Evolving a Software Requirements Ontology

Ricardo de Almeida Falbo[1], Julio Cesar Nardi[2]

[1]Computer Science Department, Federal University of Espírito Santo – Brazil
[2]Federal Center of Technological Education of Espírito Santo - UnED Colatina - Brazil
falbo@inf.ufes.br, julionardi@yahoo.com.br

Abstract. Requirements Engineering (RE) is a complex process. Establishing a common conceptualization about its domain is important for several reasons, such as communication and interoperability between RE tools. Truthfulness to reality and conceptual clarity are fundamental quality attributes of domain ontologies, and are directly responsible for the effectiveness of these models as reference frameworks. A way to achieve these quality attributes is by grounding domain ontologies in a foundational ontology. This paper presents an evolution of a Software Requirements Ontology that was reengineered by mapping the concepts of its previous version to the Unified Foundational Ontology (UFO).

1 Introduction

Gather the right requirements in a software project is one of the most important activities of the software process. Deficient requirements are one of the main causes of software project failure [1], and thus the Requirements Engineering (RE) process is crucial for the success of a project and it should be carefully performed.

Despite of the heterogeneous terminology and the diversity of RE processes defined in the literature, a RE process should take into account the following related activities: elicitation, analysis, specification, verification and validation, and management of software requirements [1] [2] [3]. These activities may vary in timing and intensity for different projects, but it is widely recognized that software projects are critically vulnerable when these activities are performed poorly [3].

For a RE process to be well implemented, however, several factors should work well, among them the communication between people, the interoperability between automated supporting tools, and requirements reuse. Barriers to these factors many times arise from the lack of a shared understanding for the terms used to describe the RE domain. Ontologies are an emerging mechanism for dealing with this problem. According to Guizzardi [4], a domain ontology is a conceptual specification of the semantics of a certain domain that describes knowledge about it. An ontology aims to restrict vocabulary interpretations so that its logical models get as near as possible to the set of structures that conceptualize the domain. Thus a domain ontology can be used to establish a common conceptualization about the RE domain in order to support communication, requirements reuse and RE tools integration.

In [5], we presented a Software Requirements Ontology (SRO) that was developed aiming at partially formalizing the knowledge involved in the RE domain. Its main intended use was to support the integration and development of RE tools. However, during the RE tools development, we notice that there are some problems with it, resultant mainly from implicit ontological commitments. Thus we decide to evaluate and reengineer it by mapping its concepts to the Unified Foundational Ontology (UFO) [4] [6]. UFO has been used to evaluate, re-design and integrate (meta) models of different conceptual modeling languages [4], as well as to evaluate, re-design and give real-world semantics to domain ontologies [6]. By doing this, we have corrected a number of conceptual problems in the SRO by making it more truthful to the domain being represented and by making explicit some of its ontological commitments that were implicit.

This paper presents the resulting version of the SRO and is organized as follows. Section 2 talks a little bit about Requirements Engineering and ontologies. In section 3, we present the new version of the SRO. Finally, section 4 presents our conclusions and future works. Due to the lack of space, the ontological analysis that was done to achieve the new version is out of the scope of this paper.

2 Software Requirements Engineering and Ontologies

Requirements Engineering (RE) is the branch of Software Engineering concerned with the real-world goals for functions of software systems, constraints on them, and also with the relationship of these factors in the specification of the software behavior and to their evolution over time [7].

In general, the RE process involves elicitation, analysis, specification, verification and validation, and management of software requirements. Requirements elicitation is a human activity concerned with identifying requirements, what also regards where they come from and how software engineers can collect them [3].

Requirements Analysis deals with requirements classification, modeling, and allocation to components, and also with detecting and resolving conflicts between them [2] [3]. Requirements Specification, in turn, aims to produce an official document, generally called Software Requirements Specification (SRS), which is to be systematically reviewed, evaluated, and approved [3]. The quality of this document is very important because it will be widely used throughout the development process.

Requirements Verification and Validation aims exactly to ensure that the work products produced during the RE process, including the SRS, are quality products. The requirements should be validated to ensure that the software engineers have understood the requirements. It is also important to verify if the work products conform to organizational standards and if they are understandable, consistent and complete [3].

