Ontological Excavation: Unearthing the core concepts of the application

Idris Hsi, Colin Potts
College of Computing
Georgia Institute of Technology
Atlanta, GA 30032-0280
idris, potts@cc.gatech.edu

Melody Moore
Computer Information Systems Department Georgia State University
Atlanta, GA 30392
melody@gsu.edu

Abstract

Applications possess and implement a specific "theory of the world" or ontology. Recovering and modeling this ontology may help inform software developers seeking to extend or adapt an application's functionality for its next release. We have developed a method for the black-box reverse engineering or excavation of an application's ontology. The ontology is represented as a semantic network, and graph theoretic measures are used to identify core concepts. Core concepts contribute disproportionately to the structural integrity of the ontology. We present analyses of ontologies excavated from several interactive applications. From a set of several candidate metrics for identifying core concepts we find node betweenness centrality is a good measure of a concept's influence on ontological integrity and that the k-core algorithm may be useful for identifying cohesive subgroups of core features. We conclude by discussing how these analyses can be applied to support application evolution.

Keywords: domain analysis, reverse engineering, software evolution, software metrics

1. Introduction

Applications are designed to solve specific problems in the real world. Because technologies, business process, work practices, and user sensibilities in the real world change over time, the services provided by the application can drift out of synch with the current requirements. Lehman's Laws of Software Evolution state that applications have to be continually evolved to remain useful [1, 2]. Thus, during software maintenance, developers are often faced with decisions regarding how the application should be evolved or adapted for its users to meet these new demands: Functionality might be added, removed, or modified; Specific operations may need to be optimized or enhanced; The entire application may need to be reengineered for a different context of use. How are these decisions made and what costs have to be considered?

An application can be evolved by adding features to it. More features may improve its overall functionality and may cause the application to appear more attractive to customers comparing it to similar applications with fewer features. However, adding too many features may induce "feature creep" or "bloat" that may alienate or hinder the users [3, 4]. Alternatively, an application's features can be optimized or extended. Optimizing its existing functionality or extending its capabilities could improve its services but could also cause it to fall out of synch with the existing goals and procedures of its users. Informed decisions that account for these possibilities cannot be made solely on an understanding of the underlying code and architecture. Instead, software developers will need to examine what the application knows and understands about its problem area or *domain* [5].

Arango and Prieto-Díaz state that a domain exists if it has "deep or comprehensive relationships among the items of information", a community that has a stake in solving its problems, and a store of knowledge that can be used towards solving these problems [6]. Because applications are designed and constructed primarily to solve problems within a domain, we argue that all applications possess a "theory of the world" that captures this domain knowledge. For example, a scheduling or calendaring application has a theory about how its users want to organize their time or be notified about their appointments. Users that meet frequently in a shared building include a notion of room scheduling in their meeting scheduling domain that the scheduler application needs to encode. Extending the feature set of an application to accommodate new demands from the users will naturally alter the application's theory of the world. If the application's theory of the world falls too far out of synch with the theory of the world as understood by the users in the domain, then the application has failed to evolve in a useful fashion. A method for analyzing the application's theory of the world and identifying these potential inconsistencies could identify such disconnects during development. To this end, we first model this theory of the world in an *ontology*. An ontology defines and represents the domain's concepts and relationships using a formalized vocabulary [7] and representation. Under this definition, for example, entity-relationship diagrams and object-oriented models are types of ontologies.

Now, one can argue that some concepts will be more important than others within the ontology. For example, the concept of "paragraph" is likely to be more important than the concept of "font color" to a word processor. One could also argue that some of the concepts are somehow essential to that application's ontology. A personal scheduler wouldn't make much sense if it didn't have the concept of "appointment" or "event". We call these essential concepts *core concepts* because they have some central importance to the application, possibly contributing disproportionately to the structure of its ontology.

Changes to the application that involve its core concepts have the potential to be extremely difficult to engineer or can alter the application's functionality in undesirable ways. Changes to the application that affect only those non-core or *peripheral* concepts can still impact the core concepts. Any alterations to core concepts often translate to additional costs in regression testing and development. Thus, in addition to recovering an application's ontology prior to performing evolution or adaptation activities, it will be useful to have a method for identifying which concepts are core concepts.

