Evolving a Software Configuration Management Ontology

Article · January 2007

5 authors, including:

Ricardo de Almeida Falbo
Universidade Federal do Espírito Santo
172 PUBLICATIONS   **1,661** CITATIONS

Giancarlo Guizzardi
Universidade Federal do Espírito Santo
234 PUBLICATIONS   **4,096** CITATIONS

Some of the authors of this publication are also working on these related projects:

Formalization of OntoUML View project

Breaking into Pieces: An Ontological Approach to Conceptual Model Complexity Management View project

Evolving a Software Configuration Management Ontology

Lucas de Oliveira Arantes[1,2], Ricardo de Almeida Falbo[2], Giancarlo Guizzardi[2,3]

[1]zAgile Inc., San Francisco, California, USA

[2]Informatics Dept., Federal University of Espírito Santo (UFES), Vitória, Brazil

[3]Laboratory for Applied Ontology, ISTC-CNR, Trento, Italy

lucasdeo@zagile.com, falbo@inf.ufes.br, guizzardi@loa-cnr.it

Abstract. *Software Configuration Management (SCM) can be defined as the control of the evolution of complex software systems. It is a supporting software life cycle process that benefits several activities of the software process. SCM proved to be one of the most successful software engineering technologies, and there are many tools available to support it. In spite of that, SCM has some challenges to face. One of them is the limited capability of SCM tools to interoperate. In this paper, we present an evolution of a SCM Ontology that can be used as a reference model for understanding this domain and also to build an infrastructure to allow semantic interoperability between SCM tools and other software engineering tools.*

1. Introduction

Software Configuration Management (SCM) is the discipline that enables a software development organization to keep evolving software products under control, and helps in accomplishing software quality assurance goals [Estublier 2000] [IEEE 2004].

According to the Guide to the Software Engineering Body of Knowledge (SWEBOK) [IEEE 2004], SCM aims to control the evolution and integrity of a product by identifying its elements, managing and controlling changes, and verifying, recording, and reporting on configuration information. It is a supporting software life cycle process, which benefits project management, development and maintenance activities, quality assurance activities, and the customers and users of the final product.

Many researchers consider SCM as one of the most successful software engineering practices [Estublier 2000]. But to be well implemented, automated tool support is essential, in spite of being increasingly difficult to establish it as projects grow in size and as project environments become more complex [IEEE 2004]. Without some sort of automated support, it is very difficult (if not impossible) to successfully implement a SCM process.

Different types of tool capabilities are necessary to support SCM activities, such as [IEEE 2004]: a SCM Library; software change request and approval procedures; work products (including different types of artifacts, such as documents, diagrams and code) and change management tasks; reporting software configuration status; software configuration auditing; managing and tracking software documentation; performing software builds; managing and tracking software releases and their delivery.

Depending on the situation, these tool capabilities can be made available with some combination of manual tools, automated tools providing some SCM capabilities, automated tools integrating a range of SCM (and perhaps other) capabilities, or integrated tool environments [IEEE 2004]. This scenario leads to an important requirement for SCM tools: interoperability. In fact, in general, as the use of software tools to support software development efforts has evolved, the capability of software tools to interoperate has become increasingly important.

Some sort of interoperability can be achieved by translation programs that enable communication from one specific tool to another. But this approach presents problems. As the number of tools increases and the information becomes more complex, it is more difficult for software developers to provide translators between every pair of software tools that need to exchange information [Schlenoff et al. 2000].

Obstacles to interoperability arise from the fact that the tools are generally created independently, and do not share the same semantics for the terminology used to describe its domain of interest. Different terms can be being used to represent the same concept. Without an explicit definition for the terms involved, it is difficult to see how these concepts in each tool correspond to each other. On the other hand, simply sharing terminology is insufficient to support interoperability. The tools must share their semantics, i.e., the meanings of their respective terminologies [Schlenoff et al. 2000].