Finally, requirements can change or evolve due to a variety of reasons, and thus it is necessary to manage requirements. In this context, traceability is essential. It is possible only if there are explicit links between requirements and other assets of the software process. Identifying how requirements are decomposed, dependencies and conflicts between them, their sources, stakeholders, and work products that deal with them are also essential in order to trace requirements [2] [8].

As we can see, the RE process is very complex. It is a multi-disciplinary process, employing several people, techniques and tools at different phases of the software development. This shows that we need a shared understanding about the requirements domain, and thus developing an ontology about the requirements domain is important to support several tasks, such as communication and reuse, as well as to improve RE tool interoperability. The importance of ontologies for the requirements domain is recognized by several researchers. Riechert et al. [9], for instance, developed a requirements engineering ontology, capturing requirements relevant information that were used to develop a tool for semantic based RE.

As told before, we developed an ontology about the requirements domain, presented in [5]. However, we detected some problems with it, and we decided to analyze it. As pointed by Guizzardi et al. [6], a foundational ontology[1] can be used to evaluate, re-design and give real-world semantics to domain ontologies. For evaluating our SRO, we used the Unified Foundational Ontology – UFO [4] [6]. UFO has been developed based on a number of theories from Formal Ontology, Philosophical Logics, Philosophy of Language, Linguistics and Cognitive Psychology. It is composed by three main parts: UFOs A, B and C.

UFO-A, an ontology of *endurants* (objects), is its core. A fundamental distinction in this ontology is between the categories of *Particular (Individual)* and *Universal (Type)*. Particulars are entities that exist in reality possessing a unique identity, while Universals are patterns of features, which can be realized in a number of different particulars. UFO-A is presented in depth and formally characterized in [4].

UFO-B is an ontology of *perdurants* (events). Perdurants are individuals composed of temporal parts. They *happen in time* in the sense that they extend in time, accumulating temporal parts [6]. They contrast to endurants, in the sense that endurants are wholly present whenever they are present, i.e., they *are in time*, in the sense that if we say that in a circumstance c_1 an endurant e has a property P_1 and in a circumstance c_2 it has the property P_2 (possibly incompatible with P_1), it is the very same endurant e that we refer to in each of these situations [6].

UFO-C is an ontology of social entities (both endurants and perdurants) built on top of UFO-A and UFO-B. One of its main distinctions is between Agentive and Non-agentive substantial particulars, termed *Agents* and *Objects*, respectively. Agents can be physical (e.g., a person) or social (e.g., an organization or a society). Objects can also be further categorized into physical and social objects [6].

Due to space limitations, it is impossible to discuss here all the distinctions made in UFO. So, in Figure 1 we present some of its concepts that are important for this paper. The ones that are directly used here are shown detached in grey, and are described in sequel.

Concerning universals, the following concepts were considered important for this paper [4] [6]:

- Kind (from UFO-A): a substance sortal[2] universal (see [4]) that supplies a principle of identity for its instances (rigid sortals). Every object in a conceptual specification should be an instance of a kind.

[1] Foundational ontologies are theoretically well-founded domain-independent systems of categories that can be used to improve the quality of conceptual models [6]. They describe very general concepts like object, event, action etc, which are independent of a particular domain.

[2] Substantials are entities that persist in time, keeping their identity. Substantial universals are patterns of features that can be realized in a number of different substantials. Some of them are sortal (sortal universals), thus providing a principle of individualization, persistence and identity. Others are merely characterizing (said mixin universals) [4].

