Using Domain Ontologies in a Dynamic Analysis for Program Comprehension

Javier Belmonte, Philippe Dugerdil
Department of Information Systems
HEG-Univ. of Applied Sciences
Geneva, Switzerland
{javier.belmonte@hesge.ch, philippe.dugerdil}@hesge.ch

ABSTRACT

The use of domain knowledge for program comprehension has been advocated by many authors. However, as far as we know, most of the analysis techniques using domain knowledge are static, it seems that dynamic analyses have not yet taken full advantage of any domain knowledge. This might be a consequence of ontologies, the most common technique for domain knowledge representation, being static by nature. In this article we present a new kind of dynamic analysis that attempts to use domain knowledge from two ontologies: that of the domain concepts and another one we called the "Ontology of Domain Actions". To take advantage of this later source of knowledge, we had to specify what actions were expected to be performed by the software at any moment in time. This has been done using a variant of the CRC cards formalism. As a result, we are able to match the actions actually performed by the software with expectations using dynamic analysis based on the action ontology.

Categories and Subject Descriptors

D.2.7 [**Distribution, Maintenance and Enhancement**]: Restructuring, reverse engineering and reengineering

General Terms

Documentation, Measurement, Theory

Keywords

program comprehension; dynamic analysis; domain knowledge; ontology

1. INTRODUCTION

Program comprehension has been a hot topic in software engineering for more than three decades with pioneering work in software psychology [16]. Nowadays a large set of papers published in specialized conferences present techniques claimed to help software engineers with "program comprehension". However few of them define precisely what kind of comprehension they refer to. In fact, the variety of software comprehension models that have been published for the past three decades accounts for the variety of meanings of the expression "program comprehension".

As early as 1983, Brooks proposed that "program understanding" be defined as the process of re-creating the links between the domain problem and the program code by hypothesis generation, refinement and validation [3]. As of 1995, the main theories of program comprehension for maintenance were analyzed by Mayrhauser and Vans who proposed a program comprehension metamodel [18]. The authors explained that top-down hypothesis generation should sometimes be complemented by bottom-up program analysis.

These early works highlighted the need for domain knowledge to be explicitly taken into account in program understanding. This vision has since gained an increasing acceptance in the software engineering community [15]. In the mid 90's Biggerstaff, Mitbander and Webster introduced the term "concept assignment" [2] to name the search and assignment of "human-oriented" concepts to the elements of the program code. The authors explain that during program understanding the software engineer would discover and interrelate informal "human-oriented" concepts step by step to build an understanding of the program (i.e. create a mental model [18][14]). In fact, this vision is not different from that of Brooks [3]. However, since the notion of "concept" is not precisely defined in the article, many researchers have since worked on the "concept assignment problem" while speaking about widely different things. Rajlich and Wilde recognized this problem and presented a way "concepts" can be represented in programs and the role they play in program comprehension [13].

In summary, it is clear that any work in program understanding today should include a statement on what kind of "understanding" is meant and in case the term "concept" is used, what it means. Without such a reference, it is impossible to assess the scientific contribution of the work. Our definition of program understanding rests on the statement of Biggerstaff et al. [2]:

> *"A person understands a program when able to explain the program, its structure, its behavior, its effects on its operational context, and its relationships to its application domain in terms that are qualitatively different from the tokens used to construct the source code of the program."*

This statement let us draw two important insights. First, the understanding of something is a capacity to explain some

ODSE'10 October 17-21, Reno, NV USA

of its characteristics (interestingly, this statement is backed up by research in pedagogy [5]). Second, the explanation must be given with respect to another knowledge domain we are supposed to know already (the application domain in this case). Therefore, in our definition, we take for granted that the maintenance engineer knows the application domain of the program he is trying to understand. For example, in the banking domain, he would be able to explain what an interest rate is, how it is computed, in what banking operation it is used and so on.