One way to deal with this problem is to establish a formal specification of the semantics of the domain, using ontologies [Schlenoff et al. 2000]. An ontology is a formal description of the entities within a given domain: the properties they possess, the relationships they participate in, the constraints they are subject to, and the patterns of behavior they exhibit [Uschold and Gruninger 1996]. It provides a common terminology that helps to capture key distinctions among concepts in a given domain. We should emphasize that the focus of an ontology is not only on terms, but also on their meaning. We can include an arbitrary set of terms in our ontology, but they can only be shared if we agree on their meaning. It is the intended semantics of the terms that is being shared, not simply the terms [Schlenoff et al. 2000].

Thus, ontologies can be used to establish a common conceptualization about the SCM domain in order to support SCM tools integration. Using a SCM ontology as an interlingua, we aim to facilitate tools interoperability by means of the development of translators between native formats of those tools and the SCM ontology. This approach has potential to easy the process of integrating tools that focus on different activities of the SCM process and therefore it can be used for creating a proper SCM environment.

Nunes and Falbo (2006) have already proposed a SCM ontology. It covers well some activities of the SCM process, such as configuration items identification and change request control, but it lacks some important concepts mainly related to change and version control. Concepts such as repository, branch and copy are presented in most all SCM tools, but are not addressed by the ontology. In order to fulfill this gap, we evolved this ontology, and this paper presents the resulting ontology.

This paper is organized as follows. Section 2 talks about SCM and ontologies and shows briefly the SCM ontology proposed by Nunes and Falbo (2006). In section 3

we present an evolution of the SCM ontology. Section 4 discusses some related works and finally, section 5 describes our conclusions and future works.

2. Software Configuration Management and Ontologies

According to the CMMI (SEI, 2006), the purpose of SCM is to establish and maintain the integrity of the work products using configuration identification, configuration control, configuration status accounting, and configuration audits. SCM practices taken as a whole define how an organization builds and releases products, and identifies and tracks changes [Berczuk and Appleton 2002].

The importance of SCM is widely recognized. For example, well succeeded quality models and standards, such as CMMI [SEI 2006] and ISO/IEC 12207, define processes and practices related to it. In the CMMI, there is a generic practice (2.6 - Manage Configurations) that is entirely focused on SCM. Additionally, CMMI defines Configuration Management as a Support Process Area at Maturity Level 2. In ISO/IEC 12207 [ISO/IEC 1995] the Configuration Management Process is a supporting lifecycle process, which purpose is to establish and maintain the integrity of all the work products of a process or project and make them available to concerned parties [ISO/IEC 2002].

Looking at ISO/IEC 12207, CMMI, SWEBOK [IEEE 2004], and books such as [Pressman 2005], the activities of a SCM process can be summarized as:

1 SCM Process Implementation: a plan should be developed describing: the configuration management activities; procedures and schedule for performing these activities; the organization(s) responsible for performing these activities; and their relationship with other organizations, such as software development or maintenance. The plan shall be documented and implemented [ISO/IEC 1995].

2 Software Configuration Identification: identifies items to be controlled (called Software Configuration Items – SCIs), establishes identification schemes for the items and their versions, and establishes the tools and techniques to be used in acquiring and managing controlled items [IEEE 2004].

3 Version Control: combines procedures and tools in order to manage different versions of SCIs that are created during the software process [Pressman 2005].

4 Change Control: is concerned with managing changes during the software life cycle [IEEE 2004]. A change management process should be established including activities for: (i) requesting changes; (ii) evaluating change requests; (iii) checkout; (iv) change execution; (v) change review and (v) check in of modified items.

5 Configuration Audit: checks, among other things: if the change specified in the request has been made, and if any additional modifications have been incorporated; if a technical review was conducted to assess correctness; if organizational standards have been properly applied; if related SCIs have been properly updated [Pressman 2005].

6 Configuration Status Report: is the recording and reporting of information needed for effective management of the software configuration [IEEE 2004]. These reports include information about SCIs and changes made on them, in

order to answer questions such as: what happened? Who did it? When did it happen? What else would be affected? [Pressman 2005].

On an organizational perspective, the set of libraries, off-the-shelf components and software tools, and their respective versions, are important for future product maintenance and for this reason, they could also be seen as SCIs.