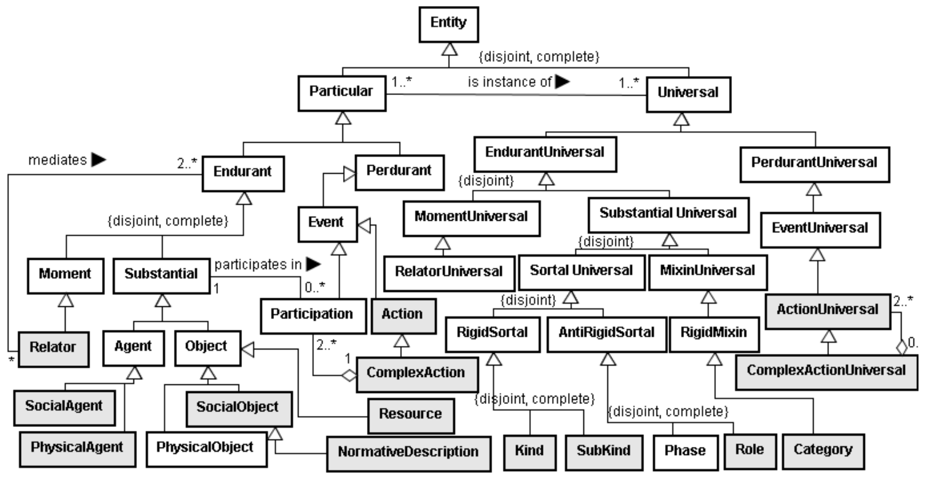

Fig. 1. A Fragment of UFO, including concepts from UFOs A, B and C.

- Role (from UFO-A): a possible role that a substance sortal can play along its history. An entity plays a role in a certain context, demarcated by its relations with other entities.
- Category (from UFO-A): a rigid mixin[3] that subsumes different kinds.
- Action Universal (from UFO-C): type of intentional event, describing patterns of features instantiated by multiple action occurrences.
- Complex Action Universal (from UFO-C): an action universal that is composed by other action universals.

Regarding particulars, the following concepts of UFO are important [6] [4]:

- Action (from UFO-C): an intentional event[4], i.e., an event which instantiates an action universal with the specific purpose of satisfying some intention.
- Complex action (from UFO-C): an action that is composed of two or more participations. These participations can themselves be intentional (i.e., be themselves actions) or unintentional events.
- Object (from UFO-C): a non-agentive substantial particular.
- Social Object / Agent (from UFO-C): an object / agent that is not physical.
- Normative description (from UFO-C): a social object that defines rules/norms recognized by at least one social agent.
- Resource (from UFO-C): an object participating in an action.
- Relator (from UFO-A): an individual with the power of connecting entities.

In this paper, these distinctions are shown in the concepts of the Software Requirements Ontology as stereotypes, indicating that they are subtypes of concepts of UFO, in an approach analogous to the one defined in [4]. When a concept is not stereotyped, then it presents the same stereotype of its super-type in the model.

[3] Mixins are dispersive universals, covering many concepts with different principles of identity.

[4] Events are possible transformations from a portion of reality to another, i.e., they may change reality by changing the state of affairs from one (pre-state) situation to another (post-state) situation. Events are ontologically dependent entities in the sense that they existentially depend on their participants in order to exist (UFO-B) [6].

3 Software Requirements Ontology

In this section we present part of the reengineered Software Requirements Ontology (SRO). Due to the lack of space, in this paper we concentrate on the conceptual model developed, although competency questions were also defined. It is worthwhile to point yet that this work was done in the context of the ODE (Ontology-based software Development Environment) Project, in which there are other software engineering ontologies developed. Thus we reused the conceptualization of the following ontologies:

- Software Process Ontology [6]: this ontology is composed by sub-ontologies about activities, resources, procedures and work products, and addresses the basic conceptualization regarding the software process domain;
- Software Configuration Management Ontology [10]: conceptualize the software configuration management domain, treating issues such as change management, versioning, and repository structure;
- Software Organization Ontology: exploits the domain of software organizations, describing their structure and also covering aspects related to competencies of their members.

Figure 2 shows each ontology as a UML package, and the dependency associations indicate that the SRO borrows part of the conceptualization defined in each one of the others.

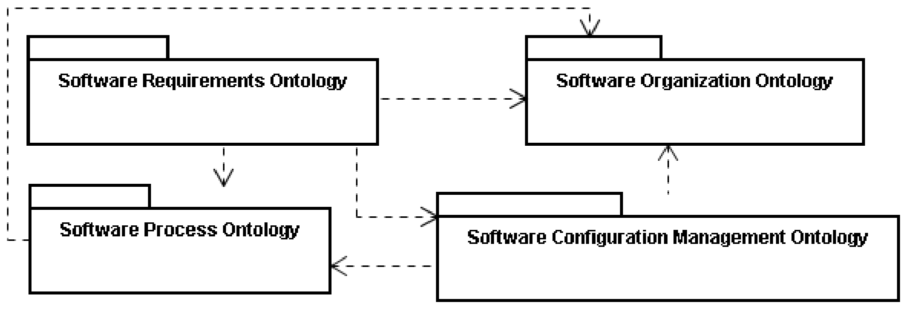
Software Requirements Ontology
Software Organization Ontology
Software Process Ontology
Software Configuration Management Ontology

Fig. 2. The SRO and its dependencies with other ODE's ontologies.

