Ontology Support for Abstraction Layer Modularization

Hyun Cho and Jeff Gray
Department of Computer Science
University of Alabama
Tuscaloosa, AL USA
{hcho7, gray}@cs.ua.edu

Jules White
Bradley Dept. of Electrical and Computer Engineering
Virginia Tech
Blacksburg, VA USA
jules.white@vt.edu

***Abstract*—Abstraction layers have been widely used to increase the portability of a software system by hiding the implementation details of underlying resources (e.g., OS, hardware, and reusable libraries). Abstraction layers have also been adopted in Software Product Lines (SPLs), which assist in the creation of a family of products by reusing common core assets and managing variants in a family domain. An abstraction layer provides transparent and unified access to the APIs of underlying resources. Abstraction layer APIs are modularizd by generalizing the APIs of underlying resources based on semantic similarity across common resources. Thus, an abstraction layer inherently needs to handle the semantic variants of the underlying APIs. However, the lack of a systematic approach for evolving an abstraction layer in accordance with the evolution of underlying resources may restrict its usage. This paper describes an approach toward ontology-based feature modeling to build and maintain the abstraction layer in a modularized and systematic way. The combination of ontologies and feature modeling can assist in modularizing abstraction layers by identifying the semantic similarities in APIs and provide insight into the variability of the underlying resources.**

Keywords; Software Product Lines, Abstraction Layer, Ontology, Feature model, Model-Driven Engineering

I. INTRODUCTION

Software complexity continues to increase because of the need to accommodate frequently changing requirements, and to utilize the advances in hardware and software components. Software systems often need to be ported onto various underlying resources (e.g., OS, hardware, and libraries) to reduce the cost of development and to provide better customer satisfaction. Each underlying resource provides its own set of Application Programming Interfaces (APIs), which may also evolve over time. Portability is also an important factor when designing and implementing core assets, such as reusable libraries and device drivers, in software product lines [5][30]. Abstraction layers can assist in reducing the dependency between core assets and the underlying resources.

Abstraction layers provide a set of APIs that classify and generalize underlying resources (e.g., OS, hardware, libraries) with the goal of increasing the portability of core assets and enabling the reuse of their design and implementation. For example, a hardware abstraction layer (HAL) is typically located between the hardware and operating system (OS) or device drivers. By calling APIs in the HAL, an OS can be programmed more flexibly to support various underlying hardware. This programming flexibility helps in porting the OS or device drivers onto other target processors and boards. Windows CE OAL [34] and eCos HAL [35] are examples of popular HALs.

The OS Abstraction Layer (OSAL) [1][23] is another example of an abstraction layer. Several OS implementations [34][35][36] have been developed and maintained to support either a specific domain or a general domain. OSAL provides common OS APIs by generalizing OS architectures that support a target domain. Portable Operating System Interface (POSIX) [33], which provides standard APIs for OS as well as thread management, is an early example representing the principles of OSAL. These abstraction layers are also widely used in software product lines to develop core assets that support various underlying resources. Although many researchers have proposed the principles and guidelines for development of abstraction layers, the systematic management of the abstraction layer as the underlying APIs evolve has not been considered deeply. This paper introduces an ontology-based feature modeling technique for managing abstraction layers in a modularized way.

The term ontology originated from philosophy and has been ascribed several definitions. Grube et el. [12] and Uschold et el. [28] define an ontology as a set of definitions that represent shared knowledge of a specific domain with modeling primitives such as classes, relations, functions, and constraints. Studer et el. [25] define ontology as an explicit specification of consensual domain knowledge and the specification is structured for machine manipulation. The notion of ontology is widespread in the areas of information integration and retrieval [13][29], knowledge engineering [11][25], and natural language processing [17]. In these areas, an ontological approach can transform the intellectual and conceptual knowledge into a form of computation, such that the transformed knowledge can be shared and reused. Recently, technologies based on ontology have been recognized as a means of addressing interoperability issues stemming from semantic differences between systems. For example, web services use ontology to interpret and invoke services with different calling conventions [24]. The enabling technology for solving the interoperability problem is based on matching semantics through ontologies [6][19][20][22][26] and has been applied to conceptual structures such as database and XML schemas. The matching method is also the core of our approach for modularizing underlying APIs by exploring API documents.