Previous work in domain analysis and reverse engineering has developed methods for extracting the domain from program documentation [8], requirements specifications [9], code [10, 11], and interviews with domain experts [12]. Of these techniques, code domain analysis offers the closest method for obtaining the application's ontology but code itself contains a meta-domain with concepts and relationships that concern software engineering and programming. We wish to uncover only the concepts and relationships that are visible to the users as they interact with the application. To this end, we have developed a method for the black-box reverse engineering of an application's ontology. We call this reverse engineering process *ontological excavation* because the ontology is recovered by digging through the application's external interfaces.

In Section 2, we explain how we excavate the ontology and review the candidate metrics we used to identify the core concepts. In Section 3, we present the results from our excavations of three modern and interactive systems: the Windows 95/98 CD Player, the Palm Pilot Scheduler, and Windows Notepad. In Section 4, we discuss our findings and possible criticisms of this work. Lastly, we conclude with our plans for future work.

2. Ontological Excavation and Analysis

2.1 Building a map of the morphology.

In our research framework, all applications have a *morphology*, the external interface elements of the system that give its users access to the implemented functionality. In interactive systems, the morphology is the user interface. The components comprising the morphology represent windows or *portals* through the external "shell" of the application to the underlying ontology [4]. Through systematic interaction with the application's outer shell, we can identify or "excavate" the concepts and the basic relationships between those concepts and model them in a semantic network.

We first model the user interface in an *interface map*. This map consists of the UI's containers (e.g. windows, dialog boxes, toolbars), interactive elements or interactors (e.g. buttons, text fields, check boxes), and information displays. The visual icons representing these major components of the UI are also linked using arrows to show either their point of containment or their point of activation. Figure 1 shows a portion of the Notepad menu.

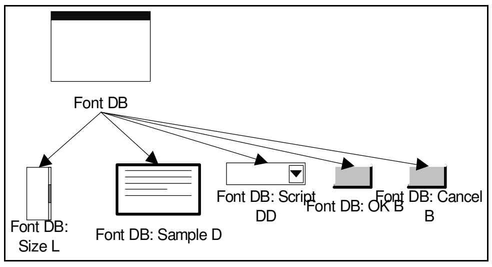

Figure 1 - Notepad Morphology showing Font Dialog Box, Font List, Font Display, Script Dialog Box, and OK/Cancel Buttons

We build this map by systematically traversing and activating all the user interface elements in a depth first fashion. These elements, their labels, and their interconnections are modeled using Microsoft Visio as the drawing tool. Currently, the reconstruction of the morphology into Visio is a manual process.

2.2 Excavating the ontology

There are many representations for ontologies that typically support data modeling and database exchange activities [13-15]. We've chosen a semantic network because the basic structure of a network, semantic or otherwise, consists of nodes and edges allowing us to use graph theory to analyze it. Thus, we can analyze the ontology in a domain independent fashion using graph

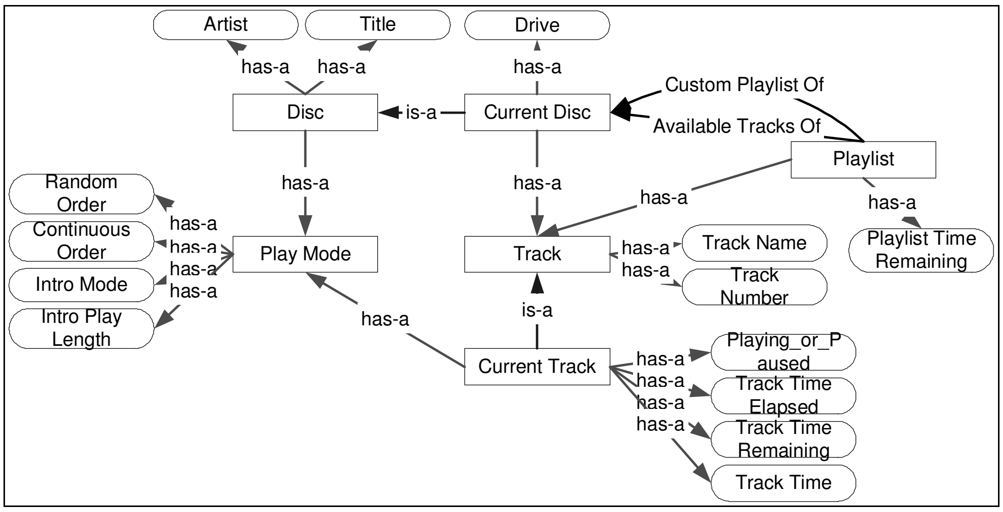

Figure 2 - Ontology for the Windows 95/98 CD Player

theoretic metrics developed for social network theory [16] and city planning [17]. These metrics enable us to identify what are candidate core concepts as well as potential subgroups of concepts that may have tightly linked functionality or importance.

