Engineering the ontology for the Software Engineering Body of Knowledge: Issues and Techniques

Alain Abran

École de technologie supérieure, Université du Québec, alain.abran@etsmtl.ca

Juan-José Cuadrado

Computer Science Department, University of Alcalá, jjcg@uah.es

Elena García

Computer Science Department, University of Alcalá, elena.garciab@uah.es

Olavo Mendes

École de technologie supérieure, Université du Québec, olavomen-des@gmail.com

Salvador Sánchez

Computer Science Department, University of Alcalá, Salva-dor.sanchez@uah.es

Miguel-Angel Sicilia

Computer Science Department, University of Alcalá, msicilia@uah.es

(alphabetical order)

Abstract

The process of collaborative elaboration of the Guide to the Software Engineering Body of Knowledge (SWEBOK) has produced a notable consensus on the scope of this discipline, and the continuous review process provides a mechanism for its enhancement and extension. The SWEBOK has motivated several research initiatives that attempt to engineer an ontology of Software Engineering both as an artefact for applications and also as a vehicle for the review and evolution of the Guide. Existing approaches to develop an ontology of the SWEBOK provide different perspectives on the process, from the more conceptually oriented to the more logics-operational. This chapter summarizes the different perspectives and provides an integrated synthesis of approaches in addition to a discussion of the main concepts that cross-cut the Knowledge Areas defined currently in the SWEBOK.

1 Introduction

Auyang (2004) described *engineering* as "the science of production". This and many other definitions of engineering put an emphasis on disciplined artefact creation as the essence of any engineering discipline. However, the material object produced by every engineering discipline is not necessarily of a similar nature. The case of Software Engineering is particularly relevant in the illustration of such differences, since *software* as an artefact is acknowledged as a very special piece of human work. The special nature of software was attributed by Brooks (1987) to "*complexity*" as an essential characteristic. The following quote from Brook's paper illustrates the presupposed impact of complexity in the activities of engineering.

> Many of the classic problems of developing software products derive from this essential complexity and its nonlinear increases with size. From the complexity comes the difficulty of communication among team members, which leads to product flaws, cost overruns, and schedule delays. From the complexity comes the difficulty of enumerating, much less understanding, all the possible states of the program, and from that comes the unreliability. From complexity of function comes the difficulty of invoking function, which makes programs hard to use. From complexity of structure comes the difficulty of extending programs to new functions without creating side effects. From complexity of structure come the unvisualized states that constitute security trapdoors.

The term “essential” (as opposed to “accidental”) is a well-known tool for ontology engineers (Welty and Guarino 2001), which helps in determining the properties of concepts that objects possess “always and in every possible world”. The position of Brooks on the essentials of the object of the discipline leads to a particular conception of Software Engineering as a human endeavour that attempts to tackle an inherently complex problem, since it takes as a point of departure that complexity is a feature that can not be removed from the engineering process. Consequently, it is difficult to consider methods that are definitive for the production of software, and the field is expected to be changing as methodologies are introduced and applied in an attempt to manage to the extent possible the complexity of the activities. This has a consequence on research and inquiry, since the qualities of a tool or method to tackle with software complexity are difficult to assess, and this in turn leads to a plurality of approaches. Such diversity in many leads to difficulties in contrasting the appropriateness of techniques in terms of rational inquiry methods as those established by Popper (1959) in his method for scientific discovery.

Empirical research on proposed software methods, processes, tools and techniques are of course fundamental to the discipline. In addition, ontology engineering is from our viewpoint also important for the evolution of the science of Software Engineering, at least in two dimensions. On the one hand, ontology may help in the organization and meta-analysis of empirical data and empirical approaches (Brooks, 1997), also facilitating the adequate comparison and evaluation of methods, techniques or tools. On the other hand, ontologies translated into machine-understandable representations may help in the development of computerized tools that, to some extent, take into account the purpose and consequences of the diverse Software Engineering activities. Even though we de not believe that ontologies would become a “silver bullet” for every software production problem, they are promising tools to help in the work of researchers and practitioners, and they would also serve as an element of analysis and discussion for engineers and for learning about the discipline.