Ontologies contribute to early stage domain analysis by identifying the representative vocabularies of a domain and by representing the body of knowledge with the representative vocabularies [4]. Feature modeling [16] can reconcile the view of a domain by capturing the commonality and variability. The combined use of ontologies and feature modeling can be leveraged to represent the properties of a domain, especially how underlying APIs can be modularized by their signatures and descriptions. This feature model and ontology synergy can also be used to represent the semantic relationships among underlying resources. The remainder of this paper is organized as follows: Section II describes an approach for modularizing the abstraction layer with ontologies and feature modeling. Section III relates this work to previous related research. Section IV concludes the paper with expected contributions.

II. ONTOLOGY-BASED ABSTRACTION LAYER MANAGEMENT

Feature modeling was proposed by Kang et el. [16] and has been widely used to analyze a specific domain and represent variants of the domain. Feature modeling has also been used to design domain-specific languages because feature models provide a comprehensive way for modeling the commonalities and variabilities of the domain with a simple notation [31]. Feature modeling also provides a fundamental form of modularity that can be referenced across the software lifecycle by different downstream models (e.g., software product line architecture, core asset design, and programming model).

The combinatory use of feature modeling and ontologies can help to modularize an abstraction layer in two ways:

- APIs that have similar semantic meaning are grouped into a feature. This grouping helps to determine how to modularize the abstraction layer to cover underlying APIs.
- Each feature can show variability of APIs that provide similar functionality. APIs grouped into a feature by their semantic similarity and APIs in a feature can have different signatures. Thus, by examining variants in a feature, engineers can define abstraction layer APIs that cover all variability in the feature.

An ontology for an abstraction layer can be built by following the process described in [7][21]. In order to obtain a feature model for the abstraction layer, two questions are asked: "How are the APIs related to each other?" and "How can the commonalities and variabilities in the APIs affect the construction and maintenance of the modularity in an abstraction layer?" The ontology for an abstraction layer is constructed by examining APIs, which have the following characteristics:

- Documented in a single file. If the document is released as HTML, the document can have multiple HTML files. However, as they are also released as a package, multiple HTML files can be individually considered as a single document.
- Multiple semantic units. APIs can be decomposed into multiple semantic units. For example, OS APIs have several semantic units (e.g., thread management, memory management, and I/O management).
- Formed as a tree. APIs are normally structured with a hierarchy (class hierarchy) and can be represented in the form of a tree.

The goals of our approach are to identify semantic units in APIs, modularizing underlying APIs according to their semantic units, and then construct the feature model to represent the modularity of an abstraction layer. The framework for ontology-based abstraction layer management is shown in Figure 1. The framework consists of three parts: Abstraction Layer Modeler, Traceability Relation Manager, and API Generator.

The Abstraction Layer Modeler is comprised of three components: Term Extractor, Rule Composer, and Matcher. The Term Extractor extracts the domain terms from the API documents. The extracted domain terms are classified based on semantic similarity and the classified terms will be the basis to guide how to modularize the abstraction layer.

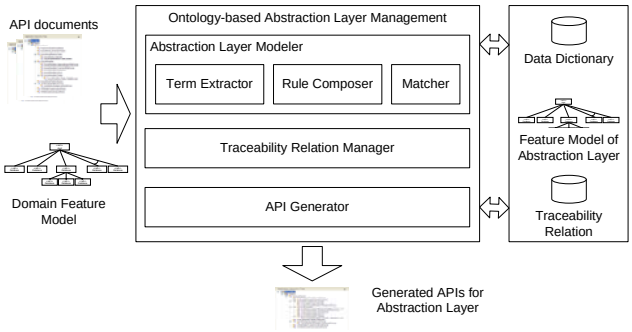

Figure 1. The framework of the ontology-based abstraction layer management

The domain feature model can be used as a reference for constructing ontologies (the domain feature model is the result of the domain analysis and it is modeled with the standard terms of the domain). The Rule Composer provides preliminary classification of the extracted domain terms based on their ontological similarity and allows users to compose the matching rules. After the rules are composed, the Matcher retrieves API documents and classifies APIs with a two-pass matching technique. During the first pass, the Matcher identifies the structure of API documents to build the relationships between ontologies, and then analyzes the semantic similarity of APIs to create a feature model.