Using the interface map as an information source, we first identify the concepts indicated by the labels attached to those elements, looking for noun phrases and the indirect objects implied by verbs, a process borrowed from object-oriented analysis methods [18]. For example, a “File Menu” implies that there is a concept of “File”. A “Font” dialog box informs the concept “Font Size”. In cases where a noun does not exist in the label, concept identification requires interaction with the system. “Play” on a CD Player plays a “Track” on a “Disc”. Once we’ve identified a concept, we also model those attributes and subtypes associated with it. For example, a “Disc” in the CD Player has an “Artist” and a “Title” as seen in Figure 2. Attributes are modeled as first class objects, similar to approaches used in some Entity-Relationship Models [19].

After identifying some candidate concepts, we identify the relationships between them. For constructing a semantic network, we use the basic relationships from object modeling: generalization (*is-a*), aggregation (*has-a*), and associations [20]. During this process, we also have to refine the concepts. In the CD Player example, we have to make a distinction between a track on the CD (“Track”) and the track being played (“Current Track”) and, likewise, a similar contrast between “CD” and the current CD being played (“Current CD”). Figure 2 shows the ontology for CD Player.

We don’t model any concepts that are specific to the operating system that runs the application, such as mouse movements, file handling, or printing capabilities. This also includes all functions and supporting applications that operate independently of the one being studied. For example, the Win 95/98 CD Player does have a volume command but it activates the Volume Control dialog box of the operating system so we don’t model this in the CD Player’s ontology.

Once identified, the concepts and relationships are modeled as boxes and arrows in Visio. We wrote a macro that takes this graph and puts it into an adjacency list representation which we use for our analyses.

2.3 Ontological metrics

We analyze the semantic network in its adjacency list form using UCINET, a software package designed for social network analysis [21]. UCINET allows the calculation of *centrality* for nodes in a graph. Centrality metrics assess the *importance* of a node to the rest of the graph. In social networks, nodes represent individuals or groups. A node with a high centrality measure could have some significance to the social network [16]. That node could represent a CEO or an executive of some sort. In an ontology, concepts with high centrality values should be good candidates for core concepts. However, different centrality metrics check for different structural attributes in the graph and not all will be suitable for our purposes. There are five different metrics that we examined.

- *Degree Centrality* measures the number of edges on a node. The more edges on a node, the higher the centrality.

- *Closeness Centrality* measures the average distance from that node to all other nodes.
- *Betweenness Centrality* measures the number of shortest paths between all pairs of nodes in the graph that use a particular node. The higher the centrality measure, the more dependencies on that node. Because leaf nodes only serve as start and end points for paths, they automatically have a betweenness value of 0.
- *Information Centrality* measures the information contained in all paths originating with a specific node.
- *Eigenvector Centrality* measures the centrality of a node relative to the importance of its surrounding nodes.

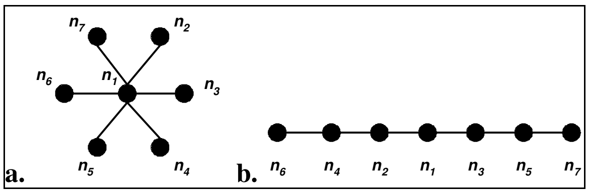

Figure 3 - a) Star Graph b) Line Graph

Each metric returns a normalized value from 0.0 to 1.0 for each node in the graph where 0.0 represents a node that is not central (a leaf node) and 1.0 represents a node that is completely central relative to the other nodes. Figure 3 has two examples to motivate these metrics [16]. In the Star Graph, n_1, the most central node in the graph has degree and betweenness centralities of 1.0 because of its direct connections to all other nodes in the graph. In the Line Graph, n_1, also the most central node, the degree centrality is only 0.333 as are all nodes with two edges (the two outer nodes have a degree centrality of 0.167). The betweenness centrality of n_1 is 0.6. The other nodes flanking n_1 have values of 0.533 because they are fairly central but not the most central. The nodes on the outside have a betweenness value of 0.0 because they do not fall on any shortest paths between pairs of nodes.

