An Ontological Software Comprehension Process Model

W. Meng[1], J. Rilling[1], Y. Zhang[1], R.Witte[1], P. Charland[2]

Department of Computer Science[1]
and Software Engineering
Concordia University, Montreal, Canada
{w_meng,rilling,yongg_zh,rwitte}@cse.concordia.ca

System of Systems Section[2]
Defence R&D Canada Valcartier
Val-Bélair, Canada
philippe.charland@drdc-rddc.gc.ca

Abstract

Comprehension is an essential part of software maintenance. Only software that is well understood can evolve in a controlled manner. In this paper, we present a formal process model to support the comprehension of software systems by using Ontology and Description Logic. This formal representation supports the use of reasoning services across different knowledge resources and therefore, enables us to provide users with guidance during the comprehension process that is context sensitive to their particular comprehension task.

Keywords: *Software maintenance, program comprehension, process modeling, ontological reasoning*

1. Introduction

Program comprehension is a major factor in providing effective software maintenance and enabling successful evolution of computer systems. Some estimate that up to 50% of the maintenance effort is spent understanding source code [3]. This significant comprehension effort is due to several factors, e.g., differences among comprehension tasks and their specific settings [6]. These variations lead to a non-well defined multi-dimensional problem space that creates an ongoing challenge for both the research community and tool developers. Solutions developed to address these challenges are commonly not integrated with each other, due to a lack of integration standards or difficulties to share services and knowledge among them. The situation is further complicated by the non-existence of comprehension process models that could guide maintainers while performing comprehension tasks.

There has been little work in examining how different comprehension tools work together for end users [7, 10] and how these tools together, with other relevant program comprehension specific resources (e.g., user expertise, task and environment specific settings, etc.), can collaboratively support a specific program comprehension task. Maintainers are typically left with no guidance on how to complete a particular comprehension task within a given context. Our research aims to address this lack of context sensitivity by introducing a formal process model that stresses an active approach to guide software maintainers during comprehension tasks. Within our process model, the basic elements and their major inter-relations are formally modeled by Ontology and Description Logic

(DL) [16], and the process behavior is modeled by an interactive process metaphor. Our approach differs from existing work on comprehension models [1, 2, 4], tools integration [7, 15] and task-specific process models [7, 11] in several aspects:

1. We present a formal comprehension process model based on an Ontological and Description Logic representation. The model provides a seamless integration of different comprehension concepts, like user expertise, reverse engineering, as well as comprehension tools and techniques within such a common process model.
2. Our approach is based on an open environment to support the introduction of new concepts and their relationships, as well as enrich existing concepts with newly gained knowledge or available resources.
3. Our approach provides the ability to reason about information from the ontological representation and to provide both an active and context-sensitive support to guide the comprehension process itself.

The rest of this paper is organized as follows. The motivation for our research is introduced in section 2, followed by section 3, the relevant research background. Section 4 describes in detail the context-driven program comprehension process model, followed by section 5, the implementation and validation. Discussions and future work are presented in Section 6.

2. Motivation

Modeling program comprehension processes is a multi-dimensional problem domain, due to the interactions among different information and knowledge resources. Our process model was motivated by approaches used in other application domains, like Internet search engines (e.g., Google[1]) or online shopping sites (e.g., Amazon[2]). Common to these applications is that they utilize different information resources to provide an active, typically context-sensitive user feedback that identifies resources and information relevant to a user's specific needs. The challenges in applying similar models to program comprehension are not only caused by the required synthesis of different information and knowledge resources, but also by the need to provide a formal meta-model to enable reasoning about the potential steps of the comprehension process. The formalization of the process model enables the use of additional automated reasoning services to guide programmers in acquiring and interpreting relevant knowledge across different information resources and hierarchies.

For example, a maintainer, while performing a comprehension task, often utilizes and interacts with various tools (e.g., parsers, debuggers, source code analyzers, visualization tools, etc.). These tools interactions are caused by both the interrelationships among artifacts required/delivered by these tools and the specific techniques needed to complete a particular task. Identifying these often transitive relationships among information resources becomes a major challenge. Within our approach, we support automated reasoning across these different information resources (e.g., domain knowledge, documents, user expertise, software, etc.) to resolve the transitive relationships.