Software items evolve as a software project proceeds. On this ongoing process, versions of the SCI are stored in order to keep track of the differences between the starting point and the current state of a given SCI. A version of a software item is a particular identified and specified item. It can be thought of as a state of an evolving item [IEEE 2004]. For each version a unique identifier is given.

A revision is a new version of an item that is intended to replace an old version of the item. A variant is a new version of an item that will be added to the configuration without replacing an old version [IEEE 2004]. This way, we can have different lines of development, grouping a set of versions with a particular goal. Sometimes there must be more than one line of development to support, for instance, a product customization or to fix a given bug. Different lines of development are also called branches.

On a SCM environment, branches are grouped on repositories. A repository is a set of mechanisms and data structures that allow software teams to manage changes on an effective way [Pressman 2005]. Generally, organizations create different repositories for different purposes (a repository for programming code, another related to contracts and agreements, other to organization guidelines, and so on).

When a given SCI or a set of SCIs achieves a desired state of maturity or an important point of the development is achieved, a baseline is created. Baselines provide a stable basis for continuing evolution of SCIs [SEI 2006]. According to CMMI a baseline is a set of SCI versions that has been formally reviewed and agreed on, that thereafter serves as the basis for further development or delivery, and that can be changed only through change control procedures. As a product evolves, several baselines may be used to control it [SEI 2006]. In other words, a baseline is a set of SCIs formally designated and fixed at a specific time during the software life cycle [IEEE 2004].

An artifact can depend on another one (for example, a requirement specification can depend on a use case diagram), and thus the same applies for the corresponding SCIs. Thus, when a change on a version of a given CI is requested, an analysis has to be done in order to look for possible impacts on versions of other SCIs. A configuration manager shall analyze if a change should be done, and which are the versions involved that should be checked out [IEEE 2004]. In a *checkout* action, copies of each requested version are generated. Once the copies are changed, they should be checked in. The configuration manager should evaluate the changes and according to his/her decision, new versions can be created.

As we can see, SCM is essential to every software organization. Thus, the concepts that surround it should be well defined and people and tools should share a common conceptualization about this domain. In this context, ontologies can be used to avoid misunderstanding and for establishing a common conceptualization.

The majority of SCM-related tools normally focus on a particular set of activities of the SCM process. Thus, a proper SCM environment cannot be achieved with a single tool. A solution for this problem lies on integrating different tools. In this scenario, tools focused on different activities of SCM process can be combined using techniques for interoperation. Commonly used techniques, such as translation mechanisms, have simplified integration for software developers in various domains [Schlenoff et al. 2000]. But this approach can be problematic if it is done for each pair of tools, especially as the number of tools to be integrated grows and when many different interpretations can emerge due to the complexity of the domain of interest. In this context, ontologies can be used as a reference model for achieving semantic interoperability.

2.1. Ontologies

According to Guizzardi (2007), an ontology is a conceptual specification that describes knowledge about a domain on a language-independent way. Moreover, an ontology aims to restrict vocabulary interpretations so that its logical models gets as near as possible to the set of structures that conceptualize that domain.

As any software engineering artifact, ontologies must be developed following software engineering practices. To build the SCM ontology, we used SABiO (Systematic Approach for Building Ontologies) [Falbo 2004]. SABiO encompasses the following activities:

- Purpose identification and requirement specification: concerns to clearly identify the ontology purpose and its intended uses, i.e. the competence of the ontology;

- Ontology Capture: the goal is to capture the domain conceptualization based on the ontology competence. Relevant concepts, relations, properties and constraints should be identified and organized. A model using a graphical language (a UML profile) and a dictionary of terms should be used to aid communication with domain experts;

- Ontology Formalization: aims to explicitly represent the conceptualization captured in a formal language;

- Integration of existing ontologies: during ontology capture or formalization, it could be necessary to integrate the current ontology with existing ones, in order to reuse previously established conceptualizations;

- Ontology evaluation: the ontology must be evaluated to check whether it satisfies the requirements;

- Documentation: all the ontology development must be documented.