1.1 Goals

With this hypothesis in mind, program understanding is, in our work, the capacity to build a coherent set of relationships between the constructs of the program and the relevant elements of the application domain. For a particular program, the relevant elements of the domain are identified through the use-cases and the corresponding domain model (in the sense of the Unified Process [8]), thus making program understanding the process of linking these elements to the program code. Because linking a text-based description directly to the program is not a trivial task, the maintenance engineer must build an intermediate technical model. In our work, this is the role of the robustness model [1, 8] and its associated objects' responsibilities. The interpretation of such a model (together with its links to the use-case and domain model) is no trouble for the maintenance engineer as it is actually him who re-designs them from the recovered use-cases (cf. section 2).

In this article, the responsibilities associated to each robustness model object constitute the relevant elements of the application domain. The goal of our work is to map each of these responsibilities to the source-code classes that implement them. The choice of the implementation classes as the source-code constructs to be mapped to robustness objects' responsibilities was made because both the number of classes and the number of responsibilities participating in a use-case scenario are in the order of $O(100)$. Therefore we can hope to be able to map a few number of responsibilities to a few number of implementation classes, assigning them a domain-related meaning.

In summary, the "program understanding" process that we propose is represented by the creation of the links between the source code classes and the individual responsibilities of analysis objects. Since the latter can be derived from the system use-cases, this truly represents a link between the code and its purpose in the business domain.

1.2 Outline

In section 2 we will briefly describe the steps our reverse-engineering methodology goes through before applying the dynamic analysis technique. We also present the way in which we formally describe responsibilities. The formalization of the responsibilities rests on the concepts structured as two ontologies that will be described in section 3. In section 4 we explain our technique for measuring the likelihood for a source code method to implement the basic components of a formal responsibility description. Section 5 briefly describes the dynamic analysis technique using the likelihood measure from section 4 to link each responsibility to the methods implementing it. Section 6 describes related works and section 7 concludes our article.

2. THE METHODOLOGY

The reverse-engineering methodology we use was introduced in detail in [6]. In this methodology we assume the worst-case scenario for program comprehension: the absence of any documentation on the system. Moreover, the original developers are also supposed to be absent in this worst-case scenario, leaving the users of the system as the only ones with a good perspective on the system. In fact users are usually well aware of the business context, business functionalities and business relevance of the part of the software they interact with. Accordingly, by a process comparable to the initial capture of requirements of the system, our methodology starts by re-documenting the use-cases by interviewing its actual users. Then, the robustness models are rebuilt from the use-cases using well known analysis heuristics [1][8].

The original methodology re-documents the use-cases up to their robustness model. However, since our goal is to individually map robustness objects' responsibilities, the robustness models are not enough, we must re-document the objects' Class-Responsibility-Collaboration (CRC) cards. Furthermore, not only are these responsibilities documented individually, but also we request them to be attached to the use-case steps in which they are expected to be involved. We refer to the resulting model as "Enhanced Use-Case Flow" (EUCF).

The association of responsibilities to individual steps of the use-case implies a temporal distribution of those responsibilities. In other words it specifies a sequence in which those responsibilities need to be carried out to perform a particular use-case scenario. The next logical step is then to compare this temporal distribution with an execution trace gathered from the execution of the use-case scenario by system (cf. section 5.1).

2.1 Describing responsibilities

To take advantage of the knowledge in the EUCFs, we need to express responsibilities in a way that can be used by an automaton, ruling out any description made in Natural Language (NL). We decided to tackle this by injecting some business knowledge into our analysis, this knowledge should answer the following question: *how would the system's users or domain experts carry out the responsibilities without the system?*

Given that systems are normally implemented to fulfill a set of requirements captured by asking its future users what they expect the system to do for them (while abstracting as much as possible the actual way it could do it [8]), we assume that the above question can be answered in most situations. In fact, since the requirements and responsibilities to be carried out by the system should be independent from the actual implementation, we can expect the answers to the question above to use business domain knowledge only.

