Features and Feature Interactions in Software Engineering using Logic

Ragnhild Van Der Straeten
rvdstrae@vub.ac.be
System and Software Engineering Lab
Vrije Universiteit Brussel, Belgium

Johan Brichau*
jbrichau@vub.ac.be
Programming Technology Lab
Vrije Universiteit Brussel, Belgium

Abstract

Feature interactions are common when composing a software unit out of several features. We report on two experimental approaches using logic to describe features and feature interactions. The first approach proposes description logic as a formalization of feature models which allow reasoning about features. In the second approach, a metalevel representation of the software is proposed to capture conditions on features. These conditions are written in terms of the software's implementation providing a uniform formalism that can be applied to any software unit.

1 Introduction

The concepts *feature* and *feature interaction* originated in the telephony-domain. In this context, a feature is an addition of functionality to the basic telephone system providing new behaviour. *Feature interaction* occurs when the behaviour of one feature influences the behaviour of another. When features interact in an unwanted way, a *feature interference* occurs [12].

A *feature* in software engineering can be seen as a concern of the software application [7]. The composition of features will always lead to interactions between features. A *feature interference* occurs when existing or new features interact such that a feature does not behave correctly. This paper discusses two experiments about feature interactions in software engineering.

In the first section we will describe our first approach, in which Description Logic is introduced to formally specify features in the problem domain.

*Research Assistant of the Fund for Scientific Research - Flanders (Belgium) (F.W.O.)

This logic and its reasoning mechanisms are already used to detect feature interactions in the telecommunication domain [1, 3, 4]. In our approach, however, we want to support feature descriptions and interaction detection for the *feature modeling* and *configuration* in domain engineering and in Generative Programming [6]. We want to start a discussion about which information should be described and which reasoning tasks a feature modeling tool should provide.

The next section will introduce our second approach, in which we use Logic Meta-Programming to detect feature interactions in the solution domain. We model features of a system using a logic metalevel representation of the system's implementation. Additional conditions and constraints about the implementation's structure can also be expressed using this metalevel structure. Adding new software artifacts (implementing a new feature) to the system will change the metalevel representation. A feature interference will lead to falsification of the conditions and constraints (imposed by the developer).

2 Description Logics for Feature Modeling

The general idea is to use Description Logic (DL) to formally specify features. As an experimental approach, we initiated the development of a language capturing feature modeling as it is used in Generative Programming [6]. Feature interaction is specified by the dependencies between the different features and the constraints applied on them.

2.1 Description Logics

The family of Description Logics originated from knowledge representation research in Artificial Intelligence. Their main strength comes from the different reasoning mechanisms they offer. The complexity of reasoning in these different languages is and has been widely investigated. These languages have been applied at an industrial level.

The basic elements of a Description Logic are *concepts* and *roles*. A *concept* denotes a set of individuals, a *role* denotes a binary relation between individuals. Arbitrary concepts and roles are formed starting from a set of atomic concepts and atomic roles applying concept and role constructors. For an introduction to Description Logics we refer to [2].

2.2 Feature Modeling using DL

We want to start a discussion about which information should be described and which reasoning tasks a feature modeling tool should provide. The latter is covered in the next section. Feature models appear in the Feature-Oriented Domain Analysis method (FODA) [8] and are used as such by Generative Programming [6]. In this context, a feature model consists of a feature diagram and additional information. This information consists of descriptions of each feature, rationales for each feature, stakeholders and client programs, examples of systems with a given feature, constraints, default dependency rules, availability and binding sites, binding modes, open/closed attributes and priorities. The aspect configuration in [9] can also be interpreted as the modeling (i.e. configuration) of features (i.e. aspects). This section introduces a basic feature language $\mathcal{FML}$ describing the semantics of features and some constraints. $\mathcal{FML}$ is based on the DL $\mathcal{ALCQ}$.

2.2.1 Syntax and Semantics of the Logic

In $\mathcal{ACLQ}$ concepts (denoted by C, D) are formed as follows:

$$C, D \longrightarrow A \mid \neg C \mid C \sqcap D \mid \exists R.C \mid (\leq 1R) \mid \exists^{\leq n} R.C \mid \exists^{\geq n} R.C$$

where A denotes an atomic concept, R denotes an atomic role and n denotes a strict positive integer. The following abbreviations are used: $\top$ for $A \sqcup \neg A$, $C \sqcup D$ for $\neg(\neg C \sqcap \neg D)$, $\forall R.C$ for $\neg \exists R.\neg C$, $\exists^{=n} R.C$ for $\exists^{\leq n} R.C \sqcap \exists^{\geq n} R.C$, $\exists R^{=n|\leq n|\geq n}$ for $\exists R^{=n|\leq n|\geq n}.\top$, $CxorD$ for $(C \sqcup D) \sqcap \neg(C \sqcap D)$. Descriptive semantics, defined by an interpretation function, are adopted, see [10]. A *knowledge base* $\mathcal{K}$ in $\mathcal{ALCQ}$ is a pair $< T, A >$ such that:

- T is the T(erminological)-Box, a finite, possibly empty set of expressions of the form $C_1 \sqsubseteq C_2$ where C_1, C_2 are concepts. This inclusion specifies that C_2 only gives necessary conditions for being an instance of C_1. $C_1 \doteq C_2$ is equivalent to $C_1 \sqsubseteq C_2$ and $C_2 \sqsubseteq C_1$. The formulas in the T-Box are called terminological axioms.

- A is the A(ssertional)-Box, a finite, possibly empty set of expressions of the form $a : C$ or $(a, b) : R$ where C is a concept, R is a role and a, b are individuals.

No restrictions are posed on the terminological axioms. This means that each atomic concept may appear more than once at the left side of an axiom. The terminological axioms may contain cycles, i.e. the concept in the right part of the axiom may refer to the concept in the left part of the axiom.

The *feature language* $\mathcal{FML}$ is completely based on the DL $\mathcal{ALCQ}$. We fix a signature $\Sigma =< Con, Rol, Ind >$, where Con is a countable set of atomic concepts, Rol is a countable set of atomic roles and Ind is a countable set of individuals. A $\mathcal{FML}$ feature model is a set of terminological axioms. The ABox is empty in $\mathcal{FML}$. A feature diagram can be translated to axioms of $\mathcal{FML}$. The set Con consists of all concepts corresponding to the nodes of the diagram. The set Rel consists of all the roles corresponding to the edges of the diagram. The edge decorations are translated using the concept constructors of $\mathcal{ALCQ}$. Consider as an example the following feature description of a car [6]. *A car consists of one transmission and one horsepower and optionally an airconditioning. The transmission is manual or automatic but cannot be both.* This feature model expressed in $\mathcal{FML}$ is shown in figure 1.

$$
\begin{array}{lll}
\texttt{TRANSMISSION} & \sqsubseteq & \exists^{=1}\texttt{man.MANUAL} \text{ xor } \exists^{=1}\texttt{aut.AUTOMATIC} \\
\texttt{CAR} & \sqsubseteq & \exists^{=1}\texttt{trans.TRANSMISSION} \sqcap \exists^{=1}\texttt{power.HORSEPOWER} \\
 & & \sqcap((\leq 1\texttt{airco}) \sqcap \forall\texttt{airco.AIRCONDITIONING}
\end{array}
$$

Figure 1: The Knowledge Base Corresponding to Features of a Car.

Cardinality constraints. Cardinality constraints can be expressed in the feature model language $\mathcal{FML}$. The constructors $\exists^{\leq n}$ and $\exists^{\geq n}$ admit these kinds of constraints.

If-then constraints. If-then constraints can be integrated in the concept definitions[1]. The constraint *"if there is airconditioning in a car then the horsepower of the car must be greater than or equal 100"*, can be written down as follows[2]:

$$
\begin{array}{lll}
\texttt{CAR} & \sqsubseteq & \exists^{=1}\texttt{trans.TRANSMISSION} \sqcap \exists^{=1}\texttt{power.HORSEPOWER} \sqcap ((\leq 1\texttt{airco}) \sqcap \\
 & & \forall\texttt{airco.AIRCONDITIONING} \sqcap (\neg\exists\texttt{airco} \sqcup (\geq_{100} \texttt{power}))
\end{array}
$$

This constraint involves the use of concrete domains and implies the integration of such a domain in the language $\mathcal{FML}$. The integration of concrete domains into DL has been described in [5]. Another constraint, naturally expressed in $\mathcal{FML}$ is the disjointness of features. The fact that the `MANUAL` and `AUTOMATIC` feature are disjoint can be expressed as $\texttt{MANUAL} \sqsubseteq \neg\texttt{AUTOMATIC}$.

[1] Note that in first order logic $p \to q$ is equivalent with $\neg p \vee q$.

[2] $\geq_{100}$ stands for the unary predicate $\{n; n \geq 100\}$of all strict positive integers greater or equal 100.

2.3 Reasoning Tasks in Feature Modeling

Tool support for feature models should at least contain support for the feature notation and the different dependencies and constraints. The use of DL to formalize feature models enables the execution of certain tasks that are now left to the developer. This section shows how standard reasoning tasks of DL can be used to accomplish certain tasks.

The standard reasoning tasks considered in DL at the terminological level are *subsumption, concept consistency* and *knowledge base consistency.* C_2 subsumes C_1 iff in all models of the knowledge base $\mathcal{K}$ the interpretation of C_1 is a subset of the interpretation of C_2. A concept C is consistent in $\mathcal{K}$ if $\mathcal{K}$ admits a model in which C has a non-empty interpretation. A knowledge base $\mathcal{K}$ is consistent if there exists a model for $\mathcal{K}$. $\mathcal{ALCQ}$ is EXPTIME-complete.

Feature model consistency A feature model is consistent if it is possible to implement a system obeying this model. Checking if a model is consistent corresponds to the verification of the feasibility to build a system.

Feature consistency A feature is consistent if it can be instantiated with respect to the feature model. A feature is inconsistent due to, e.g. overconstraining.

Feature subsumption A feature F_2 subsumes a feature F_1 if in all possible instantiations of the feature model the interpretation of F_1 is a subset of the interpretation of F_2. Subsumption gives rise to a classification of all the features appearing in a feature model. It also allows the deduction of properties of one feature from those of another one.

Feature and constraint addition The addition of a feature or constraint can lead to the replacement of some specific subformula within a terminological axiom. This boils down to the addition of a specification. This can be seen as a function θ mapping specifications to specifications. This function is analogous to the δ-function in [4]. The consistency of this addition w.r.t. a knowledge base $\mathcal{K}$ reduces to knowledge base consistency of $\theta(\mathcal{K})$.

In this approach, DL seems to be a natural way to express feature diagrams and some constraints. The inclusion of additional information of a feature model still needs further investigation. Also to which extend the connection between the **GR(K)** language, the multi-modal counterpart of $\mathcal{ALCQ}$ can be useful in this context and the idea of a description language being able to define notions involving self-reference [2].

3 Logic Meta-Programming for Feature Interaction Detection

In the development of a software application, the addition of particular software artifacts (components, aspects, objects, ...), implementing a certain feature, will introduce interactions with other software artifacts. In this approach we try to use a declarative (i.e. logic) metalevel representation of the feature's implementation to detect feature interaction and interference.

3.1 The Figure Editor Case

Consider a simple figure editor in which the user is able to draw points on the screen and interconnect them to form lines and polygons. The basic system's service only allows this functionality. Using an object-oriented language we implement this system according to figure 2. Afterwards, we want to add

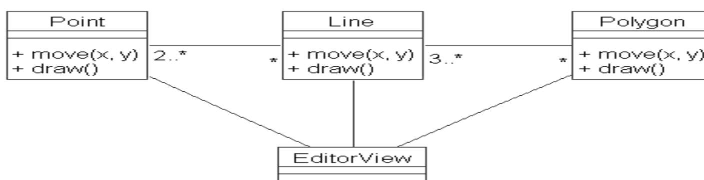

Figure 2: UML Diagram of the Figure Editor.

additional features to the simple figure editor. For instance, we add a feature implementing the archival of figures on a disk and later on, we add a *color* feature allowing to color the points, lines and polygons. After the introduction of the *color* feature, a feature interference can occur between the *archival* and the *color* feature because the original *archival* feature does not store the color of a point, line or polygon.

3.2 Logic Meta-Programming Approach

The Logic Meta-Programming (LMP) technique has an inate capability of declaratively capturing the structure of a program. The metalevel representation of a software application consists of logic facts. A possible representation of our figure editor example is:

```
class(Point).  class(Line).  class(Polygon).
method(Point,move,arguments(x,y),statements(...))
method(Point,draw,arguments([]),statements(...))
...
```

Using logic rules, we can derive a higher-level representation (i.e. towards the design-level). Using such rules, the LMP-technique has been extensively used to detect programming patterns, to trace the impact of changes in the implementation and to check conformance with the corresponding design and architectural description [11], [13].

We now augment the automatically generated metalevel representation of the software program with logic assertions classifying every software artifact in one or more features. For our figure editor example, this means:

```
feature(figures,[class(Point),class(Line),class(Polygon)])
feature(archival,[method(Point,store),method(Line,store),method(Polygon,store)])
feature(figuremovement,[method(Point,move),method(Line,move),method(Polygon,move)])
feature(UI,[class(EditorView),method(Point,draw),method(Line,draw),
         method(Polygon,draw)])
...
```

Because the features are now explicitly defined in terms of the software artifacts that implement them, we are able to reason about the interaction between features using the metalevel representation of the entire program. For each feature we can now determine with which features it interacts directly (through method calls or access of shared variables). In a system implementing a lot of features, a developer could at least derive which features that could be affected by a change in a particular feature (or the addition of a new feature). A logic rule that detects access to the same instance variable by two different features is written as follows[3]:

```
sharedInstanceVariable(?feature1,?feature2,?sharedInstanceVariable) if
    methodInFeature(?feature1,?method1),
    accesses(?method1,?sharedInstanceVariable),
    methodInFeature(?feature2,?method2),
    accesses(?method2,?sharedInstanceVariable).
```

For clarity, we also include the implementation of predicates used in the rule above:

```
methodInFeature(?feature,?methodDescription) if
    feature(?feature,?list), member(method(?className,?methodName),?list),
    methodInClass(?className,?methodName,?methodDescription).

accesses(?method,?instVar) if
     reads(?method,?instVar).
accesses(?method,?instVar) if
     writes(?method,?instVar).
```

The rules `methodInClass`, `reads` and `writes` are part of the SOUL framework developed in [13]. We will not show them here, but they are implemented by several logic rules that reason about the logic metalevel representation.

[3]Logic variables are written using a '?'

However, using these rules, we can only detect feature interactions. To detect feature interference we include logic rules that express constraints or invariants on the implementation. For example:

```
archivalInvariant() if
    classInFeature(figures,?class),
    instVar(?class,?instVar),
    methodInFeature(archival,?method), accesses(?method,?instVar).
```

This rule expresses a simple invariant which states for the *archival* feature that every instance variable in classes of the feature *figures* should be accessed by a method of the *archival* feature. This expresses the condition that the *archival* feature should save every part of the state of the figures.

Adding a new feature to our figure editor might introduce conflicts with the other features, depending on the implementation. As we illustrated, adding colors to the figures will interfere with the *archival* feature. Whether we change the original feature *figures* or we add a complete new *color* feature, the change will boil down to introducing new state variables to `Point`, `Line` and `Polygon` classes. If we do not change the implementation of the `store` method, the archival invariant will not be satisfied and a feature interference will be detected by the resolution engine.

In this experimental approach, LMP promises to be a viable technique to support feature interaction problems in software development. Future research will investigate on a general methodology for feature interaction detection, using LMP.

4 Summary

We described two approaches dealing with feature interactions in software engineering using logic. The first approach defined a formal language for feature modeling in the problem domain using DL. The second approach uses the LMP approach to detect feature interaction in the solution domain. In both approaches, open research questions are related to which kind of information is necessary and sufficient to allow reasoning about feature interactions.

References

[1] R. Accorsi, C. Areces, and M. de Rijke. Towards Feature Interaction via Stable Models. In *Proceedings of the 2nd WFM*, Florianópolis, Brasil, October 1999.

[2] C. Areces. *Logic Engineering. The Case of Description and Hybrid Logics.* PhD thesis, ILLC University of Amsterdam, 2000.

[3] C. Areces, W. Bouma, and M. de Rijke. Description Logics and Feature Interaction. In P. Lambrix, A. Borgida, M. Lenzerini, R. Möller, and P. Patel-Schneider, editors, *Proceedings of the International Workshop on Description Logics (DL'99)*, pages 28–32, 1999.

[4] C. Areces, W. Bouma, and M. de Rijke. Feature Interaction as a Satisfiability Problem. In *Proceedings of* MASCOTS'99, October 1999.

[5] F. Baader and P. Hanschke. A Scheme for Integrating Concrete Domains into Concept Languages. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence, IJCAI-91*, pages 452–457, Sydney (Australia), 1991.

[6] K. Czarnecki and U. W. Eisenecker. *Generative Programming (Methods, Tools, And Applications).* Addison Wesley, 2000.

[7] Jonathan D. Hay and Joanne M. Atlee. Composing Features and Resolving Interactions. In David S. Rosenblum, editor, *Proceedings of Eighth International Symposium on the Foundations of Software Engineering*, pages 110–119. ACM Press, November 2000.

[8] K. Kang, S, Cohen, J. Hess, W. Nowak, and S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute and Carnegie Mellon University, Pittsburgh PA, November 1990.

[9] H. Klaeren, E. Pulvermueller, A. Rashid, and A. Speck. Aspect Composition applying the Design by Contract Principle. In *Proceedings of the Net.ObjectDays2000*, Erfúrt, Germany, October 2000.

[10] Buchheit M., Donini F., and Schaerf A. Decidable Reasoning in Terminological Knowledge Representation Systems. *Journal of Artificial Intelligence Research*, 1:109–138, 1993.

[11] Kim Mens. *Automating Architectural Conformance Checking by means of Logic Meta Programming.* PhD thesis, Vrije Universiteit Brussel, October 2000.

[12] Keck D. O. and Kuehn P.J. The Feature and Service Interaction Problem in Telecommunications Systems: A Survey. *IEEE Transactions on Software Engineering*, 24(10):779–796, October 1998.

[13] Roel Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Vrije Universiteit Brussel, January 2001.