Consensus-reaching approaches to ontology engineering are deemed as appropriate for the crafting of representations of the concepts of some concrete domain. Nonetheless, in some domains the engineer can find pre-existing processes of consensus-reaching on conceptual frameworks. This is the case of Software Engineering, in which the SWEBOK project is the result of a considerable effort on the collaborative production of a subset of the knowledge of the discipline that is as of today subject to little controversy in the community of researchers. In addition to the collaborative

effort, that will be briefly described next, the project adopts a literature-based approach (Sicilia, García-Barriocanal, Díaz and Aedo, 2003) in selecting some relevant articles. Thus, the SWEBOK guide provides a ground of rationality and consensus that constitutes a valuable input for ontology engineering.

The chapter of Ruiz and Hilera in this volume has provided an overview of current approaches to the ontology of Software Engineering, some of them based on the SWEBOK. This chapter concentrates now on the specifics of two approaches to SWEBOK-based ontological inquiry that are complementary in their objectives and methods.

The rest of this chapter is structured as follows. Section 2 provides an account of the SWEBOK as a project, its main principles and its method from creation and revision. Then, Section 3 describes some results of a process of inquiry on SWEBOK-based ontology from the viewpoint of the experimental study of the process of rational argument and consensus-reaching by software engineers. Then, Section 4 provides the complementary view of producing ontological representation linked to commonsense knowledge bases, which provide the benefits of reuse of existing ontological engineering and of being prepared for the construction of ontology-based tools. On the basis of the experiences described in Section 3 and 4, Section 5 sketches the main ontological elements distilled.

2 History and principles of the SWEBOK project

The Guide to SWEBOK should not be confused with the Body of Knowledge itself, which already exists in the published literature. The purpose of the Guide is to describe what portion of the Body of Knowledge is generally accepted, to organize that portion, and to provide a topical access to it. The Guide to the Software Engineering Body of Knowledge (SWEBOK) was established with the following five objectives:

1. To promote a consistent view of software engineering worldwide
2. To clarify the place–and set the boundary–of software engineering with respect to other disciplines such as computer science, project management, computer engineering, and mathematics
3. To characterize the contents of the software engineering discipline
4. To provide a topical access to the Software Engineering Body of Knowledge

5. To provide a foundation for curriculum development and for individual certification and licensing material.

The first of these objectives, a consistent worldwide view of software engineering, was supported by a development process which engaged approximately 500 reviewers from 42 countries in the Stoneman phase (1998-2001) leading to the Trial version, and over 120 reviewers from 21 countries in the Ironman phase (2003) leading to the 2004 version. More information regarding the development process can be found in the Preface and on the Web site (www.swebok.org). Professional and learned societies and public agencies involved in software engineering were officially contacted, made aware of this project, and invited to participate in the review process. Associate editors were recruited from North America, the Pacific Rim, and Europe. Presentations on the project were made at various international venues and more are scheduled for the upcoming year.

The second of the objectives, the desire to set a boundary for software engineering, motivates the fundamental organization of the Guide. The material that is recognized as being within this discipline is organized into the first ten Knowledge Areas (KAs) listed in Table 1. Each of these KAs is treated as a chapter in this Guide.

Table 1 The SWEBOK Knowledge Areas (KAs).

Software requirements
Software design
Software construction
Software testing
Software maintenance
Software configuration management
Software engineering management
Software engineering process
Software engineering tools and methods
Software quality

In establishing a boundary, it is also important to identify what disciplines share that boundary, and often a common intersection, with software engineering. To this end, the Guide also recognizes eight related disciplines, listed in Table 2. Software engineers should, of course, have knowledge of material from these fields (and the KA descriptions may make reference to them). It is not, however, an objective of the SWEBOK Guide to characterize the knowledge of the related disciplines, but rather what knowledge is viewed as specific to software engineering.

Table 2 Related disciplines.