We also examined the ontologies using subgraph identification algorithms. Core concepts rarely exist in isolation and are often found in subgroups of related items. We hypothesized that these could be identified structurally and looked for specific subgraph types such as cliques and *k-cores*. A *k-core* is a connected, maximal, induced subgraph of nodes such that each node has a minimum degree greater than equal to *k* [22].

3. Case studies

We recovered the ontologies of three applications: Windows 95/98 CD Player, Palm Pilot Scheduler, and Windows Notepad. These were chosen partly for their simplicity but also because they represented implemented solutions to larger problem domains: media playing, scheduling, and word processing. We then analyzed their ontologies using each centrality metric and then analyzed the ontologies for connected subgroups. A summary of the applications can be found in Table 1.

Table 1 - Application comparisons

	# of nodes	# of non-leaf node concepts	# of k-cores
CD Player	21	6	1
Palm Pilot Scheduler	58	32	1
Notepad	78	31	3

3.1 Evaluating the Candidate Centrality Metrics

We first checked to see which concepts were identified by the centrality measures to see whether the concepts in the ontologies could be measured in this matter and also to see whether certain metrics were more effective than others in returning the core concepts. Each of the metrics in each application identified a slightly different subset of the concepts as being important with some differences. We examined the data to determine whether the metric returned concepts that seemed to be good candidates for being core concepts. In the absence of a preexisting structured representation for these specific applications, we used an *ad hoc* method for determining whether a concept truly was a core concept. Specifically, we ask ourselves the common sense question “Does the application require this concept to function?” For example, the Scheduler needs Event to be a scheduler but not necessarily Backup Copy. Therefore, Event is more likely to be a core concept then Backup Copy.

An interesting characteristic of the ontologies is that when they are visualized (using NetDraw, part of the UCINET Package [21]), they all have a cluster of nodes in the middle with ‘satellite’ notes around the periphery (see Figure 4) This seems to verify the intuitions behind the idea of a core concept that structures the ontology.

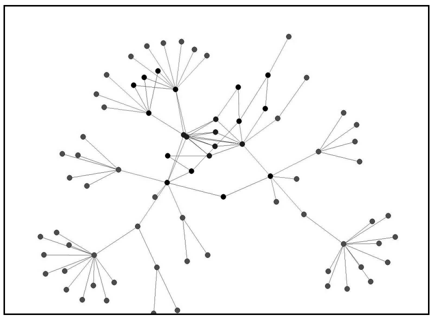

Figure 4 - Visualization of Notepad Ontology

Using this visualization and some deeper analysis of the metrics themselves, we were able to eliminate some of the metrics. For example, degree centrality measures are strongly affected by the number of attributes or subtypes that a concept possesses. It turns out that concepts with many different subtypes (for example, Script in Notepad) can skew the metrics even if they exist on the periphery. This depends, of course, on the size of the graph.

We also eliminated closeness and information centrality. The intuition informing this decision is that core concepts would stand out by having significantly higher values than the non-core concepts. When all the values are graphed (independent of where they are in the graph), we can see that closeness and information centrality are very "flat" relative to betweenness and eigenvector centralities. Those centrality metrics don't enable us to discriminate core from non-core concepts. This leaves betweenness and eigenvector centrality. Betweenness has the nice property of automatically eliminating leaf nodes. In addition, betweenness measures, by debatable degrees, succeeded in identifying what we considered to be the core concepts in each application.

3.2 Testing the "common sense" approach

To further test the metrics, we performed a series of tests on the data. The intuition is that a core concept makes a significant contribution to the underlying structure of the ontology. Therefore, removing a core concept should cause large changes to the measurements of the other concepts. We systematically removed each concept from the graph and recalculated the values in the new graph using all the centrality measures. This turned out to be very easy to do in Visio. To avoid the problem of disconnected graphs, as happened when a peripheral node with many attributes was removed, we simply removed attributes when we removed the node.

We calculated the average absolute values and sum-squared values of the difference between the centrality values of all the nodes in the original graph and the new graph. We then sorted the concepts according to the size deltas and checked to see where concepts ended up in the new rankings.

The results of the test showed that betweenness had the most consistent behavior in that the rankings of the concepts in the original graph very closely matched the rankings of the concepts produced from the difference calculations. This means that the concepts identified by betweenness centrality as being important had profound effects on the other nodes when they were removed from the graph.