According to [18], the structured relation strongly influences the computation of similarity and a graph-based comparison tends to improve the performance of the computation. If the API design is object-oriented, the APIs are structured as a tree. Thus, the Matcher can leverage the structural relation of APIs for classification. In addition, the identified structure of API documents will assist in determining the depth of features when constructing the abstraction feature model.

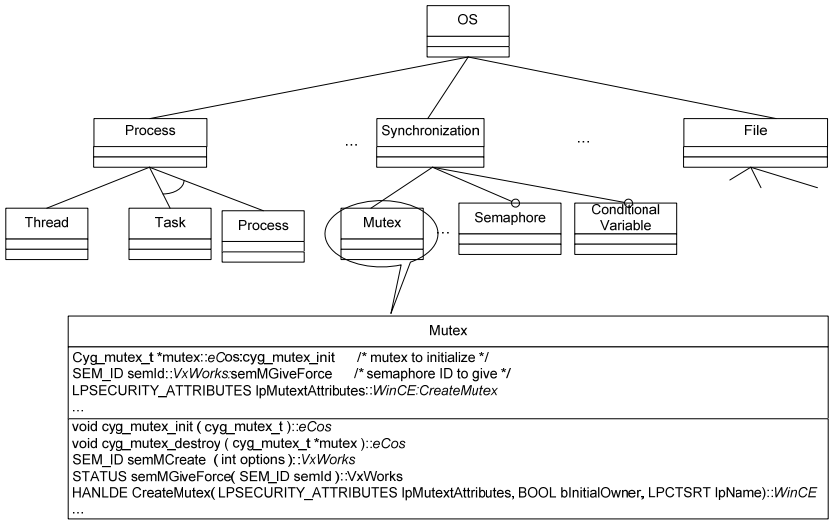

Figure 2. Feature model of Abstraction Layer

To represent the modularity of APIs in the abstraction layer, the original feature model is modified to group APIs based on their ontological similarity. The top part of the feature model captures features, identified ontologies from the term extractor, and the middle part of the feature model lists all attributes used in the APIs. The syntax of an attribute is defined as *datatype VariableName::PackageName:APIName*. The first part, *datatype VariableName*, represents the data type and variable name and the second part, *PackageName:APIName,* indicates the origin of the attribute. The bottom layer lists APIs that are classified as having semantic similarity. APIs in this layer have similar syntax with the attribute section, *ReturnType APIName:: PackageName*, to indicate the origin of the API.

An example feature model for an OS abstraction layer is shown in Figure 2. OS vendors (e.g., Microsoft Windows CE [34], eCos [35], and WindRiver VxWorks [36]) provide various APIs to manage memory, file, process/task, etc. Because most OS' have APIs to synchronize processes/tasks, synchronization is modeled as a mandatory feature. However, each OS provides a different set of synchronization mechanisms. For example, some OS' may provide Mutex as a primitive process for synchronization, but not semaphores and conditional variables.

The Traceability Relation Manager (TRM) manages the links between APIs and the feature model, and allows users to query how the APIs are mapped onto the feature model (and vice versa). The links are created when Matcher finds APIs that have similar semantics and maps those APIs onto a feature. In addition, the traceability relation is referenced by the Matcher to help the modularization process when a new set of APIs or new versions are introduced to the abstraction layer. Finally, the API Generator generates APIs for the abstraction layer by referring to the feature model. The feature model of the abstraction layer covers the entire range of underlying resources. For example, the feature model of OS abstraction can cover APIs from general-purpose to very small OS. Thus, APIs in the abstraction layer should vary for targeted OS and the code generator can generate APIs selectively from feature modules. Currently, the API Generator is hard-coded to support a specific programming language. In the future, it will be implemented using model-driven engineering to support various programming languages.