To describe how a responsibility is carried out, system users or domain experts will decompose it in sub-steps. In our case we want these sub-steps to correspond each to a single action, which is to be applied to a business entity to contribute to the responsibility's fulfillment. We decided to formalize these sub-steps, that we called *Atomic-Actions*, as composed of an *action* and a *business concept*. From a linguistic point of view, these pairs correspond to simple verbal phrases composed by a verb and a noun playing the

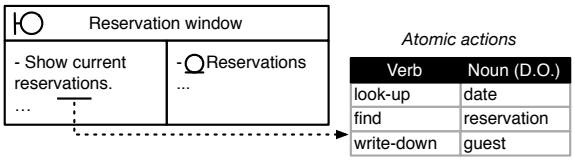

Figure 1: Set of atomic-actions describing the "Show current reservations" responsibility.

role of the verb's Direct Object (DO).

Therefore, responsibilities will be formalized as sets of atomic-actions. To illustrate this, we chose to analyze jHotel[1], an open source hotel management tool. One of the robustness model's objects we found while documenting the system's use-cases is the boundary object "Reservation window". This object has, among others, the following responsibility: "Show current reservations". Briefly, this corresponds to showing, in a calendar, all the reservations made for the current month. The description of the business domain process in NL could be:

> *"We **look up for every date** for which we can **find a reservation** and we **write down the guest's information** in the corresponding calendar date-box."*

Figure 1 shows the set of atomic-actions describing the responsibility "Show current reservations".

3. KNOWLEDGE REPRESENTATION

Both parts of the atomic-actions are represented together in the form of a verbal phrase in NL. Therefore, we decided to analyze them using the same tool: an ontology. Ontologies are nowadays a popular knowledge representation technique; one of the reasons for this is that while being coded in a machine-readable form, they represent real-world knowledge as it is usually understood by humans.

However, because of the different nature of the atomic-actions parts (verb vs. noun), the disponibility of a preexisting ontology allowing us to analyze each part is not the same. Indeed, although domain concept ontologies for the noun part of the atomic-actions can usually be found, we have not been able to find an ontology representing verbs or actions, which forced us to create one. This absence of verbs in business ontologies is a consequence of the fact that verbs are not, in their majority, domain specific. In fact, verbs can not, in general, carry by themselves a domain-specific action. For example, in the phrase "register a new guest" (from the hostelry domain), the verb "register" has by itself multiple meanings (the most general being the action of putting something in a register), but, although it may be frequently used or otherwise important in some domain, it can not be said to belong to any business domain if it can also be used elsewhere: Libraries, Education Institutions, etc. It is only when associated with a particular domain concept: "guest" (from the hostelry domain), that a verb becomes domain specific.

To analyze the jHotel application (section 2.1 and figure 1) we looked for an ontology of the hostelry domain. Although we didn't find a single ontology comprising the whole domain[2], we found a set of ontologies that could be used together to represent most of the needed domain concepts: a hotel description ontology (*HDeO* [19]), a temporal concepts ontology (*Time Ontology* [7]) and an ontology of persons (*Person Ontology [10]*).

[1]http://sourceforge.net/projects/jhotel/

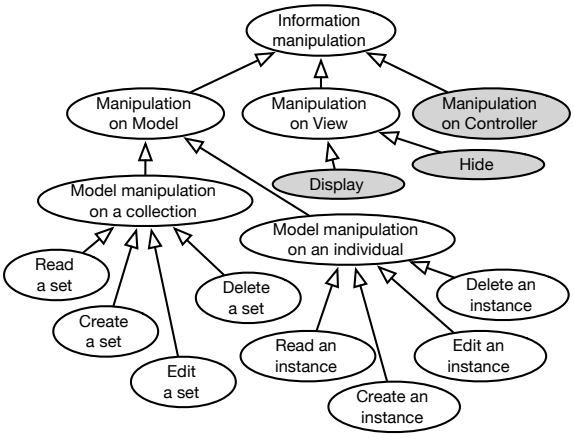

Figure 2: Update on knowledge production and application.