The second issue was choosing a reasonable threshold that could act as a 'cutoff' point between core concepts and peripheral ones. We graphed the normalized centrality values sorted from lowest (least central) to highest (most central). As Figure 5 shows, betweenness measures returns a narrow range of values and automatically ignores leaf nodes. A simplistic analysis could arbitrarily decide that all concepts greater than 0 are core concepts. However, a concept on the periphery

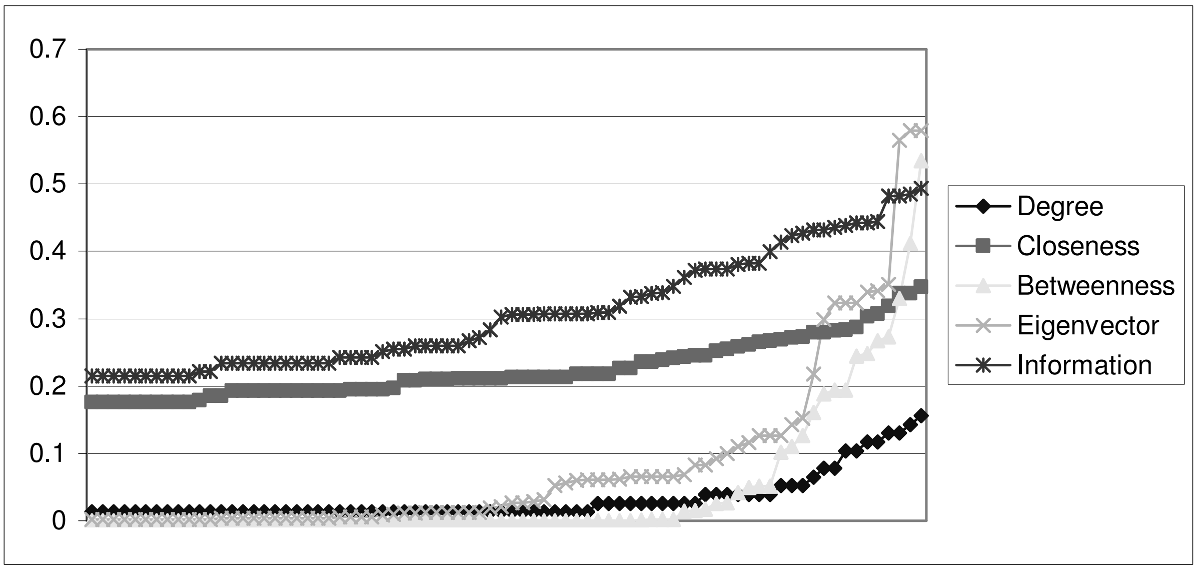

Figure 5 – Comparing the different metrics using the Notepad ontology

with one attribute will have a positive betweenness value but may not be a core concept. Therefore, for this study, we chose a reasonable cutoff point for each ontology based on the slopes displayed by their respective graphs to identify a set of candidate concepts (Table 2). The identified concepts seemed to be reasonably important in the context of their applications.

Table 2 – Candidate Core Concepts found in the 3 ontologies in order of their betweenness centrality values

Application	Candidate Core Concepts
CD Player	Current Track, Play Mode, Track, Disc, Current Disc, Playlist
Palm Pilot Scheduler	Event, Date, To Do Item, Hot Synch, Day, Month, Time, Alarm, Repetition, Note, Every
Notepad	Page Setup, Font [Setting], Paper, Text, Paper Size, Font, Script, Header, Footer, [Configuration], [Header/Footer Code], Margins, Alignment, Font Style

3.3 Subgroup Identification

We know that there are core concepts that make contributions to the same functions and hypothesized that these related concepts might form a cohesive subgroup that would be detectable in the ontology. These subgroups may resemble standard graph theoretic subgraphs such as cliques and clans. We examined the ontologies using the subgroup identification algorithms provided by UCINET but found them to be relatively ineffective, especially for small graphs.

What may be promising is a *k*-core analysis. A *k*-core is a maximal induced subgraph such that each node in the subgraph has edges connecting it to *k* or more nodes. When we performed a *k*-core analysis on CD Player and scheduler, we obtained 1 *k*-core consisting of many of the concepts identified by the betweenness centrality metric. Notepad shows multiple groups, 1 3-core and 2 2-cores as shown in Table 3.

