Conceptualizing Measures of Required Software Functionality

Concepción López[1], Juan-José Cuadrado[2] and Salvador Sánchez-Alonso[3]

[1] Engineering Department, Francisco de Vitoria University
Ctra. Pozuelo-Majadahonda Km. 1.8 (Madrid), Spain
c.l.rodriguez@ufv.es
[2,3]*Information Engineering* Research Unit
Computer Science Dept., University of Alcalá
Ctra. Barcelona km. 33.6 – 28871 Alcalá de Henares (Madrid), Spain
jjcg@uah.es, salvador.sanchez@uah.es
http://www.cc.uah.es/ie

Abstract. Software functionality expressed in user requirements is a key element for the measurement and planning of the software process. As such, it is important to have an upper model of existing function analysis models as those provided in function point counting methods. This paper discusses an ontological analysis of the concepts related to the specifications of functionality, in the context of existing ontological work on Software Engineering. Concepts from well-known function measurement methods are mapped to existing formal definitions, and the main conceptual pitfalls and fuzzy issues are analyzed.

Keywords: Software functionality, function points, ontologies

1 Introduction

The term *function* is defined in the IEEE Std 610.2 as "(1) A defined objective or characteristic action of a system or component. For example, a system may have inventory control as its primary function.". This sense of function emphasizes the dynamic aspects of software, and is complemented in the normative dimension by the definition of functional requirement as "A requirement that specifies a function that a system or system component must be able to perform". To complement these definitions, functionality is defined in the Merriam Webster's online dictionary as "the particular set of functions or capabilities associated with computer software or hardware or an electronic device". From these definitions it becomes clear that functionality is a matter of behaviors.

However, when considering components, we must consider that the behavior carried out by some of the components in non-trivial software systems are *internal*, in the sense that they come from a process of design and are not directly specified as user requirements. Typically, user-prescribed functionality is mapped in design time to

software components, which expose functionality to other components. Then, required functionality can be defined as the functions at the level expressed in functional requirements, emphasizing that we stay at the level of final user functionality. Even in the case that functional requirements are expressed for a software that has not a user interface (as a *framework*), the distinguishing characteristics of required functionality in the sense provided here is that it **was stated as functional requirements in the context of engineering**. That is, user requirements are *performative*, in the sense that they trigger development activities.

As engineering artifacts, one can expect required functions to be measurable, and as a matter of fact, there is a significant tradition in measuring required functions. *Function Point Analysis* (FPA) is one the most widely used software functional size measurement methods. Since 1984 this method is promoted by The International Function Point Users Group or IFPUG[1]. This group did not produce a measurement standard, but a set of standards and technical documents about functional size measurement methods, known as the ISO/IEC 14143 series. Starting in 1998, a set of experts in software measurement created the Common Software Measurement International Consortium (COSMIC)[2], and proposed an improved measurement method known as Full Function Points. This method becomes the standard ISO/IEC 19761 in 2003 and is also ISO/IEC 14143 compliant.

Function measurement methods rely on some definition of what constitutes software functionality, and obviously the outcomes of the measurement process is dependant on the understanding of such concepts. This is also especially relevant for converting or comparing measures that used a counting method with measures obtained through other one (e.g. from IFPUG to COSMIC and viceversa). Thus, these kinds of measures require a sound conceptual analysis which determines what the things to be counted are. Such kind of analysis was the motivation for the inquiry reported in this paper. Since existing research has dealt with ontologies of software and software engineering (Calero, Ruiz and Piattini, 2006), the point of departure of such an analysis was that of synthesizing and revising existing ontological definitions of software and software function. This was accomplished by using the general `<onto-SWEBOK>` framework as the underlying conceptual structure (Abran et al., 2006). Further, the definitions provided are based on the IFPUG and COSMIC methods for measuring functionality, since they are specific of a user's view of functionality, which is the object of interest here.

The ontological version of a coherent and comprehensive view of software functionality allows for the reconciliation of measurement methods, and also for the development of ontology-driven software configuration tools that depart from abstract representations of functions, and not from source code. It should be noted that Software Configuration and requirement tracing tools deal with functionality at the level of documentation pieces, but do not use models of functionality.

The rest of this paper is structured as follows. Section 2 discusses the main ontological issues of software functionality in connection with the `<onto-SWEBOK>` ontology. Then, conceptual mismatches and some issues of fuzziness in the concepts

[1] http://www.ifpug.org/

[2] http://www.gelog.etsmtl.ca/cosmic-ffp/

described are addressed in Section 3. Finally, Section 4 provides conclusions and outlook.

2 Main concepts and properties

2.1. Conceptualizing software

The view of software that needs to be addressed here is actually that of *software specification*. There has been a considerable debate on the formal/informal form of such specifications – a review of the main original issues can be found in (Colburn, 2000, pp. 129-). However, the discussion here concerns a given specification, assuming it correctly captures the real world entities being modeled and the right user needs. In addition, software has a dual nature concerning its form of expression and its form of execution. But since here we deal only with the inputs and expected outputs of software behavior (be it yet developed, functioning software or only specifications), the discussion is not relevant.