◆ Computer engineering	◆ Project management
◆ Computer science	◆ Quality management
◆ Management	◆ Software ergonomics
◆ Mathematics	◆ Systems engineering

2.1. Hierarchical Organization

The organization of the KA descriptions or chapters supports the third of the project's objectives – a characterization of the contents of software engineering. The Guide uses a hierarchical organization to decompose each KA into a set of topics with recognizable labels. A two- or three-level breakdown provides a reasonable way to find topics of interest. The Guide treats the selected topics in a manner compatible with major schools of thought and with breakdowns generally found in industry and in software engineering literature and standards. The breakdowns of topics do not presume particular application domains, business uses, management philosophies, development methods, and so forth. The extent of each topic's description is only that needed to understand the generally accepted nature of the topics and for the reader to successfully find reference material. After all, the Body of Knowledge is found in the reference material themselves, and not in the Guide.

2.2. Reference material and Matrix

To provide a topical access to the knowledge–the fourth of the project's objectives–the Guide identifies reference material for each KA, including book chapters, refereed papers, or other recognized sources of authoritative information. Each KA description also includes a matrix relating the reference material to the listed topics. The total volume of cited literature is intended to be suitable for mastery through the completion of an undergraduate education plus four years of experience.

In this edition of the Guide, all KAs were allocated around 500 pages of reference material, and this was the specification the associate editors were invited to apply. It may be argued that some KAs, such as software design for instance, deserve more pages of reference material than others. Such modulation may be applied in future editions of the Guide.

It should be noted that the Guide does not attempt to be comprehensive in its citations. Much material that is both suitable and excellent is not refer-

enced. Material was selected in part because–taken as a collection–it provides coverage of the topics described.

2.3. Depth of Treatment

From the outset, the question arose as to the depth of treatment the Guide should provide. The project team adopted an approach which supports the fifth of the project's objectives–providing a foundation for curriculum development, certification, and licensing. The editorial team applied the criterion of *generally accepted* knowledge, to be distinguished from advanced and research knowledge (on the grounds of maturity) and from specialized knowledge (on the grounds of generality of application). The definition comes from the Project Management Institute: "The generally accepted knowledge applies to most projects most of the time, and widespread consensus validates its value and effectiveness".[1]

<table>
<tr><td rowspan="2">Specialized
Practices used only for certain types of software</td><td>Generally Accepted
Established traditional practices recommended by many organizations</td></tr>
<tr><td>Advanced and Research
Innovative practices tested and used only by some organizations and concepts still being developed and tested in research organizations</td></tr>
</table>

Figure 1 Categories of knowledge

However, the term "generally accepted" does not imply that the designated knowledge should be uniformly applied to all software engineering endeavors–each project's needs determine that–but it does imply that competent, capable software engineers should be equipped with this knowledge for potential application. More precisely, generally accepted knowledge should be included in the study material for the software engineering licensing examination that graduates would take after gaining four years of work experience. Although this criterion is specific to the U.S. style of education and does not necessarily apply to other countries, we deem it useful. However, the two definitions of generally accepted knowledge should be seen as complementary.

[1] *A Guide to the Project Management Body of Knowledge*, 2000 Edition, Project Management Institute, Newport Square, PA. www.pmi.org.

3 The ontology of the SWEBOK from a conceptual and consensus-reaching perspective

This Body of Knowledge is currently organized as a taxonomy subdivided into ten Knowledge Areas designed to discriminate among the various important concepts only at the top level. Of course, the software engineering knowledge is much richer that this high level taxonomy and currently resides in the textual descriptions of each knowledge areas. Such textual descriptions widely vary in style and content. The *conceptual* ontology approach is therefore used to analyze the richness of this body of knowledge, to improve its structuring, and develop consensus on its detailed terminology.
The development of the software engineering domain ontology requires three phases: 1) Proto-ontology construction; 2) Internal validations cycle; 3) External validation (and possibly extension) cycle.

Proto-ontology construction: analysis and extraction (one SWEBOK KA at a time) of the concepts, relations between concepts and axioms (asserted necessary or necessary and sufficient conditions), terms and definitions existing in the SWEBOK Guide and related IEEE and ISO standards. Automatic term extraction tools having as input a corpus of text in natural language have been used to complete the list of concepts and relationships, identified through the analysis of the documents already mentioned.

Internal validation cycle: a series of validation (and possibly extension) cycles, at various instances levels (internal: ETS – UQAM – SPIN, etc.), aiming to build a progressively larger consensus, concerning the elements in the software engineering proto-ontology

External validation cycle: a series of external validation cycles will be required, aided by internationally reputed software engineering domain experts, to build progressively a consensus concerning the concepts, attributes and relations between class/concepts that should be present in the final ontology.