---

1 www.google.com

2 www.amazon.com

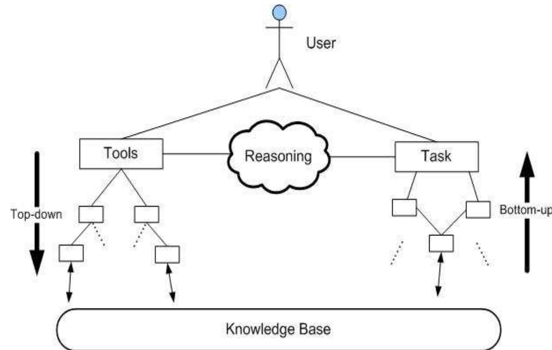

Fig. 1. Conceptual Model

Figure 1 shows a simplified illustration of this process by restricting the available information resources to tools and task only. Our process model will be able to answer questions like:

1. Given the current knowledge of the system, what tools are applicable to perform a particular comprehension task using a top-down approach?
2. Given a current knowledge level acquired in a bottom-up manner, what are the potential (direct/indirect) related tasks that can be performed?

From a more pragmatic viewpoint, process models have to be able to adapt to ever changing environments and information resources to be used as part of a process. In our approach, we address this problem by providing a uniform ontological representation that can be extended and enriched to represent any newly gained knowledge or change in the information resource(s). Furthermore, this newly gained knowledge will become an integrated part of the process that can be further utilized and reasoned on by our process model.

3. Background

In this section we review research relevant to process modeling as well as some of the technical foundations utilized in our process model.

3.1. Program Comprehension Process

Historically, software lifecycle models and processes have focused on the software development cycle. However, with much of a system's operational lifetime cost occurring during the maintenance phase, this should be reflected in both the development practices and process models supporting maintenance activities. One approach is to focus on the software maintenance process (including program comprehension) being a major part of a total system life cycle/process models, as suggested for example in [17, 19]. These process models focus mainly on the creation of additional software artifacts during software development to support future maintenance efforts.

A more specific instance of a software maintenance model is program comprehension. Program comprehension is typically referred to as the process involved in constructing an appropriate mental model of a software system to be maintained [1, 2]. There exist various aspects affecting program comprehension [3, 6], making it an inherently complex and difficult problem to address. Some of the major issues affecting the comprehension process are the user's comprehension ability (e.g., experience, knowledge background), the characteristics of the software system to be comprehended (e.g., its application domain, size and complexity), the comprehension task to be performed (e.g., maintenance, reverse engineering, architecture recovery), as well as the tools and software artifacts (e.g., source code, documentation) available to support the comprehension process. Mental models are applied to describe a maintainer's current understanding of a software system. These mental models are formed through the use of cognitive models that describe both the cognitive processes and information structures needed to create a mental model [6]. In the past decades, several cognitive models have been developed to explain how maintainers comprehend source code. Bottom-up [1], top-down [2], and integrated [3, 4] are the three major theories of program comprehension that try to model both the activities and the processes involved in creating the task-specific mental models.

These mental models build the foundation for most comprehension process models. One also has to deal with the integration of resources affecting the comprehension process and the ability to guide users during the comprehension process itself. Existing research in modeling program comprehension has focused in the past on describing the comprehension process in the context of a specific task domain. An example for such a task-specific comprehension domain is reverse engineering, which often combines, in an ad hoc manner, simple query tools and a significant effort by the user to interpret the retrieved or abstracted information [10]. Existing reverse engineering models therefore focus on modeling informally the extraction of information from the source code or documentation in order to reconstruct higher level abstractions and knowledge from a subject system [17]. Another example for a task-specific semi-automatic process model is architectural recovery [11], which can be seen as a more specific reverse engineering task. In [11], the recovery process is composed of six general phases and their sequence is described as part of this process model.

It is generally accepted that even for these more specific comprehension task instances, like architectural reconstruction, a fully-automated process is not feasible [11]. Furthermore, existing models share the following challenges:

- These models use existing information resources (e.g. user expertise, source code information, tools) to help constructing a mental model of a program. However, they lack the necessary formalism and representation that allow both the resource integration in a uniform model and the ability to infer additional knowledge from these models.
- Knowledge management (extendibility, flexibility and integration of newly gained resource and knowledge) is a major challenge due to the lack of a formal process model representation.
- These models provide typically only general descriptions of the steps involved in a process, without detailing these steps or providing guidelines on how to complete these steps within a particular comprehension setting.

In our approach, we provide a formal representation that integrates these information resources and allows reasoning and knowledge management across these resources. Fur-

thermore, we address the issue of context-sensitive support, providing the maintainer with guidance on the use of the different information resources while accomplishing a particular task.

3.2. Ontology and Description Logic

Intuitively, “software comprehension” refers to activities that humans perform: understanding, conceptualizing, and reasoning about software. In this regard, a crucial aim of providing support for software comprehension is to assist and improve human thinking processes. Research in cognitive science suggests that mental models may take many forms, but the content normally constitutes an ontology [8]. Ontologies are often used as a formal, explicit way of specifying the concepts and relationships in a domain of understanding [16], with Description Logic (DL), a knowledge representation formalism, being used as a standard ontology language. DL is a major foundation of the recently introduced Web Ontology Language (OWL) recommended by the W3C [21].