Table 3 - k-cores in Notepad

k value	Concepts in the core
3	Text, Header, Footer, File Name, Page, Number, Date, Time
2 (a)	Header/Footer Code, Left/Right/Center Alignment, Alignment (of Header/Footer)
2 (b)	File, Current File, [Configuration], Line, Word, Font [Setting], Page Setup, Document, Page

What's interesting about the subgroups is that the 3-core group, the most tightly coupled by definition, has the one critical concept that you would expect to find in a text viewer such as Notepad – "Text" as well as some supporting ones which are major Notepad functions. Both 2-cores have sets of related functionality, 2-core (a) deals with the Header/Footer codes and alignment of Headers and footers and 2-core(b) deals with file handling and configuration of Notepad's viewing functions.

3.4 Robustness of the ontology

A potential weakness of this work is its dependence on the correctness of the recovered ontology. Theoretically, small errors in modeling should have limited or negligible impact on the core concepts. Thus, a single mismodeled concept or misplaced edge should not affect the basic clusters or centrality values of the application. We conducted a small experiment to test this idea. In the initial model that we generated for Notepad, the concept Ampersand appeared as a 2-core concept but didn't have any direct edges to either group.

Because Ampersand was ambiguous, we took a closer look at it. It turns out "ampersand" was modeled slightly incorrectly which explains why it was "isolated" from either group that were in the set of 2-cores. Ampersand is used in Header/Footer codes to print page numbers and so on. There is a special code "&&" required to print an Ampersand. We generated 4 alternate models based on varying modeling decisions to test the robustness of the basic ontology.

#	Modeling Decision	Result
1	Header and Footer has-a Ampersand (reflecting that it's specially printed as a Header/Footer code)	Ampersand is an isolate but identified as part of a 2-core
2	Header and Footer has-a Ampersand and Ampersand isa Text (reflecting that it's just a character that is part of Text)	Ampersand belongs to the 2-core group that concerns Header/Footer codes
3	Header/Footer Code has-a Ampersand, Header and Footer has-a Ampersand, and Ampersand is-a Text	Ampersand moves to the 1 3-core group which also makes Header/Footer Codes part of the 3-core..
4	Ampersand isa special character code.	Ampersand now disappears from the 2- and 3-cores and becomes a leaf node

This exercise demonstrates that a wrong modeling decisions can move a single concept from the periphery to the core. However, it's important to also point out that other subgroups were not affected in the sense that there were still 2 2-cores and 1 3-core throughout each of the models. So small errors will not perturb the basic structure of the ontology but can cause the erroneous promotion of a concept to a core concept.

4. Discussion

There are several potential problems with this methodology that need to be addressed.

First, the amount of time required to recover the ontology of an application may be problematic for very large applications with many features. The cost of manually producing an interface map in addition to the work required to ensure the correctness of the ontology may be high depending on the complexity of the interacting elements. Recovering an interface map from an application can be solved using white-box reverse engineering methods on the sections of the code that implement the user interface [23]. In cases where the source code is unavailable, Stroulia's approach to recovering legacy interfaces offers a potential avenue for the semi-automated recovery of the user interface by tracking the interactions of a user with the interface. [24-26]. Furthermore, because this work recovers specifications based on frequently occurring interaction patterns, it may provide a viable next step to this line of research which is to assess the overall match of the morphology to its underlying ontology. Currently, the automated reverse engineering of a domain is still an unsolved problem. Thus, even with the automatic recovery of an interface map, modeling the application's ontology will still require human intervention.

Second, the consistency and accuracy of the ontological model may vary from person to person and depends wholly on their level of modeling knowledge and skill. Systematic errors in the excavation process will likely produce a very different and erroneous ontology. As we illustrated in our case study, the ontology seems to be fairly robust against small errors or variations in modeling so one will be able to identify the core concepts. Poor modeling decisions are still a problem in any data modeling activity. One of the interesting side effects of this methodology is that core concepts, by their very nature, tend to have more dependencies and attributes which make them more visible to a betweenness centrality analysis. Forgetting to include a relationship to these core concepts should not affect their overall visibility. We believe that this resistance to errors, even systematic ones, only improves as the size of the ontology increases in the number of concepts and attributes that it contains.