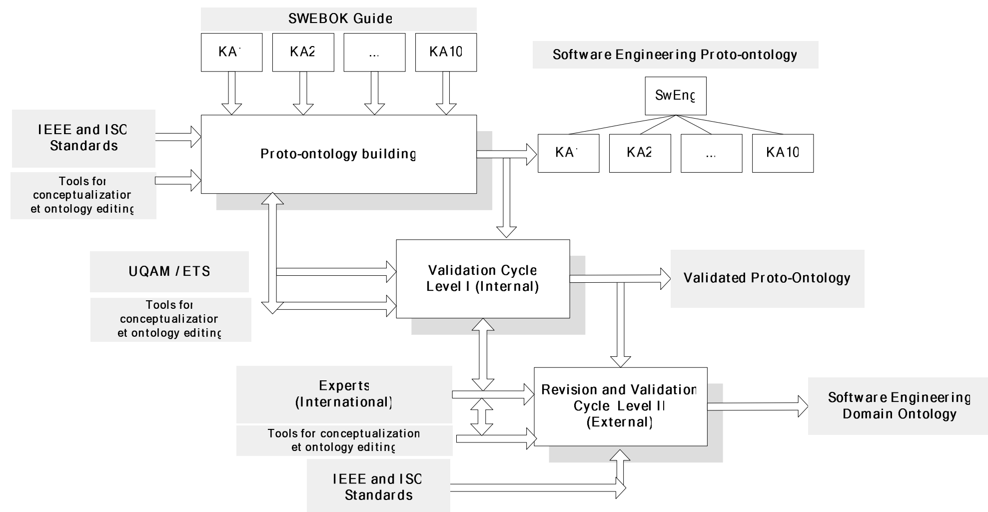

Figure 1 The SWEBOK ontology project phases

The proto-ontology development phase has identified in the SWEBOK Guide over 4,000 concepts, 400 relationships, 1,200 facts as well as 15 principles. Table 1 presents a breakdown by knowledge areas.

The testing maintenance and process knowledge areas include the largest number of concepts and relationships, while the testing and quality include most of the principles identified.

	Relationships	Concepts	Facts	Principles
SWEBOK main structure	4	48	55	0
KA 1 Introduction	0*	0*	0*	0*
KA 02 Software Requirements	24	240	72	0
KA 03 Software Design	44	307	211	2
KA 04 Software Construction	21	214	63	0
KA 05 Software Testing	96	1001	165	7
KA 06 Software Maintenance	44	706	140	0
KA 07 Software Configuration Management	31*	85*	46*	0*
KA 08 Software Engineering Management	33*	72*	46*	0*
KA 09 Software Engineering Process	45	587	134	1
KA 10 Software Engineering Tools and Methods	19	263	62	0
KA 11Software Quality	34	447	61	5
CH 12 Related Disciplines of	12	171	32	0

Software Engineering				
TOTAL	**407**	**4141**	**1087**	**15**

* : partial counting (to be finalized)

Table 1 – Overview of quantity of elements currently in the SWEBOK proto-ontology

The major contributions expected from this approach are: 1) Identification of main inputs, outputs and activities to be performed in order to develop the aimed ontology; 2) Identification of the main software engineering concepts, terms, definitions, relations between classes/concepts (IsA, Part-Whole, and other specifics relationships) and axioms describing the concepts; 3) validation (and possibly extension) of the software engineering ontology; 4), progressive building of a consensus concerning the concepts in the ontology aided by international software engineering domain experts.

Besides the benefits already mentioned in section 1, the use of the "software engineering ontology" which is a result of this project may also contribute to the development of additional content validation by *automatic* cross-correlation validation (besides that which is already done already done continuously by the SWEBOK review team) across the ten areas of knowledge integrated in the SWEBOK Guide. This would ensure that all concepts and definitions are used in a consistent fashion throughout all the SWEBOK's areas of knowledge.