3.1 Ontology of Domain Actions (ODA)

This subsection describes the construction of the task ontology we had to develop to analyze the verb in atomic-actions. We refer to this ontology as the "Ontology of Domain Actions" (ODA). Figure 2 shows this ontology in its current state, all of the concepts therein depicted will be introduced in the following paragraphs. Greyed out are concepts we plan to analyze and specialize in the near future.

A general characteristic of the actions that we want to represent in this ontology is that they all represent information manipulation. Indeed, unlike real actors, systems can only act on the information they hold, then, no matter which of the system's responsibilities we choose to describe, the actions used in their descriptions will represent some kind information manipulation. Being general, this characteristic is reflected on the root concept of our ontology: `Information manipulation`.

Most information systems (current and past) conform to the widely used Model-View-Controller (MVC) pattern. We believe that actions describing the responsibilities carried out by the system will also conform to this pattern, which means that actions can be characterized as handling input-output of information, recording the information that is being handled by the system or controlling the overall process execution. This characterization is reflected on a layer of concepts, all subsumed by the root concept: `Manipulation on Model`, `Manipulation on View` and `Manipulation on Controller`.

The information carried by a noun used in an atomic-action can be further distinguished as single instance or as a set of instances of the related Business Concept. Therefore, `Manipulation on Model` concept is further specialized in:

[2]A large specification of the hostelry domain is maintained by the Open Travel Agency (http://www.opentravel.org). Nevertheless, it is not represented as a proper ontology.

`Model manipulation on a collection` and `Model manipulation on an individual`. Next, inspired by the Create, Read, Update and Delete (CRUD) taxonomy of database manipulations [9], we define the specialization of the two latest introduced concepts.

Currently, we feature only one specialization of the `Manipulation on View` concept. It is based on the idea that manipulations of this type will either show information or hide it. The `Manipulation on Controller` concept has not yet been thoroughly analyzed.

3.1.1 Verbs and the ODA ontology

The ODA ontology represents all the different kinds of information manipulation that can be represented by atomic-actions, however, because atomic-actions are documented by NL verb phrases, we need to link the NL verbs to the instances of the ODA ontology concepts that represent them. We create these links manually, assuming that they will be highly reusable because verbs are in their majority unspecific to any particular domain.

4. BRIDGING A KNOWLEDGE GAP

The knowledge gap to which we refer in this section is the one between:

- The *informal knowledge* represented by the NL atomic-actions we use to create a description of how analysis objects responsibilities can be carried out manually.
- The *formal knowledge* represented by the source code classes.

Since traceability links exist from atomic-actions and responsibilities, bridging this gap, by recovering the links between atomic-actions and classes would represent the mapping we set us as final goal. However, our context being that of dynamic analysis and therefore based on the study of program execution traces, we must map responsibilities to source code methods before being able to map them to source code classes.

In fact, the bridging process links a method to an atomic-action through the likelihood for the method to implement the atomic-action. The computation of this likelihood can be divided in two phases: first, we create an intermediary representation of the informal knowledge, second, we use this representation to analyze the source code methods (formal knowledge) and approximate the probability for it to implement a particular atomic-action (the metric).

4.1 Intermediary representation

The intermediary representation we build from the NL terms used in the atomic actions is a map connecting those terms to the ontology concepts (or instances in the case of the ODA) that represent them. In both cases (verbs and nouns), two steps are necessary: first, we attach to every concept a set of the terms that are associated to it, then, we map the NL terms used in the atomic-actions to the concepts whose sets of terms contain them.

The construction of those sets of terms is slightly different for the two ontologies we use:

Business ontology The set of terms is automatically built by extracting all the terms we can identify in the name of a concept and in the names of its object and data properties.