DL represents domain knowledge by first defining relevant concepts (sometimes called a class or TBox) of the domain and then using these concepts to specify properties of individuals (also called an instance or ABox) occurring in the domain. Basic elements of DL are atomic concepts and atomic roles, which correspond to unary predicates and binary predicates in First Order Logic. Complex concepts are then defined by combining basic elements with several concept constructors.

For example, given atomic concepts *Artifact* and *JPG*, as well as the role *hasFormat* (refers to an artifact that has JPG format), the concept of *JPGArtifact* can be defined as follows:

$$\texttt{JPGArtifact} \equiv \texttt{Artifact} \sqcap \exists \texttt{hasFormat.JPG}$$

Likewise, given a concept *Tool* and the role *deliver* (means that a tool delivers an artifact), the concept of *JPGOutputTool* can be defined as follows:

$$\texttt{JPGOutputTool} \equiv \texttt{Tool} \sqcap \exists \texttt{deliver.JPGArtifact}$$

DL allows for the precise characterization of a concept taxonomy in a domain, by defining subsumption relationships between concepts.

For instance, $\texttt{ComprehensionTool} \sqsubseteq \texttt{Tool}$ specifies that *ComprehensionTool* is the sub-concept of *Tool*, which means all instances of the concept *ComprehensionTool* are also instances of the *Tool* concept.

Individuals and their relationships in the domain are specified as instances of the concepts and their corresponding roles as well. Take *Creole* for example. It is the instance of *ComprehensionTool* and has *class.jpg* as output. The DL expressions for *Creole* are as follows:

```
 Creole:ComprehensionTool,
 class.jpg: Artifact,
(class.jpg,JPG):hasFormat,
(Creole, class.jpg):deliver
```

Based on these given facts, a DL reasoner can automatically infer that *Creole* is a *Tool* and a *JPGOutputTool* as well.

The basic inference on concept expressions in Description Logic is subsumption written as $C \sqsubseteq D$ [16]. With the support of an existing reasoner such as Racer [20], more reasoning services can be derived. Typical services provided by Racer include terminology inferences (e.g., concept consistency, subsumption, classification and ontology consistency checking) and instance reasoning (e.g., instance checking, instance retrieval, tuple retrieval, and instance realization). For a more detailed coverage of DLs and Racer, we refer the reader to [16,20].

4. Modeling Program Comprehension

The program comprehension meta model presented in this research is based on two major criteria:

- *Flexibility and extensibility:* The process model has to be able to adapt to every changing comprehension task and its settings, and it also needs to support its own evolution.
- *Active guidance:* A limitation of existing software process models (e.g., RUP [14]) is that these models focus only on the informal description of the different process artifacts and notations used as part of the process. This informal representation does not allow for an active guidance on what artifact or notation can/should be applied within a given process context.

A model is essentially an abstraction of a real and conceptually complex system that is designed to display significant features and characteristics of the system, which one wishes to study, predict, modify or control [12]. In our approach, the comprehension process model is a formal description that represents the relevant information resources and their interactions. The information resources include:

- *Task*: Description of the comprehension process tasks.
- *User*: Participant who has the main responsibility to fulfill the task.
- *Tool*: Description of existing and available program comprehension tools.
- *Artifact*: Description of software and comprehension process artifacts.
- *SoftwareArtifact*: Description of the target software, e.g., software documents, software components, analysis artifacts.
- *Documents*: Description of the documented artifacts
- *Historical data*: Information collected during the process for later reuse.

In what follows, we describe: (1) the ontological representation used to model the information resources, (2) the task-solving approach used to model the interaction between users and the process context, and (3) the knowledge management used to support its merging and integration.

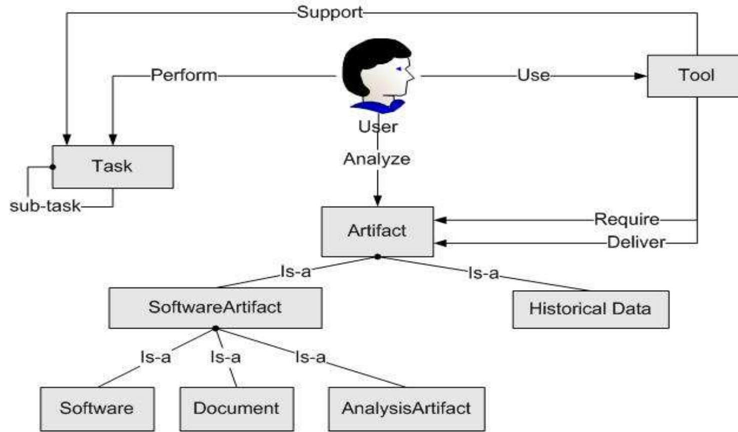

Fig. 2. Comprehension Process Meta Model

4.1 Ontological Comprehension Process Model

In this research, we use ontologies and Description Logics to formally model the major information resources used in program comprehension and their inter-relationships. The benefits of using a DL-based ontology as a means to model the structure of our process model are as follows:

Knowledge acquisition and management. As mentioned previously, program comprehension is a multifaceted and dynamic activity involving different resources to enhance the current knowledge about a system. Consequently, any comprehension process model has to reflect and model both the knowledge acquisition and use of the newly gained knowledge. The ontological representation provides us with the ability to add newly learned concepts and relationships as well as new instances of these to the ontological representation. This extendibility enables our process model not only to be constructed in an incremental way, but also to reflect more closely the iterative knowledge acquisition behavior used to create a mental model as part of human cognition of a software system [1-4].

For example, in Figure 3, each program comprehension task (e.g., source code analysis task, document analysis) utilizes information resources from the knowledge base (KB) that are needed to perform the particular task. Once the task is completed, the existing KB will be enriched with the newly derived information from the current task performed, e.g., source code analysis artifact and document analysis artifact respectively. This newly gained knowledge becomes an integrated part of the KB and can be further utilized by the following tasks.

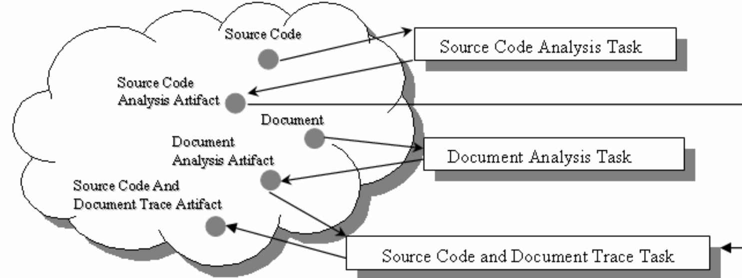

Fig. 3. Knowledge Addition in the Comprehension Process

Reasoning. Having DL as a specification language for a formal Ontology enables the use of reasoning services provided by DL-based knowledge representation systems, by inferring knowledge through transitive closure across different ontologies. The DL-based ontology and reasoning services form the basis for both, the knowledge integration and retrieval used in our process model.

Table 1. Software Comprehension Ontology (partial view)

Concept	Definition	Example Instances
	Artifact	
SourceCode	The source code of the software	Source code of uDig system [18]: SourceCode_uDig
Class	Class of the system	AddessSeeker, USGLocation
JavaDocDocument	Javadocs for the system	uDig Javadocs: JavaDocDocument_uDig
UserGuideDocument	The user's guide	uDig User Guide: UserGuideDocument_uDig
	Task	
DocumentAnalysis	Analysis of the document artifact	Document analysis for the uDig User Guide:DocumentAnalysis_UserGuideDocument_uDig
DetailedAnalysis	Analysis of the source code at or below class level	Detailed analysis for the uDig system: DetailedAnalysis_uDig
SourceCodeAndDocumentTrace	Analysis of the traceability between the document and source code	Trace between uDig source code and User Guide document: uDigSourceCodeAndDocument-Trace_uDig
	Tool	
DetailedAnalysisTool	Tool for analysis of source code at or below class level	Creole, SOUND
DocumentAnalysisTool	Tool for analysis of the document artifact	TextMiningTool
	User	
User	The analyzer	Concordia SEGroup

Building a formal ontology for program comprehension requires an analysis of the concepts and relations of the discourse domain. In particular, the outlined process model must be supported by the structure and content of the ontological knowledge base. Our approach here is twofold: We (1) created sub-ontologies for each of the discourse domains, like tasks, software, document, and tools (Figure 2); and (2) link them via a number of shared

high-level concepts, like *artifact*, *task*, *or tool*, which have been modeled akin to a (simple) upper level ontology [9].

4.2 Process Management

As mentioned previously, the interaction between users and the comprehension process in the context of a current comprehension task plays a dominant role of any comprehension process model. Users should become immersed in the program comprehension process, while a particular comprehension task unfolds. The user itself is active and interacts with different resources (e.g. support from tools, techniques, documents and expertise) and other users (e.g. system or historic user data) to complete a particular comprehension task. We introduce a process manager to establish communication and interaction between users, the process and the ontology manager. A typical tool usage scenario for our comprehension process model is illustrated in Figure 4, with the iterative nature of the process being reflected by the loop (messages 2-22). A user is completing a comprehension task and the process manager, ontology manager, reasoner and available resources are all working together to assist the user in the comprehension process.

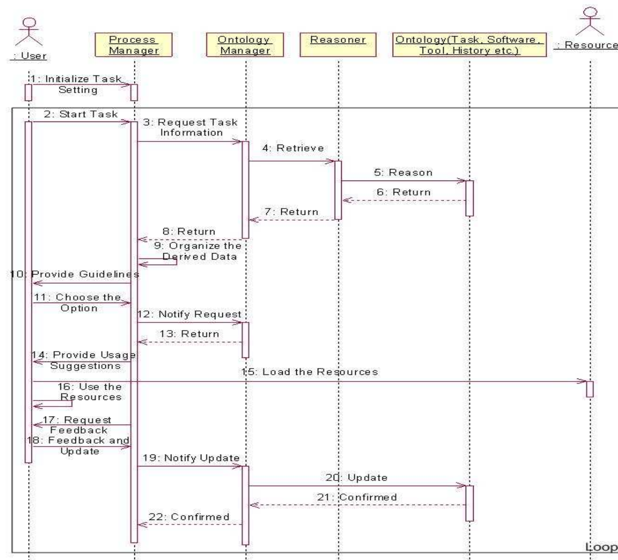

Fig. 4. Tool Usage Sequence Diagram

Based on a given task setting (message 1), the process manager applies a set of predefined queries to identify (reason about) potential resources that might be applicable to the comprehension task (messages 2-14). The user interacts with the process through the process manager by interpreting and visualizing the current comprehension context and by triggering new events and actions. Our ontological representation allows both, reasoning cross the different information resources, and adding newly gained knowledge (concepts, relations and instances) to the knowledge base, making these available for further processing. For instance, message 9 may return a list of techniques that support architecture analy-

sis, such as class model recovery, design pattern analysis, coupling analysis and metrics. The resulting set of tools will be further analyzed and a potential applicability ranking is derived. At this point, the user has the choice between three different options: (1) accept one of the suggestions (Figure 4, messages 15-18); (2) create his own queries to search and filter information for specific settings, tools or historical data; (3) explore all tools and techniques that are registered with the process model, as well as other potentially relevant information stored in the ontologies.

After completion of the tool usage, the user will provide feedback and annotate briefly his experience with the tool and his success towards problem solving (Messages 18-22). The resulting feedback is used to enrich the historical data stored in the ontology, as well as trigger the next iteration of the comprehension process (depending on the outcome of the current step success or conflict/failure).

4.3 Knowledge Management

In our environment, we have to manage knowledge from many different sources (e.g., tools or users), concerning various artifacts (e.g., source code or documentation). It is not realistic to expect all these sources to share a single, consistent view within a comprehension task. Rather, we expect disagreements between individual users and tools during an analysis. In our approach, we explicitly model those different views using a representational model that attributes information to (nested) contexts using so-called *viewpoints.*

An elegant model for managing (possibly conflicting) information from different sources has been proposed by [5]: Knowledge is structured into *viewpoints* and *topics*. Viewpoints are environments that represent a particular point of view (e.g., information stemming from a particular *tool* or entered by a *user*). Topics are environments that contain knowledge that is relevant to a given subject (e.g., design patterns, architectural recovery). These environments are *nested* within each other: viewpoints can contain either other viewpoints or topics. A topic can contain knowledge pertaining to its subject, but also other viewpoints, e.g., when the subject is another user.

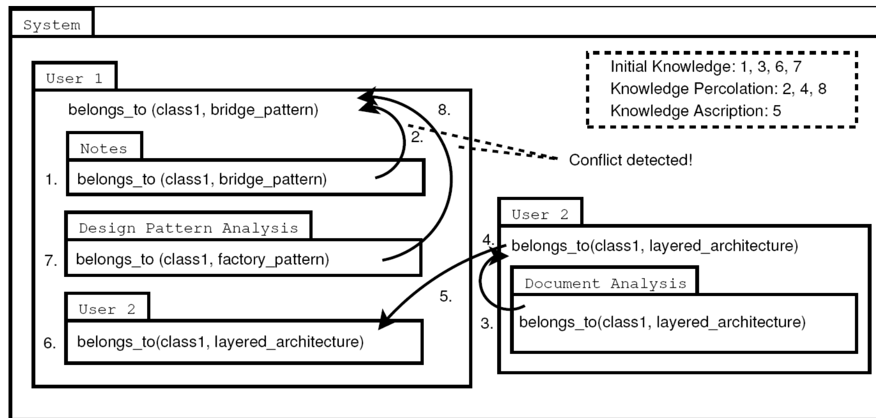

Fig. 5. Knowledge Merging Using Ascription and Percolation

Thus, we can explicitly distinguish between knowledge a user (or tool) has on certain topics and, at the same time, manage (possibly conflicting) knowledge about what *a user believes other users believe* about the same topic, since this information is contained within different, nested viewpoints. These viewpoints create *spaces* within which to do reasoning: consistency can be maintained within a topic or a viewpoint, but at the same time, conflicting information about the same topic can be stored in another viewpoint. This allows us to collect and maintain as much knowledge as possible, attributing it to its sources, without having to decide on a "correct" set of information, thereby losing information prematurely. For example, a user might believe that a certain set of classes form the *bridge design pattern*, while the documentation states they belong to an *architectural layer*, and the design pattern analysis tool identifies them as a *factory pattern* (Figure 5).

Viewpoints can be *constructed* as well as *destructed* through the processes of *ascription* and *percolation*. Stated briefly, the process of ascription allows incorporating knowledge from other viewpoints (users, tools) unless there is already conflicting information on the same topic. The mechanism of percolation is introduced for the deconstruction of nested knowledge. Here, some (assumed) held knowledge on a topic contained in a nested viewpoint may be percolated into its outer environments, up to the top-level viewpoint of a user (or the main environment) and be thus acquired as knowledge.

5. System Implementation and Evaluation

In this section, we provide both a general system overview of our process model, as well as a more detailed design description of the ontological representation used to model the knowledge base.

5.1. System Overview

A general system overview of our process model is shown in Figure 6. The process itself is based on two main components, the process and the ontology management.

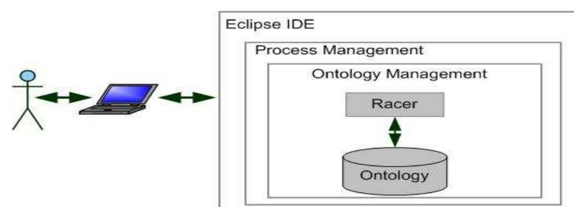

Fig. 6. System Architecture

Ontology Manager is used to manage the infrastructure of the process model, where the basic elements of the program comprehension process and their inter-relationships are formally represented by DL-based ontology. Our approach supports the addition of new concepts and their relations in a given sub-ontology, coordinates the reasoner with the ontologies, and controls querying and reasoning services across the sub-ontologies. A user can perform both pre-defined and user-defined queries. The ontology manager is an exten-

sion to our SOUND tool [13], an Eclipse Plug-in that provides both ontology management (software ontology and document ontology have been developed) and inferences service integration using the Racer [20] reasoner. So far, the ontology management interface provides the following services: adding/defining new concepts/relationships, specifying instances, browsing the ontology, and a Java Script based query interface.

Process Manager is built on top of the ontology manager and provides users with both the context and the interactive guidance during the comprehension process. The process context is established by the process manager, depending on the user process interact ions, the current state of the knowledge base and the resulting information inferred by the reasoner. For the interactive guidance, the process manager utilizes different visual metaphors to establish a representation that allows users to immerse in the process context and, at the same time, provides an approach to analyze and utilize the inferred knowledge to provide guidance during the comprehension process itself.

5.2 Initial Evaluation

At the current stage, we have successfully implemented and used our SOUND ontology management and query tool to perform comprehension tasks such as impact analysis, design pattern recovery, and component identification [13]. In addition, we have defined an initial set of concepts and relations for the remaining sub-ontologies as the foundation for our process model (see Figure 7, which shows a partial view of these ontologies).

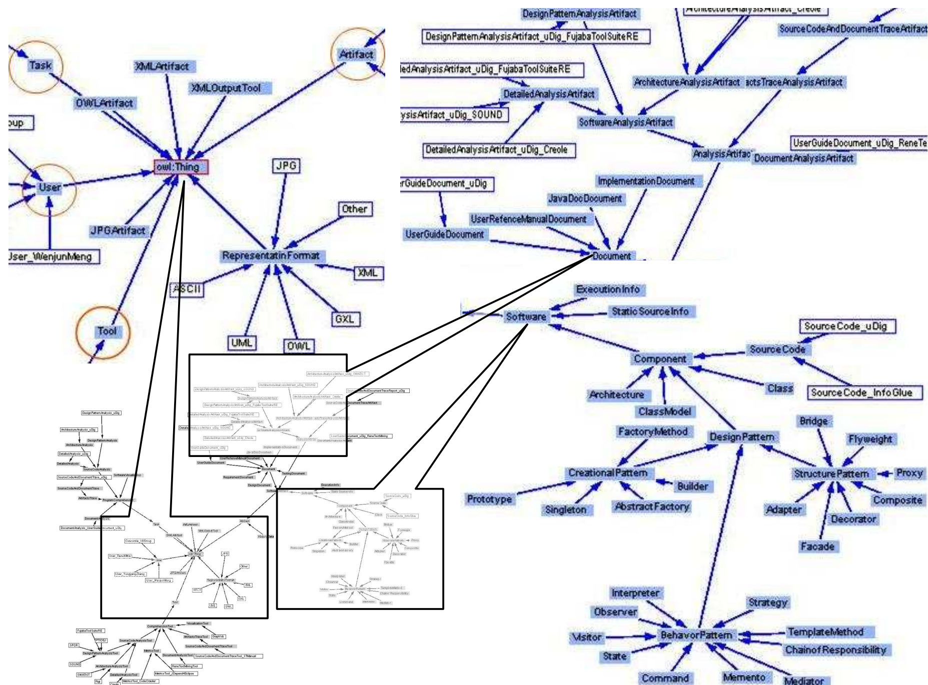

Fig. 7 Partial Ontological Design View

A set of frequently used queries has been defined in the system, e.g. identifying the coupling among classes, recovering the design pattern in the system. We are currently in the process of conducting a larger case study in collaboration with Defence Research and Development Canada (DRDC), Valcartier to explore and validate the applicability of our comprehension process model. The system used for the case study is an open source Geographic Information System (GIS) – uDig [18]. Based on the initial uDig knowledge base shown in Figure 6 and the reasoning services provided by Racer [20], we can now illustrate the use of the ontological representation inference of knowledge. The queries are all based on nRQL (new Racer Query Language) [20] and address the comprehension task related questions raised earlier in Section 2.

Q1: *Given my current knowledge of the system, what tools are applicable to perform a particular comprehension task*?

In this question, a user wants to define a query that identifies all tools that can potentially support a design pattern analysis task in the context of the uDig system. The resulting query would be as follows:

```
(RETRIEVE
     (?TOOL)
     (AND (?TOOL Tool)
          (?TOOL DesignPatternAnalysis_uDig support)))
```

The Racer reasoner provides the following query results:

```
(((?TOOL SPQR))
 ((?TOOL FujabaToolSuiteRE))
 ((?TOOL SOUND))
 ((?TOOL PTIDEJ)))
```

In this case, the reasoner would infer and identify a list of available tools in the current KB context that potentially can support design pattern analysis for our uDig case study: SPQR, FujabaToolSuiteRE, SOUND and PTIDEJ. A more advanced reasoning example is shown in the following query. This query adds one restriction to only identify the design pattern analysis tools that are applicable (all required artifacts to use these tools are available) in the current KB context.

```
(RETRIEVE
    (?TOOL)
    (AND (?TOOL Tool)
         (?ART Artifact)
         (?TOOL ?ART require)
         (?TOOL DesignPatternAnalysis_uDig support)))
```

Query results provided by the reasoner:

```
(((?TOOL FujabaToolSuiteRE))
 ((?TOOL SOUND))
 ((?TOOL PTIDEJ)))
```

In this case, the SPQR tool is excluded, since a required artifact is not available in the current KB context. Similarly, one can identify which artifacts are necessary or missing in the KB before one can apply a specific tool.

Q2: *Given a current knowledge level, what are the potential (direct/indirect) related tasks that can be performed?*

Here we assume that potential tasks refer to all tasks that can be supported by the currently available tools and artifacts in the KB. The resulting query is as follows:

```
(RETRIEVE
     (?TASK)
     (AND (?TASK Task)
          (?TOOL Tool)
          (?ART Artifact)
          (?TOOL ?ART require)
          (?TOOL ?TASK support)))
```

Query results provided:

```
(((?TASK DesignPatternAnalysis_uDig))
     ((?TASK DocumentAnalysis_UserGuideDocument_uDig))
     ((?TASK SourceCodeAndDocumentTrace_uDig))
     ((?TASK ArchitectureAnalysis_uDig))
     ((?TASK DetailedAnalysis_uDig)))
```

As part of the ongoing uDig case study, we will perform in the next step a specific component substitution task, in which a tracking component will be substituted by a client specific version. As part of this case study, individual user profiles will be set up and track the progress of users in completing the substitution task. Feedback from the process and information resource usage will be collected for further refinement and enrichment of both the process model and the knowledge base.

6. Discussion and Future Work

Program comprehension is an essential part of software maintenance, as any software evolving activity cannot skip the comprehending step. Our work promotes the use of both formal ontology and automated reasoning in program comprehension research, by providing a DL-based formal ontological representation of different information resources involved in a comprehension process.

Existing work on comprehension tool integration focuses either on data interoperability using a common data exchange format [7] or on service integration among different reverse engineering and software comprehension tools [15]. Our approach can be seen complementary to these ongoing tool integration efforts. Improving the overall capabilities and applicability of reverse engineering tools will help to enrich our tool ontology and therefore, directly/indirectly benefit the comprehension process model. However, our approach goes beyond just tool integration. It is the formal ontological representation that supports both reasoning across different knowledge sources (including tools) and provides context support and guidance during the comprehension process itself. Another major advantage of our approach is its flexibility and extensibility that support the evolution of our comprehension process model itself.

Current research in modeling software maintenance processes [7, 17, 19] typically describe only very generally the process and lack formal representations. Thus, they are unable to utilize any type of reasoning services across the different knowledge sources involved in the comprehension process. To the best of our knowledge, there exists no previous work that focuses on developing a formal process model to describe the program comprehension process.

An intuitive visual format for the task process model is still in development. Existing research in context-sensitive User Interfaces [22] can be used as a basis for our process visualization. As part of our future work, we will also conduct several case studies to enrich our current ontology and optimize the comprehension process model for different comprehension and software evolution tasks.

Acknowledgement

This research was partially funded by Defence Research and Development Canada Valcartier (contract no. W7701-052936/001/QCL).

References

[1] B. Shneiderman, "Software Psychology: Human Factors in Computer and Information Systems". Winthrop Pub.,1980.
[2] R. Brooks, "Towards a Theory of the Comprehension of Computer Programs". Int. J. of Man-Machine Studies, pp. 543-554, 1963.
[3] A. V. Mayhauser, A. M. Vans, "Program Comprehension During Software Maintenance and Evolution". IEEE Computer, pp. 44-55, Aug.,1995.
[4] S. Letovsky, "Cognitive Processes in Program Comprehension". in Empirical Studies of Programmers, pp. 58-79, Ablex Pub., 1986.
[5] A. Ballim, Y. Wilks, "Artificial Believers: The Ascription of Belief". Lawrence Erlbaum, 1991.
[6] M.-A. Storey, "Theories, Methods and Tools in Program Comprehension: Past, Present, and Future". 13th IWPC, pp.181-191, 2005.
[7] M. -A. Storey, S. E. Sim, K. Wong, "A Collaborative Demonstration of Reverse Engineering tools", ACM SIGAPP Applied Computing Review, Vol. 10(1), pp18-25, 2002.
[8] P. N. Johnson-Laird, "Mental Models: Towards a Cognitive Science of Language, Inference and Consciousness". Harvard University, Cambridge, Mass., 1983.
[9] Niles and A. Pease. "Towards a Standard Upper Ontology". Proc. of the 2nd Int. Conf. on Formal Ontology in Information System (FOIS), Maine, 2001.
[10] H.Muller, J.Jahnke, D.Smith, M.-A.Story, S. Tilley, and K. Wong, "Reverse Engineering: A Roadmap", The Future of Software Engineering, ACM Press, 2000.
[11] C. Riva, "Reverse Architecting: An Industrial Experience Report", 7th IEEE WCRE, pp. 42-52, 2000.
[12] M. I. Keller, R. J. Madachy, and D. M.Raffo, "Software Process Simulation Modeling: Why? What? How?". Journal of Systems and Software, Vol.46, No.2/3, 1999.
[13] Y. Zhang, R. Witte, J. Rilling, V. Haarslev, "Ontology-based Program Comprehension Tool Supporting Website Architectural Evolution", WSE06, Philadelphia, Sep.2006.
[14] P.Kroll, P.Kruchten, "The Rational Unified Process Made Easy: A Practitioner's Guide to the RUP". AW, 2003.
[15] D. Jin and J. R. Cordy. "Ontology-Based Software Analysis and Reengineering Tool Integration: The OASIS Service-Sharing Methodology". 21st IEEE ICSM, 2005.
[16] F.Baader, D. Calvanese, D. McGuinness, D.Nardi, P.P.-Schneider, "The Description Logic Handbook". Cambridge University Press, 2003.
[17] KH.Bennett, VT.Rajlich, "Software Maintenance and Evolution: a Roadmap". Proc. of the Conference on The Future of Software, pp.73-87, 2000.
[18] uDig tool, http://udig.refractions.net/confluence/display/UDIG/Home,accessed 29/06/2006.
[19] IEEE Standard for Software Maintenance, IEEE 1219-1998.
[20] V. Haarslev , R. Möller, M. Wessel, "Query the Semantic Web with Racer + nRQL". Proc. of the KI-2004, International Workshop on Applications of Description Logics, 2004.
[21] OWL Web Ontology Language Reference,http://www.w3.org/TR/owl-ref, accessed 24/08/2006.
[22] Mylar tool, http://www.eclipse.org/mylar/, accessed 24/08/2006.