# Ontology-based Model Comparison

Katharina Wolter, Thorsten Krebs, Lothar Hotz

HITeC e.V. c/o University of Hamburg
kwolter, krebs, hotz @ informatik.uni-hamburg.de

**Abstract.** This paper proposes an ontology-based approach for comparing software products modelled with UML. An ontology is used that defines the structure of all software products developed so far or even all products that can be developed using the specified architecture. Using such an ontology the models of different software products can be compared more effectively.

## 1 Problems in Comparing UML Models

UML (Unified Modelling Language[1]) models are often used to specify software products. The modelling facilities of UML include, among others, *classes* that can be used to represent the product's components (of any kind), *attributes* that describe properties of a class, *specialisation relations* for modelling a taxonomic hierarchy of classes and *compositional relations* (i.e. aggregation and composition) for modelling a partonomy of classes. With these modelling facilities the product architecture can be specified.

Typically, one UML model represents one software product. This means that for every product a new model is created and that there is no direct relation between the models of different products. Comparing products – i.e. comparing models – is hampered by the fact that there is no such relation between the different product models. A lexical comparison, for example can test for occurrence of classes with identical names in different models, but the outcome of this comparison heavily depends on the naming of classes. Two semantically identical classes with different names will not be recognised as being identical or similar when using a lexical comparison.

The properties and relations of a class have to be compared rather than their names. Single classes can be compared according to their position in the taxonomy: distance in the taxonomic hierarchy is a heuristic similarity measure. Sub-graphs of classes can be compared according to their compositional relations: comparing parts in partonomy is a heuristic similarity measure, as well.

## 2 Formalising Software Product Models in Structure-based Configuration Models

Configuration is a well-known approach to assemble products from a given set of components. The components are specified as *concepts* together with their *attributes* and relations to other components in a *configuration model*. Structure-based configuration explicitly defines a taxonomy and partonomy of all configurable components, forming an AND/OR graph[2]. Thus, a configuration model is a kind of ontology, but it contains additional knowledge entities: like constraints that define restrictions between a number of concepts and their properties. Within a configuration model, all admissible configurations are specified. One specific product is configured by selecting a component and configuring its descendants in both the taxonomic and compositional relations.

When a product is configured, instances of the concept definitions are created dynamically during the configuration process. Having product instances (i.e. configuration solutions) it is known from which concepts the instances have been instantiated. This enables the comparison of two distinct product models based on the configuration model.

Traditional application areas for configuration are technical domains like automobiles, computers, drive systems, etc. But the configuration approach is not limited to this field. The ConIPF (Configuration in Industrial Product Families) project[3], for example, has shown that configuration is also applicable for software domains [Hotz et al. 2006].

## 3 Using Ontology to Model Software

Similar to configuration models from structure-based configuration, multiple software products can be specified in one model. Its ontological structure enables the creation of both taxonomy and partonomy of the software components. Figure 1 shows how software can be modelled from modules, applications and libraries. An application is composed of modules and can use external libraries. Modules can be further decomposed, like a software module that consists of multiple class definitions, and aligned in a taxonomic hierarchy defining general concepts and their specialisations. All entities are software elements.

### 3.1 Building the Ontology

There are two ways how to define an ontology of software products: predefining it and incrementally building it from singe product models.

[1] http://www.uml.org/

[2] Partonomy is considered to be conjunctive (select some of the parts) while taxonomy is disjunctive (select one of the specialisations).

[3] http://www.conipf.org

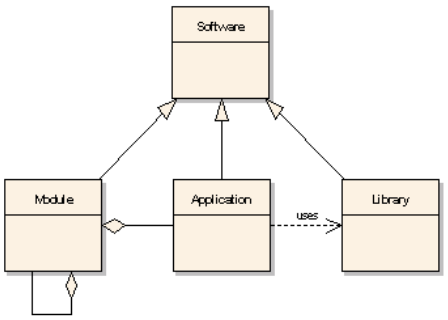


Figure 1: Modelling Software using specialisation relations and compositional relations (UML Notation).

Predefining a software ontology means first designing the architecture to derive products from and designing and implementing the components and afterwards assembling new products based on this architecture. This is a typical approach in configuration [Hotz and Krebs 2003] or (software) product lines [Clements and Northrop 2002].

Incrementally building a software ontology from single software product models includes refinement of the product architecture for every software product to be included. Concepts representing existing components can simply be linked to the concept representing the product while new concepts have to be introduced for components that are not yet modelled. For every product included in the software ontology the architecture is extended. It is therefore essential that the ontology is thoroughly checked for consistency. Tool support is expected to improve this process.

While the first approach for building the ontology requires a larger initial effort the second can result in larger restructuring effort for integration of further product models.

### 3.2 Comparing Two Software Models with an Ontology

Let us consider an example: a family of text editors is modelled in an ontology. A concept representing the editor itself aggregates concepts that represent the editor's components.

Figure 2 shows how two different text editors can be modelled within one software model. The lightweight editor has a find module, while the heavyweight editor has a find-and-replace module. The find-and-replace module is further composed of a find module and a replace module.

Having such a model, the architecture of multiple products that have been created with this model can be compared more effectively. Lexically comparing the find module and the find-and-replace module, no similarity would be recognised. Using the ontology that defines the product's architecture one can additionally recognise that the find module is a part of the find-and-replace module. This means that latter subsumes the former. Thus, both modules – and therefore both products – are similar!

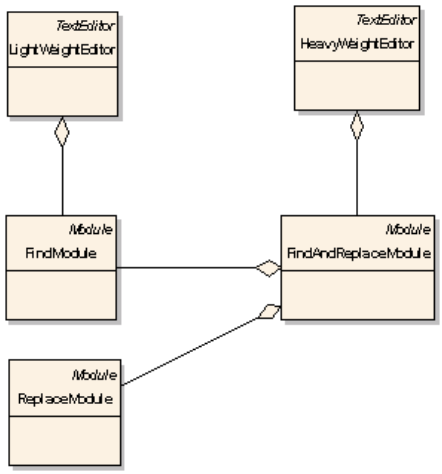


Figure 2: Modelling different software products within one model (UML Notation).

## Acknowledgement

This work is partially funded by the EU: Requirements-driven Software Development System (ReDSeeDS) (contract no. IST-2006-33596). The project is coordinated by Infovide, Poland with technical lead of Wasaw University of Technology and with University of Koblenz-Landau, Vienna University of Technology, Fraunhofer IESE, University of Latvia, HITeC e.V. c/o University of Hamburg, Heriot-Watt University, PRO DV, Cybersoft and Algoritmu Sistemos.

## References

[Clements and Northrop 2002] P. Clements, L. Northrop: Software Product Lines: Practices and Patterns. Addison-Wesley, 2002.

[Hotz and Krebs 2003] L. Hotz, T. Krebs: Configuration - State of the Art and New Challenges. In Proceedings of 17. Workshop, Planen, Scheduling und Konfigurieren, Entwerfen (PuK2003), pp. 145-157, Hamburg, Germany, 2003.

[Hotz et al. 2006] L. Hotz, K.Wolter, T. Krebs, S. Deelstra, M. Sinnema, J. Nijhuis, and J. MacGregor. Configuration in Industrial Product Families - The ConIPF Methodology. IOS Press, Berlin, 2006.