2.2. The First Version of a SCM Ontology

The first version of a SCM ontology, which we use as basis for our work, was also developed using SABiO. The following competency questions were taken into account [Nunes and Falbo 2006]:

1. What are the items placed under SCM?
2. What are the variations of a given configuration item?

3. How does a configuration item's variation is decomposed?
4. Which other variations can be impacted when a given change on a particular variation takes place?
5. A variation of an item derived from an artifact is adherent to the structure of that artifact?
6. Which variations can be modified on a given change?
7. Which are the variations produced by a given change?
8. Which variations do compose a baseline?
9. Which SCIs does a human resource has access to? What is the kind of access?
10. Who is responsible for a given change?
11. Is a given variation of a configuration item available to be changed?

To answer these questions, the concepts and relations shown in Figure 1 were modeled as part of the ontology[1] [Nunes and Falbo 2006].
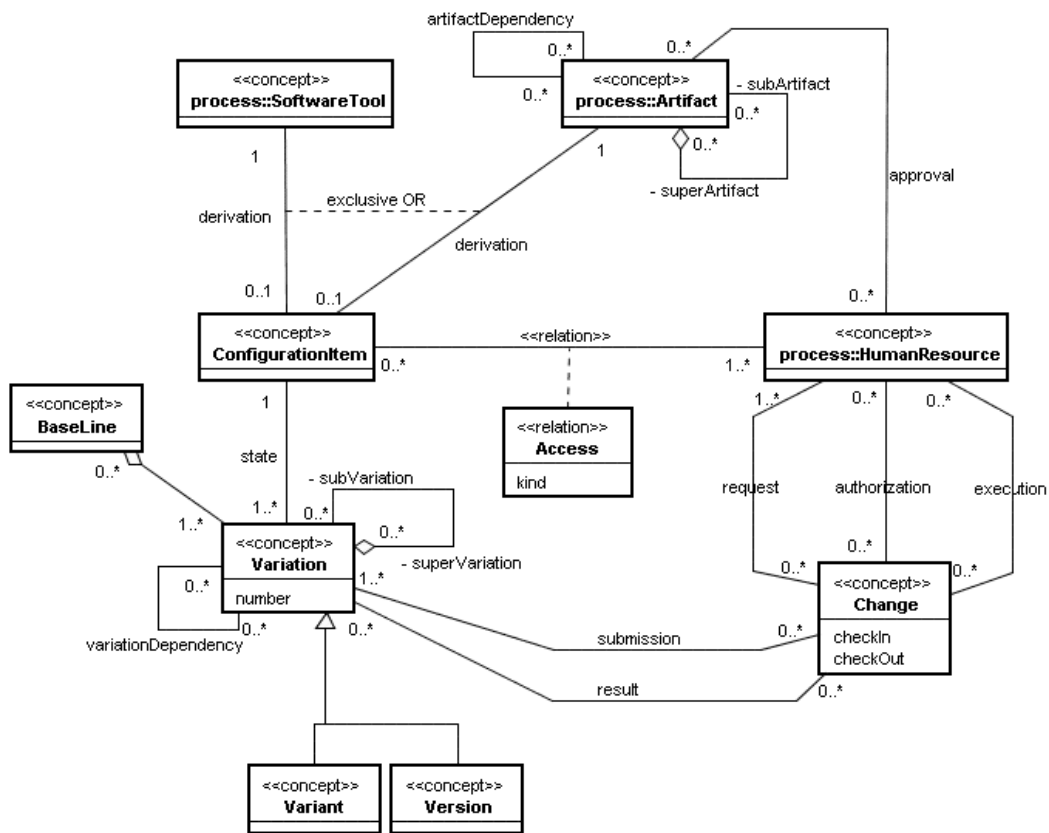

Figure 1 - The SCM Ontology proposed in [Nunes and Falbo 2006].