III. RELATED WORKS

Core assets that are designed for software product lines have to consider running on many different OS, hardware, and libraries. The reusability and portability of the core assets led to the idea of abstraction layers. Constructing the abstraction layer is the task of building transparent and unified APIs to access the underlying resources. One of the major challenges is to classify various heterogeneous APIs based on their semantic similarity. Researchers have introduced the principles and the benefits of abstraction layers. Andrea et el. [1] and Probert [23] presented the principles for building an OS abstraction layer (OSAL) that minimizes the conflicts between operating systems. Handziski [15] and Yoo et el. [32] introduced the hardware abstraction layer for wireless sensor networks and systems on a chip, respectively. These works described the issues and resolution to building the abstraction layer, but they were not focused on how to manage the abstraction layer systematically.

Ontologies are used to develop knowledge-based applications [2][8][10] by reasoning about the semantics of the domain-specific content. Several researchers [3][9][14][27] have investigated the application of metamodels and ontologies for domain analysis. Our approach is similar, but we introduced feature models, instead of class diagrams, to represent the commonalities and variabilities of the domain.

IV. RESEARCH ISSUES AND CONCLUSIONS

The approach introduced in this paper can provide a systematic way to modularize the abstraction layer through the combination of ontologies and feature modeling. Although the approach is still at a preliminary stage and under development, it has the potential to assist in maintaining abstraction layers in a modularized way. The following lists several benefits of the approach, as well as future directions for this research:

- Our approach can help maintain the abstraction layer consistently through systematic matching and

generalization techniques. When a new API version is released or the abstraction layer needs to support a new underlying resource, the approach can systematically identify the differences between existing APIs in the abstraction layer and the new API.

- A feature model can provide insight into the modularity and functionality of the underlying resources. As underlying APIs are grouped into features by their semantic similarity, the abstraction layer feature model can represent the variabilities in the API domain and each feature can represent the variabilities in the API signature. In addition, the feature model can provide the comparative information about the underlying resources. This comparative information can guide the abstraction layer maintainers and underlying resource developers to predict how APIs will be evolved to make up their deficiencies.

- The approach is transparent to the implementation technology of underlying resources. The Abstraction Layer Model in Figure 1 is also designed to deal with various underlying languages and it can modularize underlying APIs by referring to the grammar or syntax of target languages. For example, the iPhone SDK is developed based on Objective-C and the Android SDK is released in the form of Java. However, both SDKs target the same technology space (i.e., mobile applications) and they have many commonalities. Thus, even though their specific SDK implementation may differ, our approach can be applied to the two SDKs by designing more complex rules and matching algorithm.

- Generative programming has the potential to automate the creation of APIs for the abstraction layer. Two types of codes should be developed for the abstraction layer. One is the API itself and the other is mapping codes, which map APIs between the abstraction layer and the underlying resources. To maximize the benefits of modeling, generative programming techniques need to be combined with feature model concepts to generate the abstraction layer APIs for both a specific target and mapping codes.

- Assessing the degree of modularity is one of challenges in this approach. The modularity of the abstraction layer is important by itself. However, the abstraction layers are built by analyzing the underlying resources, such that the modularity of the abstraction layers may be affected by different change sources instead of (non)functional requirements (e.g., the modularity of underlying resources and ontology matching rules), which are typical sources that affect modularity. Thus, we consider using a pair-wise comparison technique to assess the modularity of the abstraction layer. By comparing the modularity between the abstraction layers and underlying resources, or changing the matching rules, we expect to understand how the modularity of underlying resources affects the modularity of the abstraction layer and how ontologies influence the modularity of the abstraction layers.

ACKNOWLEDGMENT

This research is supported in part by an NSF CAREER award, CCF-0643725.