For making the ontology presentation easy, we divided it in four fragments, discussed in sequel: RE process definition, Requirements definition and taxonomy, Requirements interest and approval, and Requirements management. Concepts introduced in the SRO are detached in grey.

Requirements Engineering Process Definition

Defining a Requirements Engineering (RE) process is the same as defining any other software process. Thus, first, we inspected the software process ontology [6] to see how it treats the definition of a software process. In this ontology, we have software process kinds (Universals), which represent process models that can be instantiated as software process occurrences (Particulars), as shown in Figure 3.

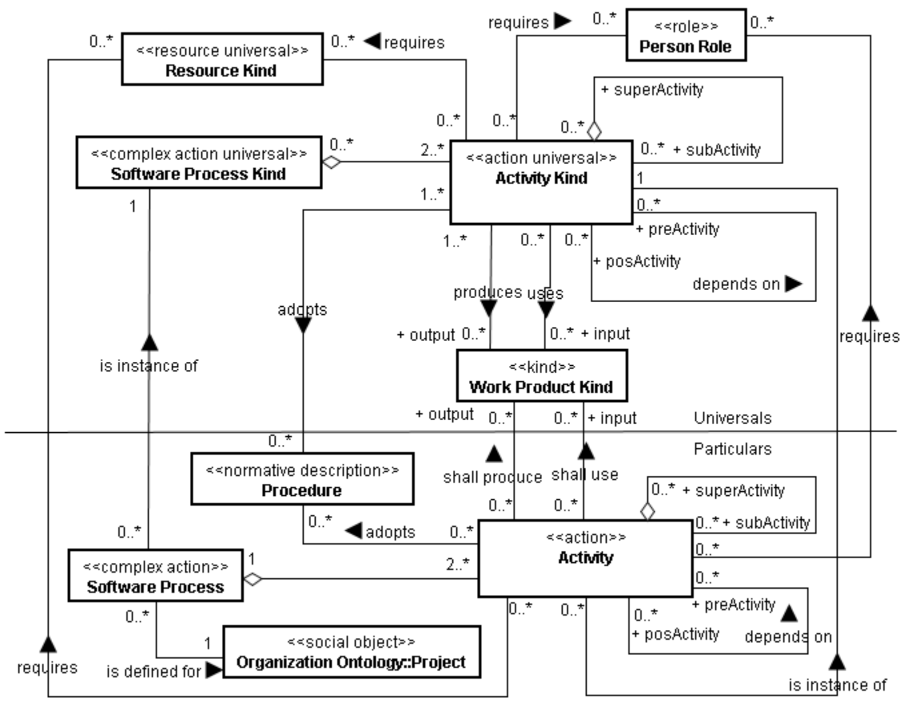

Fig. 3. A Fragment of the Software Process Ontology.

Taking the RE process into account and considering the RE process proposed in SWEBOK [3], we have a RE process model composed by the following kinds of activities: requirements elicitation, analysis, specification, verification and validation. Activity kinds that the process model prescribes are instantiated as activities occurrences (Activity in Figure 3). A specific project *P* can instantiate this process model, giving rise to the *P*'s RE process. Depending on the life cycle model adopted (concept not shown in Figure 3), several occurrences of the same activity kind can be instantiated. For instance, if an incremental life cycle model is adopted, with 3 cycles, then 3 activities of the kind requirements elicitation will be instantiated. The decomposition and precedence between activities kinds are also generally defined in a process model, and like process instantiation, they are also instantiated for projects.

In the definition of an organizational standard processes, it is usual to define, for each kind of activity: (i) which kinds of work products are expected to be produced and used, (ii) kinds of resources (software tools, equipments etc), (iii) roles of people required to perform activities of such kind, and (iv) procedures (methods, techniques, document templates and so on) that can be adopted in the execution of the activities.

When an organizational process (a process kind) is instantiated for a project, giving rise to a project's process (a process occurrence), many definitions (procedures, resources kinds, person roles and work products kinds) are directly reused by means of relating the corresponding process assets to the project's activities (activity occurrence), although some can be tailored for considering the project particularities. It is worthwhile to point that, for instance, activities produce concrete

Work Products (Particulars). Although this is not shown in Figure 3, the software process ontology also covers this.