In addition to the methodological issues, some of which may be overcome by limited automation when the source code is available, there is the issue of external validation that was not addressed in our case study. Does betweenness centrality actually identify the most important concepts in the ontology? Another potential danger is that recovering concepts from the user interface does not guarantee that all the real world domain concepts understood by the application are properly excavated. For example, security and privacy policies may affect how data is retrieved or connected and only reveals itself in emergent behavior not readily understandable through simple interactions. In our studies, we have applied an *ad hoc* common sense assessment based on our knowledge of the domain. Given the simple and constrained nature of the domains we examined, this could be no worse than asking domain experts to verify the accuracy of the model. But in addition to lacking rigor, this type of validation will be very unsatisfactory with larger and more complex domains. Part of our future work will be to identify better techniques for validating these models, possibly through a combination of existing domain reverse engineering methods.

In spite of these potential shortcomings, ontological engineering shows tremendous potential for contributing, not only to software maintenance, but other areas of software engineering. It provides a method for reverse-engineering an ontology that focuses solely on the real world concepts that the application understands. The semantic network representation allows this domain-independent analysis using graph theory to identifying potential core concepts. The same representation and analysis process will also aid the study of application evolution. By reverse-engineering several versions of a product line, one can examine how the ontology has changed, which concepts have migrated from periphery to core, which subgroups of concepts have formed in recent implementations, and which concepts are no longer being maintained. The same kind of analysis can be coupled with requirements analysis to predict how an application needs to evolve in future versions to meet the demands of its users.

5. Conclusion

We have developed a methodology for excavating an application's ontology and shown how graph theoretic metrics can be used to identify those core concepts in the ontology that can have a significant impact on the application's evolution. Another promising result may be the ontology's structure allows the identification of conceptual subgroup(s) within the ontology. Conceptual subgroups represent another type of ontological organization that will have to be accounted for by developers when designing new features for the application.

Earlier in the paper, we made a strong claim that understanding an application's ontology can benefit developers in the software maintenance phase and can enable them to avoid designing mismatches between an application's services and user expectations. We have only shown the first step of that effort; the recovery of an ontology from the application in question. A complete analysis that would benefit software developers would have to include a corresponding model of the real world domain and of the user's conceptual model. Obtaining and modeling both of these remains a difficult problem and will be addressed in our later work. Ideally, we would like to be able to compare sets of core and peripheral concepts between the application ontology and the domain ontology as well as measure the ontology's correspondence to the morphology. We believe that this ontological approach to software evolution can allow developers to maintain a reasonable 1:1 correspondence between application and domain ontologies and aid the design of the software architecture to accommodate future extensions without compromising the existing structures. However, this will have to be shown and demonstrated in future work.

Currently, we are using ontological excavation as part of the MesoMORPH project [27]. MesoMORPH is a system designed to support the activity of meso-adaptation, the adaptation of a system for a different context than the one the system was originally designed to support. Changes to the system in this intermediate or meso-layer between reengineering and user-driven customization, ranges from modifications of the user interface to match a different set of user capabilities or environments (e.g. augmenting a system with assistive technologies for disabled users) to altering the underlying system architecture by adding or subtracting system components to match a different set of hardware constraints. Ontological excavation informs this context-driven evolution by providing metrics that aid MesoMORPH in determining which concepts must remain (because they are core concepts) and which ones can be removed (because they are not required by the new context).

6. Acknowledgements

This work was sponsored by DARPA as part of the DASADA project under Dr. John Salasin, project number F30602-00-02-0516. We would also like David Yu for his contribution to the technical aspects of this work.