ODA The set of terms is manually built as one fo the data properties of the ODA concept instances (cf. 3.1.1).

In both cases, we need to pay attention to the possible terms' inflexions. To avoid part of the problem we currently use the Snowball API[3], a stemmer API developed by Dr. Martin Porter. However, to simplify this problem we decided to restrict all NL verbs to the infinitive form.

4.2 Computing the metric

The second phase of our technique is the approximation of the probability that a source code method implements a particular atomic-action. This computation will be done by combining the knowledge of the business concept mapped to the NL noun in the atomic-action (knowledge extracted from the business domain ontology) and the knowledge of the ODA concept instance mapped to the atomic-action's NL verb (knowledge extracted from the ODA).

4.2.1 About the business domain concept

The knowledge about the business domain concept that we extract from the business ontology is the set of terms we built for the intermediary representation (cf. section 4.1). We consider this to be the set of terms that could be used in the source code to refer to the concept.

4.2.2 About the ODA concept instance

The mapping from the knowledge of the ODA and the source code rests on the hypothesis that each information manipulation, represented by an ODA concept instance, can be identified by a particular set of code beacons, in the form of syntax configurations belonging to a particular programming language.

Such an information will be manually attached to the ODA concepts instances simultaneously to linking of NL verbs described in section 3.1.1. Successive additions to these beacon sets are expected to be necessary in the future because it is not easy to build an exhaustive list of such syntax configurations. Once more, the effort of attaching information to each ODA instance is expected to pay off because these actions are not specific to any particular domain; hence, the references made from its actions to particular programming language elements will be reusable across programs written in that same programming language.

The set of beacons identifying the ODA concept instance is the knowledge to be extracted from this ontology.

4.2.3 Knowledge combination and computation

The knowledge extracted from both ontologies needs next to be combined into a metric approximating the probability for a source code method to implement an atomic-action. Our technique is the following:

- Find, all the source code fragments within the method that correspond to at least one of the beacons attached to the identified ODA concept instance.
- Count, within those source code fragments only, all the occurrences of the terms belonging to set extracted from the business domain ontology. This count is later rescaled per atomic-action so that the method with the highest account is given 100% as the value resulting from the application of the metric.

[3]http://snowball.tartarus.org

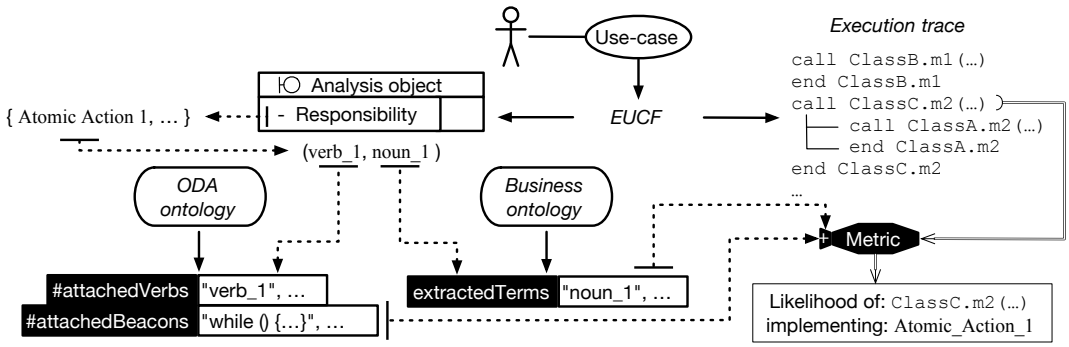

Figure 3: View of the ontologies as they bridge the knowledge gap.

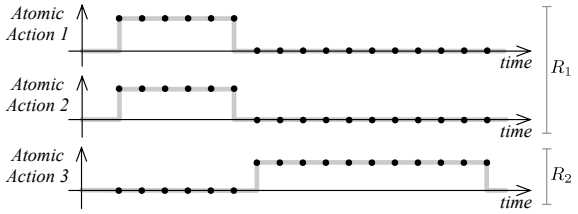

Figure 4: Atomic-action curves in a hypothetical perfect case.