4 The ontology of the SWEBOK as a formal artefact

As it has been said before, the SWEBOK guide provides a foundation for the development of an ontology for Software Engineering, since it is the result of a process of domain expert review and validation, and provides references to other relevant sources. Nonetheless, the process of analysis of the guide to come up with a logical coherent ontology is by no means a simple process. Many of the entities described in the guide to the SWEBOK are complex activities that produce interrelated artifacts. These entities have temporal, material and conceptual facets that should be clearly defined, and which are well-known in existing upper ontologies and large commonsense bases. If the emphasis of ontology is in providing computational semantics to the representation, formal approaches are required beyond the elaboration of consensual meanings as described above. This change in focus can be considered as operational, in the sense that it is a medium towards the end of providing automation or delegating tasks

to agents or software modules. This leads to a very different notion of the ontology development process in which the criteria for inclusion is usefulness for computer-based applications. Such notion is aligned with the current view of the Semantic Web (Berners-Lee, Hendler and Lassila, 2001), which emphasizes the development of a technology based on formal description logics (Baader et al., 2003).

In practice, the formal approach entails that many of the aspects and descriptions in the SWEBOK that may be considered relevant in conceptual approaches are not appropriate for operational ones. For example, a paragraph as the following "Numerous models have been created to develop software, some of which emphasize construction more than others." (page 4-3 of the SWEBOK guide) may be considered appropriate for the narrative of the Guide, but need not a formal representation, since it is simply stating a vague counting about a vague aspect of models. Even in the case that vagueness would be handled somewhat, it is not clear that this provides significant knowledge but an anecdotal statement useful for human readers. In consequence, a formal approach for the ontology of the SWEBOK can not be expected to cover every paragraph, but to extract only relevant, well-defined or well-definable sentences.

There exist proposals for the standardization of upper ontologies (Niles and Pease, 2001) that could be used as a basis for such formal semantics. In fact, the IEEE P1600.1 Standard Upper Ontology Working Group (SUO WG) is working towards that end. Given the past activity of the IEEE and other organizations in producing standards regarding the vocabulary and concepts of Software Engineering, there exists an opportunity to exercise and analyze the discipline from the perspective of upper ontology as a principal case study.

A technique for validating the semantic precision of conceptual schemas is that of providing explicit links to concepts and relations that are already described in a large upper ontology. Concretely, we here consider the *OpenCyc* 0.9 knowledge base. This can be considered as an alternative or a complement to analysis techniques as the Bunge-Wand-Weber (Wand and Weber, 1995) that fosters the reuse of existing open knowledge engineering, and the mapping to modern Web-enabled ontology languages as OWL is a straightforward step.

OpenCyc is the open source version of the Cyc Knowledge Base (Lenat, 1994), which contains over one hundred thousands atomic terms, and is provided with an associated efficient inference engine. Cyc uses as its underlying definition language a variant of predicate calculus called CycL, and it attempts to provide a comprehensive upper ontology of "common-

sense" knowledge. In what follows, some of the main issues in modelling the SWEBOK by linking definitions to OpenCyc are provided. The method used for such process can be roughly described in the following steps:

1. Find one or several terms that subsume the category under consideration.
2. Check carefully that the mapping is consistent with the rest of the subsumers inside OpenCyc.
3. Provide the appropriate predicates to characterize the new category.
4. Edit it in Protègè or other editor to come up with the final formal version.

This process has the advantage of being possible for individual work of an expert. The outcomes of the process can then be contrasted with the work of others. In any case, the process results in much more efficient and structured ontology engineering work, since the argumentation against or in favour of a given concept or predicate is put in the formal context of *OpenCyc*. This makes easier the process of decision making, and avoids the discussion on subjective or personal opinions that are not yet put in formal terms.

5 Fundamental elements of the ontology of the SWEBOK

This section summarizes the main conceptual elements that have been identified during the course of the research work of the authors of this chapter. The elements covered are cross-cutting to many Knowledge Areas of the Guide, and as such, they may be considered as a “high level” conceptual subset that gives coherence to the specifics of each KA. Here only the more pervasive and relevant will be discussed. The exposition goes from the material elements of everyday engineering activities to the representation of prescriptive knowledge, which is by its own nature much more challenging to capture.