According to this model, artifacts and software tools can be placed under configuration management and therefore they are treated as SCIs. During its lifecycle, a SCI can have many variations, tracing the SCI's evolution. A variation of a configuration item can be decomposed or it can depend on another variation. A baseline

---

[1] Due to space limitations, axioms and dictionary of terms are not presented.

is a set of variations grouped for some purpose. Variations can be versions, which are variations that are intended to replace an old one, or variants, which are not intended to replace an old one. Configuration item's variations are submitted to changes, and the dates of checkout and check in actions are treated as properties of a change. Responsibilities for requesting, authorizing and performing a change are also taken into account. Finally, kinds of access to SCIs (read/write rights) were assigned to human resources.

3. Evolving the SCM Ontology

As mentioned before, most SCM tools focus on a sub-set of the activities of the SCM process. For example, CVS and Subversion focus only on version control. Thus, to adequately support the SCM process, we need to use several tools, which in turn should be integrated. The integrated tools must be able to understand the data produced by each other, and should be able to communicate. Therefore, they need to share a common conceptualization about the SCM domain. This is our main motivation for building a SCM ontology. With this ontology, an integration middleware could be built, composed by adaptors for different tools transform their data according to a single model (the ontology). In this approach the conceptual model of a tool should be excavated and then mapped to the ontology.

We started trying to use the SCM ontology proposed in [Nunes and Falbo 2006] as our reference model. First we excavated the conceptual models of CVS and Subversion using the approach proposed in [Hsi 2005] and an approach based on XML schema excavation, respectively. Then, we tried to map the concepts to the ontology, but we found some gaps, namely:

- Different lines of development were not addressed by the ontology. I.e., the concept of branch is missing;

- The concept of repository is also missing, and it is an essential concept, being part of the SCM common sense;

- Once a change is approved, a checkout action is done to retrieve the versions to be modified. Discarding a checked out version is a possible scenario. Also, introducing a new artifact and changing existing versions are also real situations. Thus, the concept of version copy is necessary;

- This ontology does not deal properly with the problem of locking versions of SCIs;

- This ontology uses a term unfamiliar to experts in the SCM domain: variation.

Based on the gaps identified at Nunes and Falbo's SCM ontology, we decided to work on its evolution. Firstly, we decided to use the term "version" instead of "variation", and to use the definitions given in the SWEBOK [IEEE 2004]: "A *version* of a software item is a particular identified and specified item. It can be thought of as a state of an evolving item. A *revision* is a new version of an item that is intended to replace the old version of the item. A *variant* is a new version of an item that will be added to the configuration without replacing the old version".

Second, we augment the set of competency questions with the following ones:

12. Is a version available to be changed (unlocked) in a given moment?

13. If a version is locked, which person has rights to handle it?

14. What are the SCM repositories of a given project?

15. Which are the branches of a given SCM repository?

16. Which are the versions in a given branch?

17. Which are the new items (unversioned items) that are added during a change?

18. Who are the persons that have copies of a given version?

19. Which are the copies generated by a checkout action?

20. What are the copies that are checked in?

To answer these questions, we had to consider four main new aspects in the ontology: locking versions (questions 12 and 13), repository structure (questions 14 to 16), adding unversioned items (question 17), and copies (questions 18 to 20). To capture these aspects, new concepts, relations, properties and constraints arose. Figure 2 shows the resulting model. In this figure, new concepts are in gray. Concepts remaining from [Nunes and Falbo, 2006] are in white. Concepts that we changed the term to refer to them (version and revision) or that we introduced from other ontologies that were reused (project and person) are in yellow.
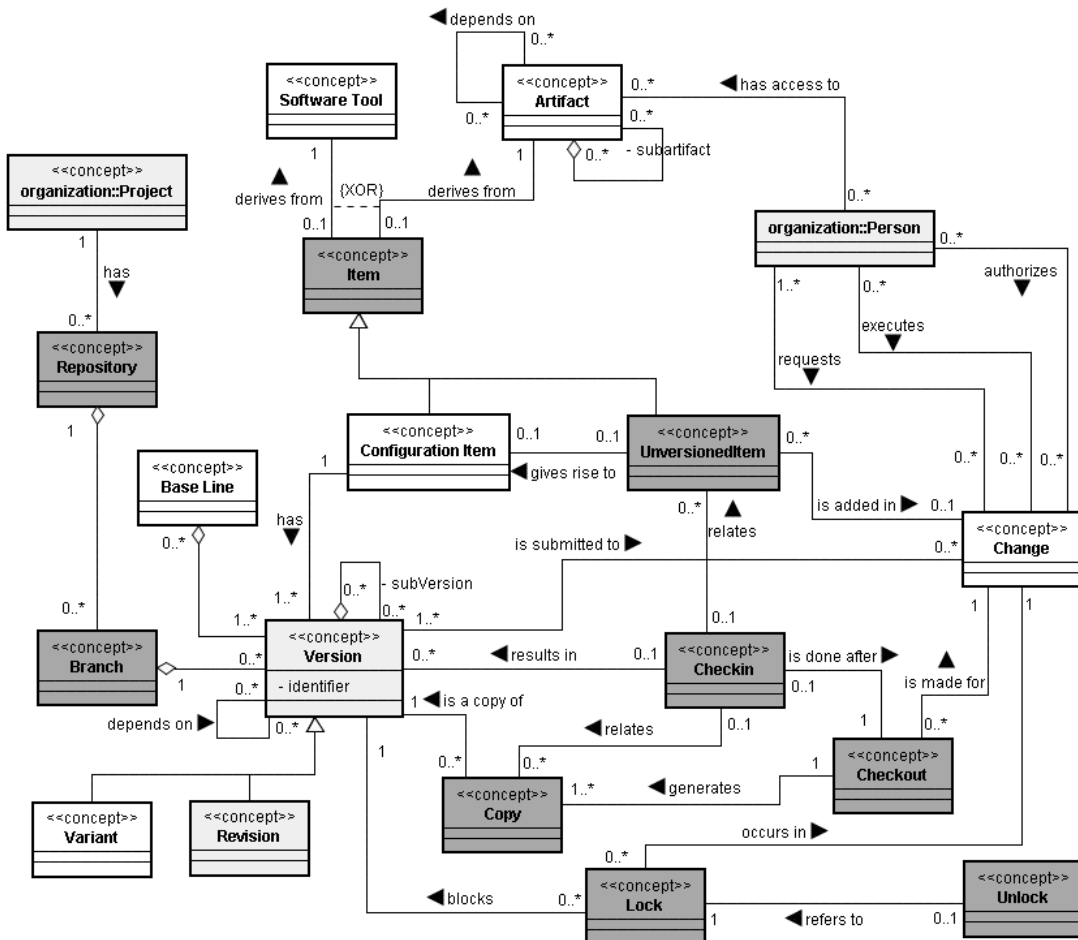

Figure 2 – The new version of the SCM Ontology.

Versions of SCIs are maintained in a project repository. A repository is a set of mechanisms and data structures that allow changes to be managed in an effective manner [Pressman 2005]. To allow multiple lines of development, repositories are organized in branches that contain versions of SCIs (questions 14 to 16).

When a change request is approved, the versions involved can be checked out. In a checkout, copies of each requested version are generated to the person that requested them (questions 18 and 19). In this moment, the developer may request a lock on a given version to guarantee that only he/she is modifying it. In order to have a version locked, the developer must be authorized to do a certain change on that version, and only after the approval and checkout he/she may have the version locked. A developer holding a locked version may say when this version is unlocked, according to his/her will. In other words, he/she should explicitly unlock a locked version (questions 12 and 13).

During a change, not only copies of existing versions of items are modified but also the first version of an unversioned item can be created (question 17).

After modifying the copies and adding new unversioned items or requesting removing versions, a *check in* is done. New versions that resulted from the last check in are created on the repository and are now ready to be checked out.

