdk1.5.0” and “wirelessToolkit2.2” are included.

If in the future the two build environments no longer conflict, then the JDK-andCLDC concept described earlier can be removed from the context model. Configurations that include both “awtGUI” and “lcduiGUI” are no longer classified as an InvalidConfiguration.

In general, the ability to describe the conditions for Valid- and Invalid-Configurations separately from the Features themselves, allows for externalising these conditions. If an externalised condition changes, only the model containing that condition needs to be updated.

5 Conclusion and Future Work

This paper has shown how feature models and external factors (or context) can be modelled using ontologies. A simple example of a feature model and some external factors has been given. While the feature model contains the actual features of the software product line, the feature interaction rules can be expressed separately in the context model. This allows the feature model to remain unchanged if any of the external feature interaction rules changes.

The ontology used for expressing feature models is still in an early state, however. In the given example, the feature instances (“awtGUI” and “lcduiGUI”) point directly to the external feature instances (“jdk1.5.0” and “wirelessToolkit2.2”). This is not entirely correct, since any JDK and MIDP instance would have sufficed for the given feature instances. As a consequence, if a new external feature instance is added, implementing both JDK and MIDP, the feature model does not leverage this. However, it is not possible in OWL-DL to have instances point to ontology classes. This would result in a higher-order logical expression.

A related issue is the general dependency of the feature ontology on the context ontology: the feature ontology refers to Software and Hardware from the context ontology for describing SoftwareFeature and HardwareFeature. This introduces the requirement that the context ontology must be usable for all feature models. The use of other contextual modelling approaches, such as the Deployment and Configuration specification of OMG [14], is not possible like this.

A possible solution may be to separate the feature model and the context dependencies into two reasoning spaces, which are both first-order. The feature instances may point to contextual instances. These contextual instances exist as classes in the context reasoning space. Like this, reasoning on the context model can be done beforehand and the results can be used when reasoning on the feature model. This process is known as stratification. Since the reasoning space for the context model is separated, it may have any form as long as there is

a mapping from the contextual instances in the feature model to the contextual classes in the context model.

In order to integrate an ontology representation of a feature model into current feature modelling tools, a mapping needs to be defined from each feature modelling language primitive to an ontological expression. Direct access to the ontology representation may remain necessary, however, to retain the full expressiveness of the ontology language. As a comparison, the Feature Modeling Plugin [3] uses XPath expressions to increase feature model expressiveness.

Acknowledgement

The author would like to thank the CoDAMoS project members and user committee for discussing their ideas for the Instant Messaging scenario, which were useful for the example used in this paper.

References

1. Kang, K., Cohen, S., Hess, J., Nowak, W., Peterson, S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report CMU/SEI-90-TR-021, Pittsburgh, PA, USA (1990)
2. Czarnecki, K., Helsen, S., Eisenecker, U.: Staged Configuration Using Feature Models. In Nord, R.L., ed.: Proceedings of the Third Software Product-Line Conference (SPLC 2004), LNCS 3154, Boston, MA, USA, Springer-Verlag (2004) 266–283
3. Antkiewicz, M., Czarnecki, K.: FeaturePlugin: feature modeling plug-in for Eclipse. In: Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange, Vancouver, Canada, ACM Press (2004) 67–72
4. Benavides, D., Trinidad, P., Ruiz-Cortés, A.: Automated Reasoning on Feature Models. In: Proceedings of the 17th Conference on Advanced Information System Engineering (CAiSE'05), Porto, Portugal (2004)
5. Gruber, T.R.: A Translation Approach to Portable Ontology Specifications. *Knowledge Acquisition* **5** (1993) 199–220
6. Smith, M.K., Welty, C., McGuinness, D.L.: OWL Web Ontology Language Guide. World Wide Web Consortium. (2004) W3C Recommendation 10 February 2004.
7. Möller, R., Haarslev, V.: Description Logics for the Semantic Web: Racer as a Basis for Building Agent Systems. *Künstliche Intelligenz* (2003) 10–15
8. Preuveneers, D., den Bergh, J.V., Wagelaar, D., Georges, A., Rigole, P., Clerckx, T., Berbers, Y., Coninx, K., Jonckers, V., Bosschere, K.D.: Towards an extensible context ontology for Ambient Intelligence. In: Proceedings of the Second European Symposium on Ambient Intelligence, Eindhoven, The Netherlands, Springer-Verlag (2004) 148–159
9. Open Mobile Alliance: User Agent Profile 2.0 Specification. (2003) Version 20-May-2003.
10. Klyne, G., Reynolds, F., Woodrow, C., Ohto, H., Hjelm, J., Butler, M.H., Tran, L.: Composite Capability/Preference Profiles (CC/PP): Structure and Vocabularies 1.0. World Wide Web Consortium. (2004)
11. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F.: *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, Cambridge, UK (2003)

12. Wagelaar, D.: Context-Driven Model Refinement. In Aßmann, U., ed.: Proceedings of Model Driven Architecture: Foundations and Applications (MDAFA 2004), Linköping, Sweden, Research Center for Integrational Software Engineering, Linköping University (2004) 107–121
13. Batory, D., Sarvela, J.N., Rauschmayer, A.: Scaling Step-Wise Refinement. IEEE Transactions on Software Engineering **30** (2004) 355–371
14. Object Management Group, Inc.: Deployment and Configuration of Component-based Distributed Applications. (2003) Draft Adopted Specification (ptc/03-07-02).