5.1. Activities, Artifacts and Agents

Engineering is basically an artefact-producing activity carried out by engineers. At this level, engineering can be seen as a flow of activities, and in an ideal world, every activity, its doer and the artefacts used, changed or created may be represented. This consideration does not care of the ways of doing the activities (the methods) but only of the representation of the

activities as actually enacted. In fact, this is the recording of the actual, real empirical experience of engineering as a human activity. That objectivity makes this a somewhat easier level to be represented. First, the engineers that do the actual work can be characterized as a subset of the class `oc_IntelligentAgent,` defined as “An agent is an IntelligentAgent if and only if it is capable of knowing and acting, and capable of employing its knowledge in its actions”. From an ontological viewpoint, the term `SoftwareEngineer` is not a rigid property (Welty and Guarino, 2001), since being a software engineer is contingent to a work position, and it is not an essential property of the individuals. This leads to the first proposition for the general mapping.

Proposition #1 `SoftwareEngineers` are a class of `oc_IntelligentAgents` (excluding collectives). Software engineering activities will require individuals of this class.

It is important to separate the individual workers from collectives (e.g. organizations or teams). This entails that `SoftwareEngineer` is disjoint with `oc_MultiIndividualAgent-Intelligent`, which concretely address collectives with capability of acting purposefully. Teams of software engineers might be considered relevant since productivity is connected to team dynamics as recognized in software estimation models (Boehm, 1981), but individuals are the unit of responsibility and possess specific competencies or skills that provide them a unique meaning.

Activities are the fabric of engineering work. Activities in OpenCyc can be represented as `oc_Action` instances. These actions are defined as “The collection of oc_Events that are carried out by some “doer” (see oc doneBy). Instances of `oc_Action` include any event in which one or more actors effect some change in the (tangible or intangible) state of the world, typically by an expenditure of effort or energy.” An `oc_Event` is in turn “a dynamic situation in which the state of the world changes; each instance is something one would say ‘happens’.” Going a step further, engineering activities are in fact `oc_PurposefulActions`, “Each instance of PurposefulAction is an action consciously, volitionally, and purposefully done by at least one actor”.

Proposition #2 Actual Software Engineering activities as enacted in software projects are a specific class of `PurposefulActions`, situated in the context of a project that has as its final outcome the creation or modification of a software program.

The term “software program” as a generic, intellectual product can be mapped to oc_ComputerProgram-CW, that are “distinct from computer code and from both running and installed programs.”. The oc_purposeOfEvent predicate can be used to explicitly declare the software-creating purpose. This provides a necessary and sufficient definition to classify SoftwareEngineeringActivity(es). From this definition of activities, the wide array of activities that are commonly identified in software processes can be characterized. Nonetheless, the definition of each kind of activity requires the specification of different aspects, including the kind of engineer, the outcomes and the usual sequence with other kinds of activities. For example, “requirements elicitation” according to the SWEBOK guide is the “first stage” and it is mainly concerned with “getting human stakeholders to articulate their requirements.”

The third class of basic elements of actual engineering practice is the artifacts used, created or changed. An oc_Artifacts is “an at least partially tangible thing which was intentionally created by an oc_Agent (or a group of Agents working together) to serve some purpose or perform some function.”

Proposition #3 The elements used, created and modified in Software Engineering activities are specific kinds of Artifacts.

An important ontological differentiation for artefacts in Software Engineering is that of Documents and its “propositional” content, i.e. the information they contain. This is clear in OpenCyc with the categories of oc_InformationBearingThings and oc_PropositionalInformationThings. This allows a clarification of the difference of the propositional content and the thing that conveys it. For example, a requirements document can be broken in several documents, but the propositional content is unique irrespective of its digital or hardcopy form. When speaking about the software process, the important part is the propositional content, while the concrete things have some degree of arbitrariness in formatting, and they are only important for cataloguing processes specific to each project.

The basic definitions so far provide room for the classification of most of the elements that are present in the SWEBOK Guide in the form of description of activities. However, there are specific elements that should be addressed since they have a special signification in engineering.

5.2. Models, Specifications and Methods

The word model amounts for 297 occurrences in the SWEBOK guide. `Model-Artifact` provides the appropriate semantics for the concept: "a collection of artifacts; a subset of `VisualInformationBearingThing`. Each element of Model-Artifact is a tangible object designed to resemble and/or represent some other object, which may or may not exist tangibly". The `ModelFn` function designates all the models of a given thing, e.g. `ModelFn(SoftwareComponent)`. This is a concrete characterization of models that seems to match all the uses of model in the SWEBOK. As information bearing objects, the models are IBTs also, so that their contents can be represented in a propositional form, through the predicate `containsInfoPropositional-IBT` IBT PIT, that links to a propositional information thing. PITs are in themselves microtheories, thus allowing the definition in logical terms of the actual contents of the model. This could for example be applied to develop systems that represent UML diagrams through logics, which will enable a degree of increased automation.

The Guide to the SWEBOK somewhat differentiates models and artifacts, as in the Software Design KA "The output of this process is a set of **models and artifacts** that record the major decisions that have been taken", but ontologically this distinction is irrelevant.

The word "Specification" appears 138 times in the GUIDE. For example "Requirements *specification* typically refers to the production of a document, or its electronic equivalent, that can be systematically reviewed, evaluated, and approved." The production of a document is an `oc_PurposefulAction`. But the `oc_Specification` itself is a `oc_PropositionalConceptualWork`, that enables a representation of the contents of the specification in logics (different from the "specification document" that is an `oc_InformationBearingThing`).

An ontologically different concept related to activities in SE is that of "methods" for activities, i.e. the normative specification of "blueprints" for potential courses of activity. These specifications have an intrinsic prescriptive character, so that they should not be specified as actions, but rather as specifications.

5.3. Theoretical standpoints and guidelines

There is not currently a uniform or standard form to represent theoretical positions or standpoints in ontology engineering. Further, the science of Software Engineering has not produced a relevant body of theories or laws that explain the discipline, and most of the knowledge is in the form

of guidelines or generic hypotheses. In fact, the SWEBOK Guide does not provide a classification of theories and frameworks according to conventional scientific terms, so that this is an area that is relevant for future revisions. However, some elements backed on empirical evidence are yet referenced in the SWEBOK, and this calls for specific representation techniques. For example, the well-known "laws of software evolution" (Lehman, 1996) require a careful consideration. For the sake of illustration, we will take here the following statement from these laws "An E-type program that is used must be continually adapted else it becomes progressively less satisfactory". This requires the following elements to be addressed:

- First, a characterization of `E-TypeProgram` is required. Computer programs as conceptual works (different from their copies or physical representation) are captured by the generic `oc_ComputerProgram-CW` term. Consequently, types of programs could be defined from such abstraction. E-type programs as "software that implements an application or addresses a problem in the real world" could be characterized by linking them to representations of the problem addressed.
- The representation of the evolution of the program. For this, OpenCyc provides the `oc_programCode` predicate connecting the programs as conceptual entities to `oc_ComputerCode` instances. In turn, these can be subject to a modelling of time-stamped revisions or versions that could be used to assess if a program is being subject or evolution or not. This enables the quantification of the adaptations (and even of its extent in terms of modifications) in the time scale. But the term "continually adapted" is by its nature vague, and some metric or statistical model would be required to assess it from a computational viewpoint.
- The representation of the "use" of a program. This would require a tracking of the lifecycle of the program that in some cases might be difficult, but for reasons outside the representation itself.
- A representation of what "satisfactory" means. This is probably the most controversial issue, since there is not a single universally accepted standard of "satisfactoriness". Satisfaction is usually mentioned as one of the aspects of usability (Van Welie, van der Veer and Eliëns, 1999), but other elements of the "software quality" concepts could also be considered. In addition, satisfaction is often measured through questionnaires or interviews with users, but there is not a standard instrument for it.

If characterizations for the above could be clearly defined, a software agent could be in a position to examine representations of actual software projects and alert of when a program is surely requiring an evolution. An inference rule for the state of "Software-RequiringAdaptation" could be formulated. Further, the provision of ontology-based tools to represent actual software projects could automatically find evidence against the statement.

However, as can be appreciated in the example, this requires the *operationalization* of a number of elements that are only vaguely defined in the original statement. This constitutes a research direction in itself, and is out of the scope of a simple representation of the SWEBOK Guide. An alternative may be that of codifying such kind of statements in a form that is useful for cataloguing and human query, but that do not entail any kind of delegation of tasks or decisions to software. This could be useful but it is not a true representation of knowledge in the area in the sense of having computational semantics. In consequence, this level of theory inside the ontology could be seen as the ultimate goal, but requiring substantial work beyond the formulation of the SWEBOK in ontology languages.

An important element introduced by the disparity of theoretical or methodological standpoints in Software Engineering is that of conflicting knowledge. This is prominent in the diversity of approaches to the software process, but it may also arise in some more specific situations. Following the example above, it might be the case that different positions on what "continuous adaptation" is in term of frequency (or on the definition for "satisfactoriness") lead to incompatible views. This would either require the provision of separate ontologies or the use of a representational mechanism that allows such kind of potential inconsistency or divergence. The concept of *microtheory* in *OpenCyc* provides such representational mechanism, intended to organize assertions that depend on "shared set of assumptions on which the truth of the assertions depends". Definitions inside the same microtheory need to be consistent, but this is not required across microtheories.

Summing up, the logical tools are prepared for the representation of theory or assumptions, but these require more elaboration. Arguably, this could be considered a future requirement for the revision and evolution of the SWEBOK.

6 Conclusions

The SWEBOK represents the outcome of a significant collaborative effort in shaping the scope of Software Engineering as a discipline. The elaboration of knowledge representations about the contents and structure of the guide represents a step further in the clarification of such knowledge, and may also serve as a revision tool for the guide itself (Sicilia et al., 2005). Nonetheless, there are different perspectives that can be taken when developing an ontology of the SWEBOK. These range for conceptual representations that attempt to unveil some conceptual links between parts of the Guide to formal approaches oriented to develop software that automates some task. While the former may take the form of topic maps and can be used for example to provide more graphical parts of the Guide, the latter are only oriented to machine-consumption. Both views and others intermediate or similar serve different purposes, but all of them are important tools for inquiry on the contents of the discipline.

References

Auyang S (2004) Engineering – an endless frontier. Harvard University Press.

Baader, F., Calvanese, D., McGuinness, D., Nardi, D., Patel-Schneider, P. (eds.). (2003). The Description Logic Handbook. Theory, Implementation and Applications, Cambridge.

Berners-Lee, T., Hendler, J., Lassila, O. (2001). The Semantic Web. Scientific American, 284(5), 34-43.

Boehm, B. 1981, *Software Engineering Economics*, Prentice-Hall, 1981

Brooks F (1987) No silver bullet: Essence and accidents of software engineering. IEEE Computer 20(4): 10-19

Brooks, A. (1997) Meta analysis - a silver bullet - for meta-analysts. *Journal of Empirical Software Engineering*, 2:333-338.

Lehman, M. (1996) Laws of Software Evolution Revisited, pos. pap., EWSPT96, Oct. 1996, LNCS 1149, Springer Verlag, pp. 108-124

Niles, I., and Pease, A. 2001. Towards a Standard Upper Ontology. In P*roceedings of the 2nd International Conference on Formal Ontology in Information Systems* (FOIS-2001), Chris Welty and Barry Smith, eds, Ogunquit, Maine, October 17-19, 2001.

Popper K (1959) The Logic of Scientific Discovery. Routledge

Sicilia, M.A., García-Barriocanal, E., Díaz, P., Aedo, I. (2003) A Literature-Based Approach to the Annotation and Browsing of Domain-Specific Web Resources. *Information Research* 8(2)

Sicilia, M.A., Cuadrado, J.J., García, E. and Rodríguez, D. (2005) The Evaluation of ontological representations of the SWEBOK as a revision tool. In *Proceed-*

ings of the First International Workshop on the Evolution of the Guide to the Software Engineering Body of Knowledge in Conjunction with COMPSAC 2005

M. Van Welie, G.C. van der Veer and A. Eliëns (1999). Breaking down usability. *Proc. of Interact'99*, pp 613–620.

Wand, Y.; Weber, R. (1995) On the deep structure of information systems. *Information Systems Journal* (5), 1995, pp. 203-223.

Welty C and Guarino N (2001) Supporting ontological analysis of taxonomic relationships. Data and Knowledge Engineering 39(1): 51-74