In this domain, there are several constraints that were formalized by writing axioms. Some of them are presented below, using the following predicates: *isMadeFor(co,ch)* denotes a checkout *co* is made in the context of a change *ch*; *isCopyOf(cp, v)* denotes that *cp* is a copy of the version *v*; *generates(co,cp)* denotes that a checkout *co* generates a copy *cp*; *isSubmittedTo(v,c)* denotes that a version *v* is submitted to a change *c*; *blocks(l, v)* denotes that a lock *l* blocked a version *v*; *occursIn(l,ch)* denotes a lock *l* occurs in the context of a change *ch*; *refersTo(ul, l)* denotes that an unlock *ul* refers to a lock *l* and thus it unlocks the corresponding version.

Axiom 1: A copy generated by a checkout must be copy of a version submitted to the corresponding change.

$$(\forall ch,co,v,cp) \; isMadeFor(co,ch) \land generates(co,cp) \land$$
$$isCopyOf(cp,v) \rightarrow isSubmittedTo(v,ch)$$

Axiom 2: A lock can only block a version submitted for the corresponding change in which the lock has occurred.

$$(\forall ch,l,v) \; occursIn(l,ch) \land blocks(l,v) \rightarrow isSubmittedTo(v,ch)$$

Axiom 3: A version is considered as being currently locked iff there is a lock blocking that version and there isn't an unlock that refers to this lock.

$$(\forall l,v) \; blocks(l,v) \land \neg(\exists ul) \; refersTo(ul,l) \leftrightarrow isCurrentlyLocked(v)$$

4. Related Works

Several works claims that ontologies are a promising way to achieve interoperability between tools, but we did not found a SCM ontology other than the one we used as basis for the one we presented in this paper.

In the context of the OASIS methodology for software maintenance, Jin and Cordy (2005) propose an architecture for interoperating software analysis and reengineering tools. In their work, domain ontologies are also used as a reference model along with conceptual service adaptors, which make the connection between the tool specific conceptual model and the ontology. But the ontology itself is not presented. Their purpose and approach are quite similar to ours, but the domains are different, and in this paper we focused on presenting the ontology that we will use to integrate SCM tools.

PSL (Process Specification Language) [Schlenoff et al. 2000] aims to treat the semantic interoperability problem in the context of the manufacturing process domain. In short terms, the purpose of PSL is to address this problem by creating a neutral, standard language for process specification to serve as an interlingua to integrate multiple process-related applications throughout the manufacturing lifecycle. This language is underlined by formal semantic definitions, given by a process ontology. Again, the purpose of PSL is the same of ours, but the domains are different. While PSL describes the basic conceptualization of the manufacturing process domain, we built an ontology for the SCM domain.

Also driven by the lack of interoperability between tools, but now in a more general perspective, Kappel et al. (2005) aim to provide a semantic infrastructure for model-based tool integration. To keep this infrastructure evolvable, a scalable architecture for realizing tool integration is provided that minimizes the effort necessary for integrating new tools while maximizing reuse of integration knowledge. Integration is specified both at syntactic and semantic levels. The syntactic level deals with metamodels which define the structures and data types of models, whereas the semantic level uses ontologies which describe the semantics of modeling concepts.

To serve as both a research vehicle and a test bed for exploring applications of semantic technologies in model-based tool integration, they built ModelCVS, a prototype of a system implementing the main ideas described above. The core of the system is based on the versioning system CVS. ModelCVS is able to perform tool metamodel integration on basis of semantics covered by tool ontologies, and thus these individual tool ontologies have to be integrated. For integrating the ontologies, they use a hybrid approach, involving a direct mapping between ontologies, an indirect mapping via an upper level ontology, and a mapping based on a library of already mapped ontologies.

Although the purpose of ModelCVS is the same of ours, our approach is different. Instead of using a hybrid approach based on direct mapping between ontologies, indirect mapping via an upper level ontology, and a mapping based on a library of already mapped ontologies, we decided to use a domain ontology as reference model, like OASIS and PSL. Thus, in this paper we presented this domain ontology, the SCM ontology.

5. Conclusions and Future Work

Different types of tool capabilities are necessary to support the SCM process. Therefore, to provide a widely automated support for this process, it is necessary to use different tools, which should be integrated in order to interoperate. In this context, a domain ontology can be used to establish a common conceptualization about the SCM universe of discourse, serving as a reference model to map the conceptual models underling SCM tools.