7. References

[1] M. Lehman and L. Belady, *Program Evolution: Processes of Software Change*, 1st ed. Orlando: Academic Press,inc., 1985.
[2] M. Lehman, "Laws of software evolution revisited," in *Proc. 5th European Workshop on Software Process Technology*, 1996, pp. 108-124.
[3] J. McGrenere, ""Bloat": The Objective and Subjective Dimensions," in *Proc. Computer Human Interaction 2000 (CHI 2000)*, 2000, pp. 337-338.
[4] I. Hsi and C. Potts, "Studying the Evolution and Enhancement of Software Features," in *Proc. Intl. Conf. Software Maintenance*, 2000, pp. 143-151.
[5] S. Rugaber, "Position Paper Domain Analysis and Reverse Engineering," in *Proc. Software Engineering Techniques Workshop on Software Engineering*, 1994.
[6] G. Arango and R. Priéto-Diaz, "Domain Analysis Concepts and Research directions," in *Domain Analysis and Software Systems Modeling*, R. Priéto-Diaz and G. Arango, Eds. Los Alamitos, CA: IEEE Computer Society Press, 1991, pp. 9-26.
[7] R. d. A. Falbo, G. Guizzardi, and K. C. Duarte, "An Ontological Approach to Domain Engineering," in *Proc. International Conference on Software Engineering and Knowledge Engineering (SEKE'02)*, 2002, pp. 351-358.
[8] N. Anquetil, "Characterizing the informal knowledge contained in systems," in *Proc. Eight Working Conference on Reverse Engineering*, 2001, pp. 166-175.
[9] S. J. Greenspan, J. Mylopoulos, and A. Borgida, "Capturing More World Knowledge in the Requirements Specification," in *Domain Analysis and Software Systems Modeling*, R. Priéto-Diaz and G. Arango, Eds. Los Alamitos, CA: IEEE Computer Society Press, 1991, pp. 53-62.
[10] J. M. DeBaud, B. Moopen, and S. Rugaber, "Domain Analysis and Reverse Engineering," in *Proc. International Conference on Software Maintenance*, 1994, pp. 326-335.
[11] R. Clayton, S. Rugaber, and L. Wills, "Dowsing: a tool framework for domain-oriented browsing of software artifacts," in *Proc. 13th IEEE International Conference on Automated software Engineering*, 1998, pp. 204-207.
[12] G. Arango, "Domain Analysis Methods," in *Software Reusability*, W. Schafer, R. Priéto-Diaz, and M. Matsumoto, Eds. Chichester, England: Ellis Horwood, 1994, pp. 17-49.
[13] R. J. Brachman, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Resnick, "Living with CLASSIC: When and How to Use a KL-ONE-Like Language," in *Principles of Semantic Networks*, J. Sowa, Ed.: Morgan Kaufmann Publishers, 1990.
[14] T. R. Gruber, "Ontolingua: A Mechanism to Support Portable Ontologies," Stanford University, Technical Report, June 1992.
[15] D. B. Lenat, "CYC: A Large-Scale Investment in Knowledge Infrastructure," *Communications of the ACM,* vol. 38, pp. 33-48, 1995.
[16] S. Wasserman and K. Faust, *Social Network Analysis*. Cambridge: Cambridge University Press, 1994.

[17] B. Hillier, *Space is the Machine: A Configurational Theory of Architecture*. Cambridge, UK: Cambridge University Press, 1996.

[18] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-oriented Modeling and Design*. Englewood Cliffs, NJ: Prentice-Hall, 1991.

[19] R. Elmasri and S. B. Navathe, *Fundamentals of Database Systems*. New York, NY: Addison-Wesley, 1994.

[20] G. Booch, *Object-Oriented Analysis and Design*. Reading, MA: The Benjamin/Cummings Publishing Company, Inc., 1994.

[21] S. P. Borgatti, M. G. Everett, and L. C. Freeman, *UCINET 5.0 Version 1.00*: Analytic Technologies, 1999.

[22] M. G. Everett and S. P. Borgatti, "Peripheries of Cohesive Subsets," *Social Networks*, pp. 397-407, 1999.

[23] M. Moore, "User Interface Reengineering," in *College of Computing*. Atlanta, GA: Georgia Institute of Technology., 1998.

[24] M. El-Ramly, E. Stroulia, and P. Sorenson, "Recovering Software Requirements from System-user Interaction Traces," in *Proc. 14th International Conference on Software Engineering and Knowledge Engineering*, 2002, pp. 447-454.

[25] E. Stroulia, M. El-Ramly, and P. Sorenson, "From Legacy to Web through Interaction Modeling," in *Proc. International Conference on Software Maintenance*, 2002, pp. 320-329.

[26] E. Stroulia and R. V. Kapoor, "Reverse Engineering Interaction Plans for Legacy Interface Migration," in *Proc. 4th International Conference on Computer Aided Design of User Interfaces*, 2002, pp. 295-310.

[27] M. Moore, C. Potts, I. Hsi, and D. Yu. ,). The MesoMORPH Project.
www.cis.gsu.edu/~mmoore/MesoMORPH [Online]. Available: