

Software Configuration Management Using Ontologies

Hamid Haidarian Shahri^{*}, James A. Hendler[^], Adam A. Porter⁺

^{*}MINDSWAP Research Group
⁺Department of Computer Science
University of Maryland
{hamid, aporter}@cs.umd.edu

[^]Department of Computer Science
Rensselaer Polytechnic Institute
hendler@cs.umd.edu

Abstract. Configuration management is an important problem in large software systems. When dealing with hundreds of components, keeping track of version changes and various dependency constraints imposed on the system, throughout its development life cycle is very challenging. Current approaches are ad hoc and proprietary, and there exists no standard for specifying valid software configurations. We propose a novel formalization for configuration management, based on the approaches developed in classic knowledge representation domain. Component constraints and version restrictions are encoded in an ontology using the standard OWL-DL language (a W3C recommendation), which facilitates the sharing of knowledge about configurations, across various systems. Detection and pinpointing of component inconsistencies, by human, is a painstaking and time consuming process. The machine readability of the OWL language enables us to apply reasoning on the specification, and automatically deduce the validity of test configurations. In addition, justifications on the validity of a configuration are provided. Keywords: Configuration management, software evolution, knowledge representation, ontologies, formalization, standardization, automated reasoning.

1. Introduction

Software Configuration Management (SCM) is a part of configuration management and is traditionally described as a set of activities designed to control change by identifying the work products that are likely to change, establishing relationships among them, defining mechanisms for managing different versions of these work products, controlling the changes imposed, and auditing and reporting on the changes made [Pre01]. In other words, SCM is a methodology to manage a software development project. Traditional SCM focused on controlled *creation* of relatively simple products. Nowadays, implementers of SCM face the challenge of dealing with relatively minor increments under their own control, in the context of the complex system being developed. As the software components evolve, it is increasingly difficult to keep track of the changes (versions) and combinations of components that

still work together as a whole. In essence, a set of constraints need to be enforced, to ensure that components are consistent.

In this paper, a new approach to managing software configurations is investigated and the contributions of this work are as follows:

- Exploring software configuration management requirements in real-world and large applications
- Formalizing various package dependency constraints and version restrictions, using ontologies, and globally representing and integrating the configuration knowledge, which is scattered between different developers (teams)
- Modeling the formalization using Web Ontology Language (OWL), which is a widely accepted W3C standard recommendation
- Reasoning using the formalization, to automate and streamline the inconsistency detection process for a given test configuration
- Implementing the formalization representation and reasoning, using available research tools
- Identifying various other benefits of using ontologies to standardize and replace the current ad hoc and proprietary configuration management strategies.

Based on this study, we advocate the use of OWL-DL ontologies, as a novel and theoretically sound formalization, for controlling the evolution process of large-scale software systems.

Section 2 provides a motivating example for software configuration management. Section 3 specifically describes the role of ontologies in software configuration management. In this section, we devise a detailed theoretical formalization of software configuration constraints, describe a fully automated reasoning procedure for detecting inconsistencies, and discuss other potential benefits of modeling configuration management issues with ontologies. Section 4 illustrates the implementation of the formalization and reasoning. Section 5 contains some of the related work. We conclude in section 6, and suggest some future research directions.

2. Motivating Example

Nowadays, software systems usually have hundreds of packages and each package depends on one or more other packages for its operation. There are various constraints on package dependencies, for example, package p might require both packages $p1$ and $p2$, while another package might only require exactly two out of five possible package combinations. It is clear that these conditions become very complicated and hard to document, in a large software system. As the software evolves and different versions of packages are released, the problems worsen. Some versions of packages only operate with other versions.

These dependencies are locally known by groups of developers, and it is hard to produce a coherent global picture of the configurations. Sometimes, local changes have unpredictable results. Some developers with knowledge of the internal configurations might leave the development team or be relocated. Hence, future development efforts could be seriously hampered, unless the configuration knowledge of these developers is specified and documented. Considering all these issues, a

standard, integrated and formal approach, to specifying complex dependency constraints and version restrictions, is of tremendous help, in the management and maintenance of large software.

3. Ontologies for Configuration Management

The advantages of using ontologies for configuration management (CM) are immense and cover various aspects of software development and evolution. A thorough and extended case study of a large software is required, to illustrate all the benefits in detail. This section briefly outlines useful features of ontologies that we have identified to be most critical for configuration management. These ideas are clarified, by using short examples throughout the discussion.

3.1. Formalization of Software Configuration Constraints and Version Restrictions

In most large software development projects, the knowledge about the configuration of the system is only *locally* known to programmers. For example, a team or a developer might only know what versions of modules are supported, and what configurations are handled correctly, by his module, and not be aware of other modules. This type of knowledge is very hard to represent formally using current approaches.

There are few or no standards available, for compatible encoding of configuration constraints, which can be reused across different software projects in a similar fashion. The encoding of configurations is proprietary and ad hoc. This hampers compatibility and standardization. Our proposed solution utilizes OWL, which is a W3C standard recommendation (not proprietary), similar to other Web languages like html. Using this standard helps the *adoptability* of the approach, as there are many tools, open source development efforts and documentations available for such standards.

Representing the *local* knowledge available to various members of the development team, and capturing the constraints explicitly within the system in a *global* and integrated manner could provide tremendous benefits in facilitating the software evolution process. The formalization of configuration constraints seems more critical, when we consider that some developers leave the team and take this knowledge out of the system, which affects future software maintenance and evolution efforts.

In this section, we provide a solid example of how various *configuration constraints* and *restrictions* can be encoded in an ontology using OWL. Consider five packages that need to work together in a software system. Each package has different versions. The name of the packages along with their abbreviations and versions are shown in Table 1. There are various constraints in terms of what one package depends on. Simple AND and OR constraints could be shown using a dependency graph.

The ideas here are explained using a real- world software project called InterComm. InterComm is our runtime library that achieves direct data transfers between data structures managed by multiple data parallel languages and libraries in different programs. Such programs include those that directly use a low-level message-passing library, such as MPI. Each program does not need to know in advance (i.e. before a data transfer is desired) any information about the program on the other side of the data transfer. All required information for the transfer is computed by InterComm at runtime. InterComm contains hundreds of modules and thousands of versions. A description of the project is available at: <http://www.cs.umd.edu/projects/hpsl/chaos/ResearchAreas/ic/>. Considering the complexities of the actual project, we only describe a very limited and simple subset of that project for the example, in this paper, which clearly illustrates the idea.

Figure 1 shows the dependency graph and the constraints between the packages in our example. There are two types of nodes in the graph, namely circles and squares. Circles represent the packages. Squares represent the relationship between packages which are shown with multiplication and addition symbols. An addition symbol in the square implies that the package above the square depends on one of the packages below the square (that is one OR the other). A multiplication symbol in the square implies that the package above the square depends on both of the packages below the square (that is AND). In this example, only two types of Boolean operations are shown. However, modeling the constraints with description logic allows us to specify any *complex Boolean expression* of the dependent packages. Also, the number of dependent packages for a given package does not need to be two and can be *any number*, i.e. the relations are not necessarily binary.

Table 1. Information about various packages in the software.

Package Name	Package Abbreviation	Package Versions
InterComm	IC	1.0
GMP for Fortran	GMP	1.0
GCC CC	GCC	1.0
MPICH	MPI	1.0
PVM	PVM	2.0, 3.0, 4.0

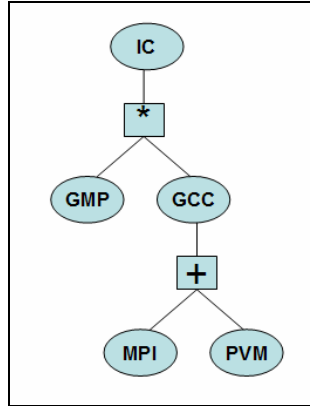


Fig. 1. Constraints between packages can be shown using a dependency graph.

In addition to the AND and OR dependency constraints mentioned previously, there are also version restrictions on various packages. The version restriction of our example is shown in Table 2. Generally version restrictions specify what range of versions work with other versions. They take the form of above or below some version number.

We formalize the above configuration in an OWL-DL ontology, which uses description logic for reasoning. ALCO DL expressivity was shown to satisfy our modeling requirements, which stands for AL - Attribute Logic: conjunction, universal value restriction, limited existential quantification; C - Complement (together with AL allows disjunction, full existential quantification); O - Nominal. For creating the ontology, and encoding of dependency constraints and version restrictions between different packages in the InterComm software (as stated above), the following steps need to be performed in order:

1. **Package versions** are treated as *individuals* (members) of classes. The individuals must be declared as pairwise distinct.
2. **Packages** are treated as *classes* in the ontology. Each class is specified via a direct enumeration of its members. This is done using the owl:oneOf construct.
3. **Package version restrictions** are represented by *classes*. For each above or below restriction, a new class must be created.
4. A *property* P is added to the ontology for correct modeling of the constraints.
5. **ValidConfiguration** is a *class* that specifies the **constraints of the dependency graph** and can be any arbitrarily complex Boolean expression, on the dependent packages. It is determined by the intersection of different *existentially* quantified statements on property P. Existential quantification is denoted by the owl:someValuesFrom construct.
6. **Test** is an *individual* that specifies the **configuration**. This configuration might be valid or invalid. Test is stated to be owl:differentFrom all other individuals in the ontology. All versions of packages in the configuration Test are added to this individual, using *object assertions* of property P.

Table 2. Version restriction between packages in the software

Package Version Restrictions
InterComm version 1.0 only operates with PVM versions 3.0 or above.

The above procedure has been implemented for the example described in this section. The creation of the ontology was done using Swoop, which is a lightweight web-based ontology editor and browser [Kal05]. More details about the implementation are provided in section 4.

3.2. Reasoning for Automated Inconsistency Detection of Configurations

In the previous subsection, we devised a general approach to formalizing and globally integrating the local configuration constraints and version restrictions. In addition to the general benefits of formal specification gained by encoding the software configuration knowledge, the *main advantage* of using ontologies is in providing a fully *automated* procedure to detect inconsistencies in the configuration, via standard reasoning services.

In the software evolution process, new dependencies arise as packages are added to the system. New versions are continuously being developed and need to be integrated into the system. The effect of these local changes on the global configuration is very hard to predict. Often, there are peculiar inconsistencies that occur and developers can not pinpoint what the causes of the inconsistencies are.

Using the formalization stated previously, a standard Description Logic reasoner can be exploited to check the validity of a given configuration, by checking all the package constraints and version restrictions that have been encoded in the ontology. The formalization was done using the machine readable OWL language and consequently the reasoning is fully *automated* and very *fast*. The reasoning procedure not only determines, if a test configuration is valid or not, it also provides *justifications* for that decision, i.e. it demonstrates the reasoning steps that lead to this decision. It distinguishes the axioms that cause the justification, and shows them to the configuration developer.

Deducing the validity of a configuration by human, is very complex, time consuming and error prone, if at all possible. Due to this complexity, in practice, the validity of a configuration is determined, by empirical testing. In our study, Pellet was used for the OWL-DL reasoning process [Sir06]. Pellet provides all the standard inference services that are traditionally provided by DL reasoners, and include:

- *Consistency checking*: Ensures that an ontology does not contain any contradictory facts. The OWL Semantics document provides the formal definition of ontology consistency which Pellet uses.
- *Concept satisfiability*: Checks if it is possible for a class to have any instances. If a class is unsatisfiable, then defining an instance of that class will cause the whole ontology to be inconsistent.
- *Classification*: Computes the subclass relations between every named class to create the complete class hierarchy. The class hierarchy can be used to answer queries such as getting all or only the direct subclasses of a class.

- *Realization*: Finds the most specific classes that an individual belongs to; in other words, computes the direct types for each of the individuals. Realization can only be performed after classification since direct types are defined with respect to a class hierarchy. Using the classification hierarchy, it is also possible to get all the types for that individual.

Based on the formalization explained in section 3.1, the inconsistency detection of a given configuration *Test*, is equivalent to running a concept satisfiability check, to investigate whether the individual *Test*, is a member of class *ValidConfig*. If class *ValidConfig* is unsatisfiable, then there are no configurations that meet the given package constraints and version restrictions. Implementation of reasoning using Pellet is described in section 4.

3.3. Other Potential Benefits of Utilizing Ontologies

New versions of software systems are created as they change during their life time. For example, different versions are created to support various machines or operating systems, or to offer a different functionality. Sometimes user requirements change and new versions are tailored for the new requirements. Configuration management is concerned with managing evolving software systems. System change is a team activity and configuration management aims to control the costs and effort involved in making changes to a system [Sum04].

Configuration management involves the development and application of procedures and standards to manage an evolving software product. Sometime, it is considered as a part of a more general quality management process. When software is released as a valid configuration, it is called a baseline, as it is the starting point for further development. Figure 2 shows various versions of a system for different platforms.

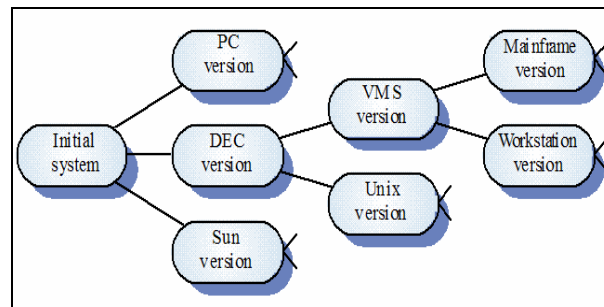


Fig. 2. Different versions of programs are used in software and they are built for various platforms.

Configuration Management Planning: In large software systems, unless there is disciplined planning, keeping track of versions could go out of control very quickly. Hence, configuration management should start during the early phases of the project. All products of the software process may need to be managed, for example specifications, designs, programs, test data, and user manuals. Thousands of separate

documents may be generated for a large software system. These documents could be organized efficiently by using a property, e.g. `GCCversion1Spec isDocumentedBy Doc133ax`. However, the main goal of this paper is to keep track of various programs and their dependencies, which is very important for shipping a product that functions correctly.

Configuration Management Plan: A configuration management plan should define the types of documents to be managed and document naming scheme. By using owl ontologies, every document or program can be uniquely named, identified and referenced using a *Universal Resource Identifier* (URI). The ability of referencing everything uniquely with URI's and in a distributed fashion is a tremendous advantage over existing naming mechanisms. This exceptional ability has been harnessed in the architecture of the World Wide Web. The CM plan should define who takes responsibility for the configuration management procedures and creation of baselines. These responsibilities can be modeled using a property, e.g. `GCCversion1 isAttendedBy Hamid`.

CM plan should specify policies for change control and version management. Again this is similar to the naming and referencing issue raised for documentation. Different versions of a program could each have properties to specify what they have been derived from, e.g. `GCCversion6.8 isDerivedFrom GCCversion4.2`. The CM plan defines the records which must be maintained in the evolution process. The plan should also describe tools which should be used to assist the process and any limitations on their use.

Configuration Item Identification: Large projects typically produce thousands of documents which must be uniquely identified. Some of these documents must be maintained for the lifetime of the software. Document naming scheme should be defined so that related documents have related names. However, the naming scheme could be a problem in practice. Using an ontology with hierarchical class and subclasses and URI's for unique referencing are helpful for this task, as they support navigation through a large set of documents by taking advantage of explicit relations encoded between entities.

Configuration Ontology: Traditionally, it is proposed that all configuration management information should be maintained in a database. We argue that modeling this information using an ontology increases expressivity and allows more flexible querying of the information. Some sample queries required in this setting are as follows: Who has a particular system version? What platform is required for a particular version? What versions are affected by a change to component X? How many faults are reported in version T? The ontology that represents this information should preferably be linked to the software being managed. Reasoning will provide more extensive information through inferring new facts from the ontology, i.e. the knowledge base. This facility is not available in a database system. Reasoning will be explained later.

Version Identification and Version Management Tools: Version is an instance of a system which is functionally distinct in some way from other system instances. Procedures for version identification should define an unambiguous way of identifying component versions. Three basic techniques for component identification are version numbering, attribute-based identification and change-oriented identification. The rich set of semantics that is available in ontologies can help in

attribute-based identification of versions, which can be represented as properties for instances (i.e. versions). Figure 3 demonstrates an example of a version derivation structure. Version management tools should assign identifiers automatically when a new version is submitted to the system. System stores the differences between versions rather than all the version code. A change history records the reasons for version creation.

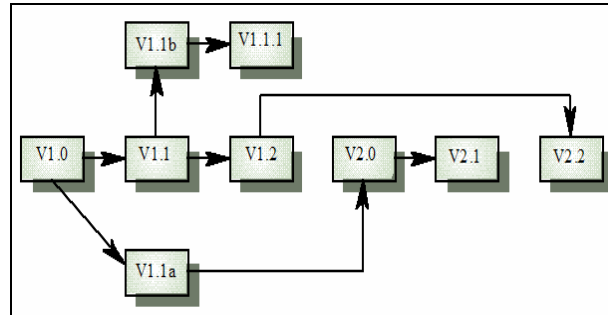


Fig. 3. Version derivation structure which shows how the software has evolved.

Release Creation: Release is an instance of a system which is distributed to users outside of the development team. Release creation involves collecting all files and documentation required to create a system release. Configuration descriptions have to be written for different hardware, as well as various installation scripts for each one. One valid configuration and its installation script is actually a consistent model of the ontology, used to represent the system. Hence, a consistent model can be saved and used as a release. The specific release must be documented to record exactly what files were used to create it, so that it can be recreated, if necessary.

System Building: System building is the process of compiling and linking software components into an executable system. Different systems are built from different combinations of components. Building process must invariably be supported by automated tools that are driven by “build scripts”. Many questions come up in building the system, for example: Do the build instructions include all required components, which should be detected by the linker? Is the appropriate component version specified? Are all data files available? Are data file references within components correct? Is the system being built for the right platform? Is the right version of the compiler and other software tools specified? Questions of this type can easily be answered using standard question answering and reasoning mechanisms. However, the modeling all these constraints, so that various important questions can be answered, could be tricky. Having said that, if an appropriate ontology is designed and the modeling is done accurately once, it can trivially be reused for other software projects.

System Building Tools: Building a large system is computationally expensive and may take several hours. Hundreds of files may be involved in this process. System building tools may provide: a dependency specification language and interpreter, tool selection and instantiation support, distributed compilation and derived object management. Figure 4 shows the system building process and different components involved in that. Build scripts can utilize an ontology to find the dependencies that

must hold between versions. Ontology provides an integrated and standard way of specifying the requirements, unlike ad hoc and proprietary current approaches.

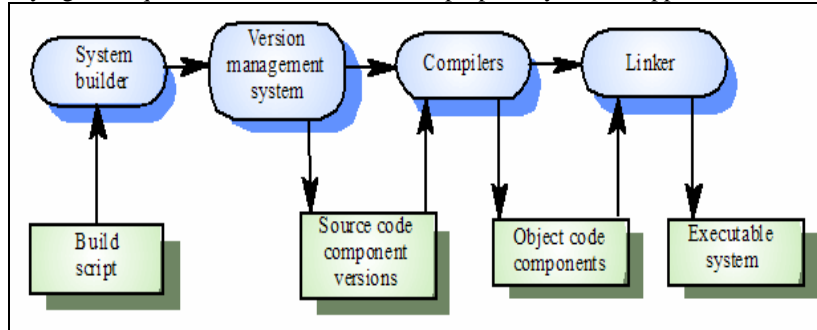


Fig. 4. System building process and various information or files involved in the procedure.

4. Implementation

The ontology for the formalization of configuration constraints described in section 3.1 has been created using Swoop ontology editor [Kal05], and is available online for browsing, via the URLs provided in Table 3. There are also simpler versions of the ontology created, which contain less packages and no restriction. The characteristics of the ontologies are provided in Table 3.

Table 3. Description of ontologies created for demonstrating the formalization of configurations.

URL	Ontology Description
http://www.cs.umd.edu/~hamid/SEproject3.owl	Three packages and one AND constraint
http://www.cs.umd.edu/~hamid/SEproject5.owl	Five packages and one AND and one OR constraint
http://www.cs.umd.edu/~hamid/SEproject5AboveBelow.owl	Five packages and one AND, and one OR constraint, plus a Above Below version range restriction

Swoop is a tool for rapid and easy browsing and development of OWL ontologies, developed in the MINDSWAP research group [Kal05]. It is a hypermedia inspired ontology editor that employs a web-browser metaphor for its design and usage. Such a tool would be more effective (in terms of acceptance and use) for the average web user by presenting a simpler, consistent and familiar framework for dealing with entities on the Web. Hence, software engineers can easily use this tool for developing their ontologies or editing sample ontologies to fit the needs of their software project.

In our implementation, Pellet was used for OWL-DL reasoning [Sir06]. Pellet is an open source tool written in Java. After formalizing the constraints and restrictions using an ontology, and creating it with Swoop, Pellet performs the reasoning. Details

of the reasoning process were provided in section 3.2. If a given configuration is valid, then it will be a member of the valid configurations class. A justification of why the configuration is valid and what axioms give rise to this conclusion are also provided for the developer.

Figure 5 demonstrates the Swoop ontology browser user interface. The left pane shows the packages as classes, versions and test configurations as individuals, and the valid configuration class. The test configuration instances are depicted by a circle on the figure. After activating the Pellet reasoner on the left pane, as shown by the circle in the image, the valid test configurations are inferred and identified as instances of the ValidConfig class. These instances named test, test1 and test2 are shown in the right pane of the figure and depicted by a circle. Each inferred valid configuration has a “Why” label in front of it, which enables the developer to view the justification for inference and understand why this configuration is valid, i.e. what constraints affect the reasoning.

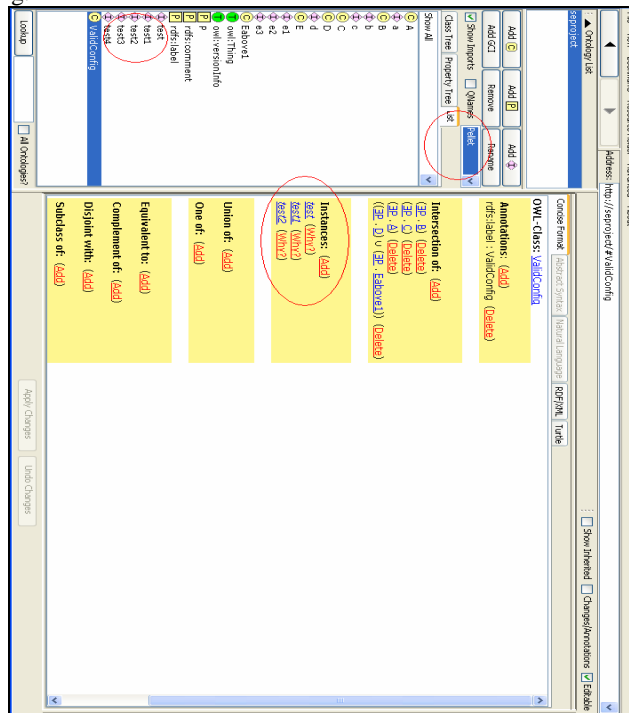


Fig. 5. Swoop graphical user interface. Test configurations, activation of Pellet reasoner, and the inferred instances of the ValidConfig class are depicted by a circle in the image. Each inferred valid configuration has a “Why” label in front of it, which enables the developer to view the justification for inference.

5. Related Work

While there has been a lot of interest in applying knowledge representation (in general) and ontologies to software engineering practices, the research is still in its infancy and there are no concrete examples of implemented methodologies that have helped the development of software. A pointer to such initiatives is the working draft of the W3C Semantic Web Best Practices & Deployment Working Group, which is part of the W3C Semantic Web activity [ODA].

Various workshops have been held in the Semantic Web research community, for example [SWESE05], [SWESE06]. To the best of our knowledge, this work presents the first study on the benefits of specifying software configuration management activities, using ontologies. More specifically, we have provided a formalization of constraints and restrictions in configurations, and exploited reasoning for inferring the validity of a test configuration.

The utility of using ontologies to combine real world domain information and software engineering knowledge, to produce up-to-date documentation for software has been studied by [Amb04].

6. Concluding Remarks and Future Work

Configuration management is the management of system change to software products. Ontologies provide a standard and powerful way of representing knowledge about a software engineering project, in general, and the configuration management tasks, in particular. *Maintaining configuration expertise*, as programmers relocate or leave the group, has always been a problem in software evolution. Our standard model facilitates the formal encoding of expertise, such that the project is not dependant on a particular developer or group of developers.

We provide a *formalization* for encoding software configuration dependency constraints and version restrictions that determine the compatibility component and correct operation of software, as a whole. The formalization is represented in standard machine readable OWL language. This enables the application of reasoning mechanisms on the specification, and *automated inference* of valid or invalid configurations. OWL is a Web standard and this facilitates easier sharing and *compatibility* of configuration specification.

To manage software evolution, a formal *documentation* naming scheme should be established and documents should be managed properly. In Addition, a consistent scheme of *version* identification should be established using version numbers and attributes. For both documents and program versions, exploitation of class-subclass property in ontologies helps in organizing the documents (or versions) and pinpointing any document (or version) with a rich set of properties, which represent the semantic relation between documents (or versions). Having Unified Resource Identifiers (URI) for *linking* to any document helps in navigation, as in the Web architecture. The configuration settings, encoded in the ontology, help in keeping track of changes, in an integrated fashion.

Acknowledgement: The Swoop ontology editor and Pellet reasoner, which are open source projects, developed in the MINDSWAP Research Group at the University of Maryland, were used for the implementation of this research.

References

- [Amb04] Ambrosio, A., et al., Software Engineering Documentation: An Ontology-Based Approach. *Proc. of 10th Brazilian Symposium on Multimedia and the Web*, 2004.
- [Kal05] Kalyanpur, A., et al., Swoop: A 'Web' Ontology Editing Browser. *Journal of Web Semantics*, Vol. 4(2), 2005.
- [ODA] Ontology Driven Architectures and Potential Uses of the Semantic Web in Systems and Software Engineering, W3C Semantic Web Best Practices & Deployment Working Group, 2006, <http://www.w3.org/2001/sw/BestPractices/SE/ODA/>
- [OWLJ] OWL Use Cases and Requirements, <http://www.w3.org/TR/2004/REC-webont-req-20040210/>
- [OWLG] OWL Guide, <http://www.w3.org/TR/2004/REC-owl-guide-20040210/>
- [Pre01] Pressman, R., *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 2001.
- [RDFP] RDF Primer, <http://www.w3.org/TR/2004/REC-rdf-primer-20040210>
- [Sir06] Sirin E., et al., Pellet: A practical OWL-DL reasoner. *Journal of Web Semantics* (To Appear), 2006.
- [Sum04] Sommerville, I., *Software Engineering*. 7th ed., Pearson, 2004.
- [SWESE05] Workshop on Semantic Web Enabled Software Engineering (SWESE 2005), Galway, Ireland, 2005.
- [SWESE06] 2nd International Workshop on Semantic Web Enabled Software Engineering (SWESE 2006), Athens, GA, USA, 2006.