By consolidating the formal knowledge we obtained from both ontologies in a single technique, we are able to identify generic actions in the source code corresponding to a domain specific action applied on a particular business concept. In a sense, once these actions become domain specific they correspond precisely to the definition of atomic-actions. Figure 3 summarizes the knowledge extraction, combination and the computation of the metric.

5. RESEARCH IN PROGRESS

The additional steps needed to reach our goal to map responsibilities to source code classes are still under development. However, in the subsections that follow, we will describe a particular process by which we should be able to go from the metrics to the goal.

5.1 Dynamic analysis

The process starts by applying the metrics on every method of a single execution trace. This results in a set of curves, one per atomic-action, showing the evolution of the probability for each atomic-action to be implemented by each of the methods called in the execution trace. Abstracting ourselves from the discrete view of the method calls, we can consider these curves as being continuous. They would show the evolution of the involvement of the atomic-actions in the execution of the use-case flow that generated the execution trace.

In an hypothetical case, where the source code beacons and the resulting metrics would be capable of unmistakably identifying the methods implementing the atomic-actions, these curves would easily let us split the execution trace in segments representing each the fulfilment of a responsibility. Indeed, we would simply have to see where all the atomic-actions defining a responsibility take place simultaneously. This is illustrated in Figure 4. Let us have responsibility R_1 characterized by the Atomic-Actions 1 & 2 and responsibility R_2 characterized by Atomic-Action 3. Their identification as segments of the execution trace's timeline is quite evident.

However, because our beacons can not be exhaustive, our metric is not perfect. Therefore, we can not expect our curves to flawlessly fit the sequence of responsibility fulfillment hidden in the execution trace. In the following section we introduce clustering, as a general algorithm that could satisfy this idea of computing a sort of "intersection" between real-world curves.

5.2 Clustering

If we come back from the abstraction we made in the previous subsection and consider the execution trace as a set of discrete moments in time (moments defined by the method calls), the segmentation process we proposed becomes a clustering process in which the resulting clusters would correspond each to the method calls carrying out a different responsibility. Indeed, by computing the chances for each atomic-action to be implemented by each method in the execution trace, we define a multidimensional space whose dimensions are the time and each of the atomic-actions. We define this space as $\mathbb{A} : \mathbb{R}^{n+1}$, where n is the number of atomic-actions and the additional dimension is that of time.

In $\mathbb{A}$, every method call is represented by a vector whose coordinates are the probability values computed by the atomic-action metrics and the moment in time at which the call was made. Moreover, responsibilities can be translated as subspaces of $\mathbb{A}$, subspaces defined only by the atomic-actions used to describe each responsibility. This is a convenient transformation because, by focusing on these subspaces of lesser dimensionality we are able to guide the clustering algorithm, therefore reducing the difficulty of the problem.

6. RELATED WORKS

The domain to which our research contributes is dynamic analysis for program comprehension. Recently, Cornelissen et al. [4] made a survey of the works published in this domain for the last ten years. We looked through that survey to check if business domain knowledge was used to perform dynamic analysis. But we couldn't find anything. In fact, to

the best of our knowledge, no one else explicitly processed domain knowledge in dynamic analysis for program comprehension. One of the biggest issues with execution trace analysis is their large size. Then, many analysis approaches try to compress or summarize these traces before processing for example [12][17]. Since the same summarization process is applied to the whole execution trace, there is a big risk of over-generalization. In contrast, our approach tries to summarize method calls by regrouping those that implement each object responsibility carried out. In that case, the summarization phase corresponds to the use of formal descriptions, making a big difference because, by automatically taking into account the characteristics of the execution trace, we are able to do a case by case summarization, reducing the risk of over-generalization. The use of curves in the analysis of execution traces is also the case in [11]. In that work, the metric used to produce the curves is the depth of the method call in the calling hierarchy captured by the execution trace. Like in our approach, the value transitions in those curves are used to identify the segments of the trace that can be considered as single elements; however, their approach does not analyze terms in programs, only the shape of the calling tree.

7. CONCLUSION

Although still under development, we believe our ontology based dynamic analysis technique to have the potential to successfully close the knowledge gap between the human-oriented informal knowledge and its formal, implementation-based, counterpart. We are convinced not being too far from achieving our final goal of mapping object responsibilities to the source code fragments implementing them. According to our definition of program comprehension, this achievement will indeed let the maintenance engineer understand the program since the fragments of the source code will be linked to fine grained business-oriented concepts and actions. This achievement goes far beyond the mere program visualisation techniques in fashion today, where only the structure of the code is displayed without reference to its business-oriented semantics. In a sense, by mapping human-oriented concepts to the source code components that implement it, our automaton will achieve what a software engineer could do mentally while building up his/her understanding of a program.

8. ACKNOWLEDGMENTS

We gratefully acknowledge the financial support from the Swiss Confederation, HESSO project N°G20539620034.

9. REFERENCES

[1] S. W. Ambler. *The object primer: agile model-driven development with UML 2.0*. Cambridge University Press, March 2004.

[2] T. J. Biggerstaff and B. Mitbander. Program understanding and the concept assignment problem. *Comm. of the ACM*, 37(5):72–82, May 1994.

[3] R. Brooks. Towards a theory of the comprehension of computer programs. *Intl. J. of Human-Comptuer Studies*, 18(6):543–554, June 1983.

[4] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering*, 35(5):684–702, April 2009.

[5] A. de La Garanderie. *Comprendre et Imaginer (To Understand and to Imagine)*. Edition du Centurion, Paris, January 1987.

[6] P. Dugerdil. A reengineering process based on the unified process. *Proc. 22nd IEEE International Conference on Software Maintenance*, pages 330 – 333, September 2006.

[7] J. R. Hobbs and F. Pan. Time ontology in owl. `http://www.w3.org/TR/owl-time/`, September 2010.

[8] I. Jacobson, G. Booch, and J. Rumbaugh. *The unified software development process*. Addison-Wesley, January 1999.

[9] H. Kilov. From semantic to object-oriented data modeling. *Proc. 1st International Conference on Systems Integration*, January 1990.

[10] D. King and M. Hall. Person ontology. `http://orlando.drc.com/semanticweb/daml/ontology/person/person-ont`, September 2010.

[11] A. Kuhn and O. Greevy. Exploiting the analogy between traces and signal processing. *Proc. 22nd IEEE International Conference on Software Maintenance*, pages 320–329, 2006.

[12] J. Quante and R. Koschke. Dynamic protocol recovery. *Proc. 14th Working Conference on Reverse Engineering*, pages 219–228, 2007.

[13] V. Rajlich and N. Wilde. The role of concepts in program comprehension. *Proc. 10th Intl. Workshop on Program Comprehension*, pages 271 – 278, June 2002.

[14] S. Rugaber. Program comprehension. *Encyclopedia of Computer Science and Technology*, 35(20):341–368, January 1995.

[15] S. Rugaber. The use of domain knowledge in program understanding. *Annals of Software Engineering*, 9(1-4):143–192, May 2000.

[16] B. Shneiderman. *Software psychology: human factors in computer and information systems*. Winthrop Publishers, January 1980.

[17] M. Smit, E. Stroulia, and K. Wong. Use case redocumentation from gui event traces. *Proc. 12th European Conf. on Software Maintenance and Reengineering*, pages 263–268, 2008.

[18] A. von Mayrhauser and A. M. Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, 1995.

[19] D. Yoo. Hotel-domain ontology for a semantic hotel search system. `http://donghee.info/research/jitt/hotel/`, September 2010.