REFERENCES

[1] M. Andree, H. Karl, M. Herlich, J. Catalano, A. Schoofs, P. van der Stok, L. Vazago, L. von Allmen, R. S. Oliver, G. Fohler, C. Brandolese, M. Hauspie, G. Grimaud, s. Buisine, E. Fleury, A. Fraboulet, A. Picu, and F. Bouwens, "Core Hardware Abstraction and Programming Model,Deliverable D3.2," IST-034963, WASP, (2008)

[2] J. Bateman, T. Kamps, J. Kleinz, and K. Reichenberger, "Toward constructive text, diagram and layout generation for information presentation," *Computational Linguistics* 27(3), 2001, pp. 409-449.

[3] J. Bezivin, V. Devedzic, D. Djuric, J.M. Favreau, D. Gasevic, and F. Jouault, "An M3-Neutral infrastructure for bridging model engineering and ontology engineering," In *Proceedings of the first International Conference on Interoperability of Enteprise Software and Applications*, (INTEROP-ESA 05), Geneva, Switzerland, February 2005.

[4] B. Chandrasekaran, J. Josephson, V. Benjamins, "What Are Ontologies, and Why Do We Need Them?" *IEEE Intelligent Systems* 14, 1999, pp. 20-26.

[5] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, Boston, MA, 2001.

[6] A. Doan, J. Madhavan, P. Domingos, and A. Halevy. Ontology matching: A machine learning approach. *Handbook on Ontologies*, pp. 385-516, 2003.

[7] B. Du Bois, "Towards an ontology of factors influencing reverse engineering," In *Proceedings of the 13th IEEE International Workshop on Software Technology and Engineering Practice*, Budapest, Hungary, September 2005, pp. 74-80.

[8] J. Davies, F. van Harmelen, and D. Fensel, Eds. *Towards the Semantic Web: Ontology-based Knowledge Management*. John Wiley & Sons, Inc., 2002.

[9] D. Gašević, V. Devedžić, D. Djurić, "MDA Standards for Ontology Development," *International Conference on Web Engineer*ing, Munich, Germany, July 2004.

[10] J. Geurts, S. Bocconi, J. R. van Ossenbruggen, and L. Hardman, *Toward Ontology-driven Discourse: From Semantic Graphs to Multimedia Presneation*, CWI Technical Report INS-R0305, May 2003.

[11] A. Gomez-Perez, R. Benjamins, "Overview of Knowledge Sharing and Reuse Components: Ontologies and Problem-Solving Methods," In: *Proceedings of the IJCAI-99 Workshop on Ontologies and Problem-Solving Methods*, Stockhol, Sweden, August 1999.

[12] T.R. Gruber. Towards principles for the design of ontologies used for knowledge sharing. In *Formal Ontology in Conceptual Analysis and Knowledge Representation*. Kluwer, Boston, MA, 1994.

[13] N. Guarino, "Semantic Matching: Formal Ontological Distinctions for Information Organization, Extraction, and Integration," In *Information Extraction: A Multidisciplinary Approach to an Emerging Information Technology*, Springer Verlag LNAI 1299, September 1997, pp. 139-170.

[14] N. Guarino, C. Welty, "Towards a Methodology for Ontology-based MDE," *First International Workshop on MDE*, Nice, France, June 2000,

[15] V. Handziski, J. Polastre, J.H. Hauser, C. Sharp, A. Wolisz, and D. Cullar, "Flexible hardware abstraction for wirelss sensor networks," In *Proceedings of 2nd European Workshop on Wireless Sensor Networks*, Istanbul, Turkey, February 2005.

[16] K.C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, *Feature-oriented Domain Analysis (FODA) Feasibility Study*, Technical Report, CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.

[17] K. Mahesh, and S. Nirenburg, "A situated ontology for practical NLP," In *Proceedings Workshop on Basic Ontological Issues in Knowledge Sharing. International Joint Conference on Artificial Intelligence*, Montreal, Canada, August 1995.