In the case of the RE process, we can say, for instance, that the activity kind Requirements Analysis requires a modeling tool as a resource kind, a Requirements Engineer as a person role, and that it uses requirements as input and produces class diagrams as output. When this part of the process model is instantiated to a project, it gives rise to the corresponding relationships between an activity occurrence *a* and these elements, i.e. *a* may also require a Requirements Engineering and a modeling tool, may also produce class diagrams, and may also uses requirements as input.

Requirements Definition and Categorization

In general, requirements are sentences describing services that a system should provide, constraints that it should obey and features that it should present. Moreover, requirements are defined in the scope of a project. As shown in Figure 4, both Requirements and Software Requirements Specifications (SRS) are artifacts, i.e. work products that are typically put under software configuration management (see also Figure 6). More specifically, a SRS is a document, i.e. an artifact that is not executable, composed by, among others, textual sentences. In the case of a SRS, it is composed, among others, by requirements, as stated by the following axiom:

$$(\forall srs)\ softwareRequirementSpec(srs) \rightarrow (\exists r)\ ((subWorkProduct(r,srs) \wedge requirement(r))$$

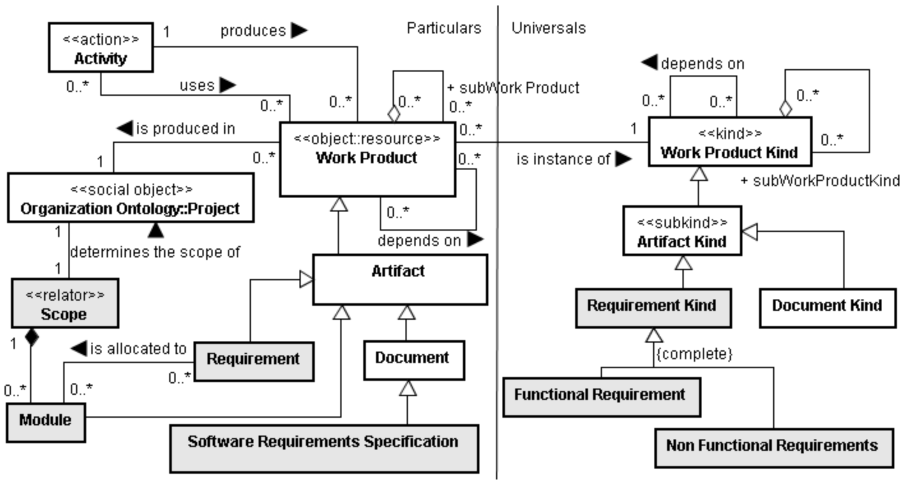

Fig. 4. A fragment of the SRO dealing with requirements definition and taxonomy.

As cited before, requirements, as any other work products, are produced in a project. Moreover, a project has a scope that is composed by several modules, to which requirements are allocated. The following constraint applies: if a requirement *r* is allocated to a module *m*, and *r* is produced in a project *p*, then *m* should belong to a scope *s* that determines the scope of the project *p*.

$$(\forall r,m,p)\ (\ allocatedTo(r,m) \wedge producedIn\ (r,p)\ \rightarrow (\exists s)\ ((partOf(m,s) \wedge determinesScopeOf(s,p))$$

A module can be decomposed into sub-modules. Thus, if requirement *r* is allocated to module *m2* that is part of another module *m1*, then *r* is also allocated to *m1*.

$$(\forall r,m1,m2)\ (\ allocatedTo(r,m2) \wedge partOf(m2,m1) \wedge module(m1)) \rightarrow allocatedTo(r,m1)$$

Requirements are categorized by requirement kinds, which in turn can be decomposed in other requirement kinds, giving rise to a requirements taxonomy. There are many possible requirement kinds, and an organization is free to define its own taxonomy. In spite of that, there is a consensus that there are two broad main classes of requirements: functional and non-functional requirements.

$$(\forall r,kr)\ (\ requirement\ (r) \wedge instanceOf(r,kr)) \rightarrow requirementKind(kr)$$

Requirements Interest and Approval

The RE process involves people from several areas and with different points of view. It is important to know the stakeholders that are interested in each requirement, in order to facilitate negotiation, elucidation and change impact determination. Moreover, it is necessary to have people responsible for a given requirement. Such people can, for example, approve a requirement before it can be treated by subsequent activities of the software process. Figure 5 shows a fragment of the software requirements ontology that deals with these questions.

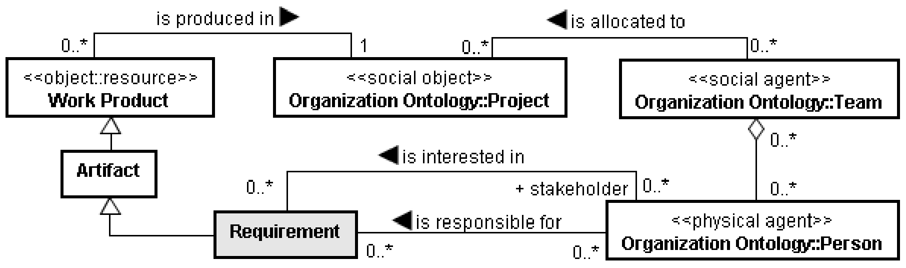

Fig. 5. A fragment of the SRO concerning requirements interest and approval.

According to the software organization ontology, people are organized in teams that are allocated to projects. This leads to the following constraint: if a person *p* is responsible for or interested in a requirement *r* that is produced in a project *prj*, then she/he should be part of a team that is allocated to *prj*.

$$(\forall r,\ p,\ prj)\ ((interestedIn(p,\ r) \vee responsibleFor(p,\ r)) \wedge producedIn(r,\ prj)) \rightarrow\ (\exists t)\ (partOf(p,\ t) \wedge allocatedTo(t,\ pr))$$

Requirements Management

Requirements management comprises change control, version control, status tracking and traceability [8]. Concerning change and version control, we are talking about putting requirements and Software Requirements Specifications (SRS) under configuration management. So, to deal with these aspects, we reused the conceptualization described in the Software Configuration Management (SCM) Ontology [10]. In this ontology, artifacts when put under SCM derive Software Configuration Items (SCIs). A SCI, in turn, has versions, as shown in Figure 6. The SCM Ontology also describes other important concepts related to version control (repository, branch, baseline, among others), and change control (change, checkout, copy, checkin etc). Due to space limitations, these concepts are not shown in Figure 6.

Concerning traceability, it is very useful to establish a net of links between requirements and other elements, such as other work products (including other requirements), people and activities, in order to maintain the integrity of the requirements (and also the integrity of related work products). The ability to keep track of these relationships is crucial not only to integrity, but also for measuring the impact of changes. These relationships includes structural relationships among a requirement and other requirements that are its constituent parts (and also the structural relationships among a SRS and the requirements that compose it), dependencies between requirements, conflicts between requirements, the source of requirements, and relationships between requirements and other work products that describe, model or implement a requirement. Figure 6 shows all these relationships.

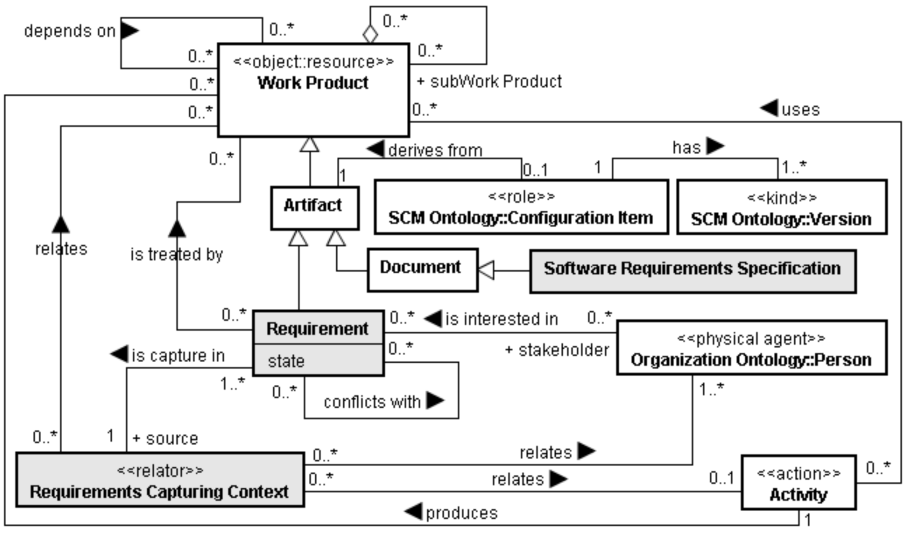

Fig. 6. A fragment of the SRO talking about requirements management.

The dependency relation between work products allows establishing dependency links between requirements, denoting that if a requirement changes, then probably its dependents need to be changed. This relation is also useful to map dependencies between requirements and other work products. But there are a stronger dependency relationship between a requirement and work products that effectively treat it, such as analysis and design models and source code. Thus, we decided to explicitly model this relation (Requirement is treated by Work Product) in order to capture this important distinction.

Analogous to the dependency relation between work products, the whole-part relation between work products allows establishing structural relationships among a requirement and other requirements that are its constituent parts, and among a SRS and the requirements that compose it. It is worthwhile to point that if a work product *wp2* is part of another work product *wp1*, then *wp1* depends on *wp2*.

$$(\forall\ wp1, wp2)\ partOf(wp2, wp1) \rightarrow dependsOn(wp1, wp2)$$

Conflicts between requirements are another type of relationship that is very important to capture. Conflicts can occur between two stakeholders requiring mutually incompatible features, between requirements and resources, or between functional and non-functional requirements. Only when conflicting requirements are known, actions can be taken to manage them, what includes negotiating with stakeholders to resolve them or to balance to maintain them acceptable.

For tracking requirements since their conception, it is necessary to capture the context in which they were elicited. This source context are generally characterized by, among others, work products been inspected, people interacting or being observed, and activities being done.

4 Conclusions

This paper presented part of the latest version of our Software Requirements Ontology (SRO), which was obtained by a reengineering process that grounded it in the Unified Foundational Ontology. The use of UFO as a basis for reengineering the SRO has shown to be very useful. When we looked at UFO, we corrected several conceptual mistakes, making SRO more truthful to the domain being represented.

The SRO also covers requirements quality evaluation that was not discussed in this paper. However, there are other aspects of the requirements domain that are not addressed by the SRO and that should be incorporated to it in future works, such as traceability of requirements to business goals.

References

1. H.F.Hofmann,F. Lehner. “Requirements Engineering as a Success Factor in Software Projects, IEEE Software, July/August 2001.
2. G. Kotonya and I. Sommerville, *Requirements Engineering: Processes and Technique*s, John Wiley & Sons, 2000.
3. IEEE (Institute of Electrical and Electronic Engineers), “SWEBOK - Guide to the Software Engineering Body of Knowledge”, IEEE Computer Society, 2004 Version.
4. G. Guizzardi, Ontological Foundations for Structural Conceptual Models, Universal Press, The Netherlands, 2005, ISBN 90-75176-81-3.
5. J.C. Nardi, R.A. Falbo, “Uma Ontologia de Requisitos de Software”, Proceedings of the IX Iberoamerican Workshop on Requirements Engineering and Software Environments, La Plata, Argentine, 2006 (in Portuguese).
6. G. Guizzardi, R.A. Falbo, R.S.S. Guizzardi, “Grounding Software Domain Ontologies in the Unified Foundational Ontology (UFO): The case of the ODE Software Process Ontology”, Proceedings of the XI Iberoamerican Workshop on Requirements Engineering and Software Environments, Recife, Brazil, 2008.
7. P. Zave, “Classification of research efforts in requirements engineering”**.** ACM Computing Surveys Journal, vol. 29, n. 4, 1997, pp. 315-321.
8. S. Robertson and J. Robertson. *Mastering the Requirements Process*. 1st edition, ACM Press, Addison Wesley, 1999.
9. T. Riechert, K. Lauenroth, J. Lehmann, S. Auer, “Towards Semantic based Requirements Engineering”, 7th International Conference on Knowledge Management (I-KNOW), 2007.
10. L.O. Arantes, R.A. Falbo, G. Guizzardi, “Evolving a Software Configuration Management Ontology”, Proceedings of the Second Workshop on Ontologies and Metamodeling in Software and Data Engineering, p. 123-134, João Pessoa, Brazil, 2007.