In this paper we presented a SCM ontology that is an evolution of the one proposed in [Nunes and Falbo 2006]. New concepts, such as repository, branch and copy, were introduced, as well as, new relations related to them. This evolution was motivated by the inspection of the conceptual models extracted from well known version control systems, such as CVS and Subversion.

Now we are working on an ontology-based architecture that allows SCM tool integration. The ontology establishes a common conceptualization about the SCM domain that the tools must share, although they can focus on different activities of the SCM process (for example, one focusing on version control, another focused on change control). We are also interested to allow a tool that treat a specific activity of the SCM process to be replaced by another one that also treats the same activity, without data loss. As a starting point for this perspective, we are trying to use both CVS and Subversion (two widely used tools), allowing the user to switch from these two different version control systems.

Finally, according to Guizzardi (2006), many benefits can be obtained if a domain ontology is developed based on a foundational ontology. Thus, concerning the SCM ontology itself, we are studying how we can reuse an upper-level ontology in order to benefit from its conceptualization.

Acknowledgments

References

Berczuk, S. and Appleton, B. (2002), *Software Configuration Management Patterns: Effective Teamwork, Pratical Integration* , Addison Wesley, 1st Edition.

Estublier, J. (2000), Software Configuration Management: A Roadmap, In: Proc. of the Future of Software Engineering, ICSE'2000, Ireland.

Falbo, R.A. (2004), Experiences in Using a Method for Building Domain Ontologies Proc. of the 16th International Conference on Software Engineering and Knowledge Engineering, International Workshop on Ontology In Action, Banff, Canada.

Guizzardi, G. (2006) "The Role of Foundational Ontology for Conceptual Modeling and Domain Ontology Representation", Companion Paper for the Invited Keynote Speech, 7th International Baltic Conference on Databases and Information Systems, Vilnius, Lithuania.

Guizzardi, G. (2007), On Ontology, ontologies, Conceptualizations, Modeling Languages, and (Meta)Models, Frontiers in Artificial Intelligence and Applications, Databases and Information Systems IV, IOS Press, Amsterdam.

Hsi, I. (2005), Analyzing the Conceptual Integrity of Computing Applications Through Ontological Excavation, PhD Thesis, USA.

IEEE (2004) SWEBOK - Guide to the Software Engineering Body of Knowledge, 2004 Version, IEEE Computer Society.

ISO/IEC (1995), ISO/IEC 12207, *Information Technology – Software lifecycle processes*.

ISO/IEC (2002), ISO/IEC 12207 Amendment 1, *Information Technology – Software lifecycle processes*.

Jin, D., Cordy, J.R., (2005), Ontology-Based Analysis and Reengineering Tool Integration: The OASIS Service-Sharing Methodology, Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05), New York, USA.

Kappel, G., Kramler, G., Kapsammer, E., Reiter, T., Retschitzegger, W., Schwinger, W. (2005) ModelCVS - A Semantic Infrastructure for Model-based Tool Integration, Technical Report.

Nunes, V.B., Falbo, R.A. (2006) "Uma Ferramenta de Gerência de Configuração Integrada a um Ambiente de Desenvolvimento de Software", V Simpósio Brasileiro de Qualidade de Software, Vila Velha, Brazil.

Pressman, R.S. (2005), Software Engineering: A Practitioner's Approach, McGraw-Hill, 6th Edition.

Schlenoff, C., Gruninger, M., Tissot, F., Valois, J., Lubell, J., Lee, J., (2000), The Process Specification Language (PSL) Overview and Version 1.0 Specification, NISTIR 6459, National Institute of Standards and Technology, Gaithersburg, MD.

SEI (2006) CMMI for Development Version 1.2, CMU/SEI-2006-TR-008, Software Engineering Institute, Carnegie Mellon University.

Uschold, M and Gruninger, M. (1996), Ontologies: Principles, Methods and Applications, Knowledge Engineering Review, vol. 11, pp. 96-137.