[18] D.L. Medin, R.L. Goidstone, and D. Gentner, "Respects for similarity," *Psychological Review*, 100, pp. 254-278, 1993.

[19] D. McGuinness, R. Fikes, Rice, J., and S. Wilder, "The Chimaera ontology environment," *In Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on innovative Applications of Artificial Intelligence*, Austin, TX, August 2000, pp 1123-1124.

[20] J. Madhavan, P. A. Bernstein, and E. Rham, "Generic Schema Matching with Cupid," *In Proceedings of the VLDB*, San Francisco, CA, September 2001, pp. 49-58.

[21] N. Noy, D. McGuinness, "Ontology Development 101: A Guide to Creating Your First Ontology," http://www-ksl.stanford.edu/people/dlm/papers/ontology-tutorial-noy-mcguinness.pdf.

[22] J. Poole and J. Campbell, "A Novel Algorithm for Matching Conceptual and Related Graphs," In *Conceptual Structures: Applications,Implementation and Theory*, Santa Cruz, CA, Springer-Verlag, LNAI 954, August 1995, pp. 293-307.

[23] D. Probert, J.L. Bruno, and M. Karzaorman, "SPACE: A new approach to operating system abstraction," In *International workshop Object Orinetation in Operating Systems*, Palo Alto, CA, October 1991, pp. 133-137.

[24] N. Srinivasan, M. Paolucci, and K. Sycara, "An Efficient Algorithm for OWL-S based Semantic Search in UDDI," In *Proceedings of First International Semantic Web Services and Web Process Composition Workshop*, San Diego, CA, July 2004.

[25] R. Studer, V.R. Benjamins, D. Fensel, "Knowledge engineering: principles and methods," *Data and Knowledge Engineering* 25 (1998) pp. 161-197.

[26] G. Stumme and M. Alexander, "FCA-MERGE: Bottom-up merging of ontologies, " In *7th International. Conference on Artificial Intelligence*, Seattle, WA, August 2001, pp. 225-230.

[27] R. Tairas, M. Mernik, and J. Gray, "Using Ontologies in the Domain Analysis of Domain-Specific Languages," *Workshop on Transforming and Weaving Ontologies and Model Driven Engineering*, Springer-Verlag LNCS 5421 (Workshops and Symposia at MODELS 2008), Toulouse, France, September 2008, pp. 332-342.

[28] M. Uschold and M. Gruninger, 1996, "Ontologies: Principles, Methods and Applications." *The Knowledge Engineering Review*, 11(2): 93-136.

[29] H.Wache, T. Voegele, U. Visser, H. Stuckenschmidt, G. Schuster, H. Neumann, and S. Huebner, "Ontology-based integration of information - a survey of existing approaches," In *Proceedings of the Workshop on Ontologies and Information Sharing at the International Joint Conference on Artificial Intelligence*, Seattle, WA, August 2001, pp. 108-117.

[30] D. M. Weiss and C. T. R. Lai. *Software Product-Line Engineering: A Family-Based Software Development Process*. Addison-Wesley, Boston, MA, 1999.

[31] J. White, J. Hill, J. Gray, S. Tambe, D. C. Schmidt, and A. Gokhale, "Improving Domain-specific Language Reuse through Software Product-line Configuration Techniques," *IEEE Software Special Issue on Domain-Specific Languages and Modeling*, July/August, 2009, Vol. 26, No.4, pp. 47-53.

[32] S. Yoo, and A. A. Jerraya, "Introduction to hardware abstraction layers for SoC," In *Proceedings of the Design, Automation, and Test in European Conference and Exhibition*, Munich, Germany, March 2003, pp. 336-337.

[33] POSIX.1 (2001). IEEE Std 1003.1:2001. Standard for Information Technology -Portable Operating System Interface (POSIX). The Institute of Electrical and Electronic Engineers, 2001.

[34] http://msdn.microsoft.com/en-us/library/aa447042(v=MSDN.10).aspx

[35] http://ecos.sourceware.org/

[36] http://www.windriver.com/products/vxworks