From the viewpoint of the coming discussion, we will start from some definitions on the OpenCyc[3] ontology. OpenCyc is the open source version of Cyc (Lenat, 1995), which contains over one hundred thousands atomic terms, and is provided with an associated efficient inference engine. It attempts to provide a comprehensive ontology of "commonsense" knowledge, including what are usually considered "upper definitions". Figure 1 provides a fragment of the OpenCyc ontology expressed in UML. It depicts a selection of concepts in OpenCyc and their relations (predicates) that are relevant to our present discussion.

[3] http://www.opencyc.org/

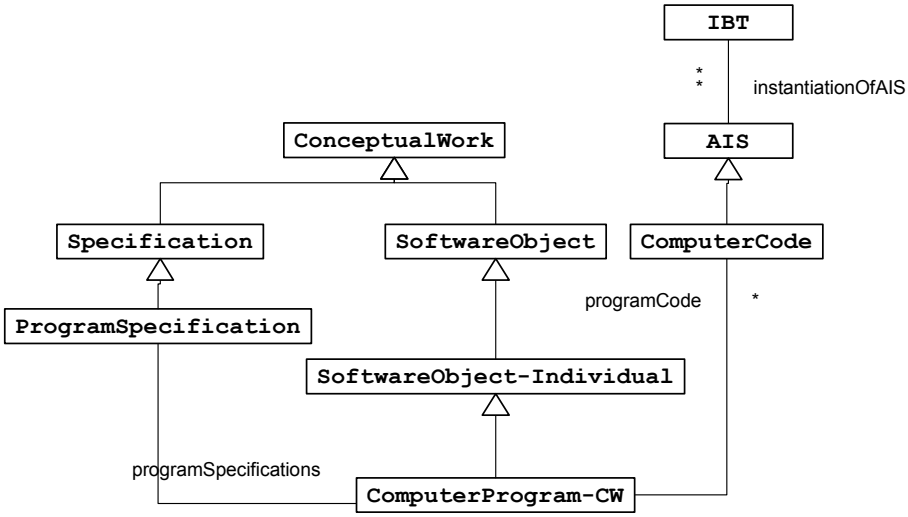


Figure 1. Software and specifications of software.

The main idea in Figure 1 is that there is a class of objects that are `ProgramSpecification`s that determine the expected behavior of `ComputerProgram`s. The OpenCyc concept `ProgramSpecification` is defined as "[…] not a computer program itself (i.e. lines of code), but an abstract characterization of how a program should behave. For instance, a sorting program can be specified by requiring that the program's output be a list of the same elements as the input such that no element follows an element that is greater than it.". Functional specifications are obviously subsumed in that category. A problem arises with the granularity of what a "program" is considered. `ProgramSpecification` instances are no limited to "single, discrete programs", thus, the mapping of computer programs (as conceptual works) and specifications is actually conventional and it cover several cases:

1. A specification covers a number of programs (e.g. in the case of a protocol specification).
2. A specification covers a single program.
3. A specification is only part of a program functionality.

The OpenCyc concept `ProgramStepSpecification` serves to cover only a part of a program to deal with case (3) that is not clearly covered in `ProgramSpecification`.

There is also a relevant distinction that appears in Figure 1 between these computer programs considered `ConceptualWork`s and the `ComputerCode` (be it source or binary) that realizes them. In turn, computer code is actually an abstract information structure (`AIS`) that has as instantiations information bearing things (`IBT`) as computer file copies containing the computer code. For example, some given software

like the statistical package *StatGraphics 7*[4] can be modeled as an instance of `ComputerProgram-CW`. Then, each of the distributions for different platforms can be specified by the following: "the code in which an instance of `ComputerProgram-CW` is expressed constitutes an instance of `AbstractInformationStructure` that can be related to the program it expresses using the predicate `programCode`.".

In summary, `ComputerCode` instances are realizations of `ComputerProgram-CW`, but we are concerned with the specifications of the latter, which in turn are instances of `ProgramSpecification`.

2.2. Static and dynamic aspects

The following Table summarizes a selection of the main definitions in IFPUG and COSMIC as inputs for the measurement process.

Data-related elements	
`COSMIC:OOI` (object of interest)	any physical thing, as well as any conceptual objects or parts of conceptual objects in the world of the user, about which the software is required to process and/or store data
`COSMIC:DataGroup`	a distinct, non empty, non ordered and non redundant set of data attribute types where each included data attribute type describes a complementary aspect of the same object of interest
`COSMIC:DataAttribute`	the smallest piece of information, within an identified data group type, carrying a meaning from the perspective of the software's functional user requirements
`IFPUG:Entity`	a fundamental thing of relevance to the user, about which a collection of facts is kept.
`IFPUG:RecordElement`	a subgroup of data elements within an internal logical file or an external interface file
`IFPUG:File`	a logically related group of data, not the physical implementation of those groups of data.
`IFPUG:DataElement`	a unique user recognizable, non-repeated field.
Process-related elements	
`COSMIC:FunctionalProcess`	an elementary component of a set of functional user requirements comprising a unique cohesive and independently executable set of data movement type; a functional process is complete when it has executed all that is required to be done in response to the triggering event.
`IFPUG:TransactionalFunction`	the functionality provided to the user to process data by an application. Transactional functions are defined as external inputs, external outputs, and external inquiries.

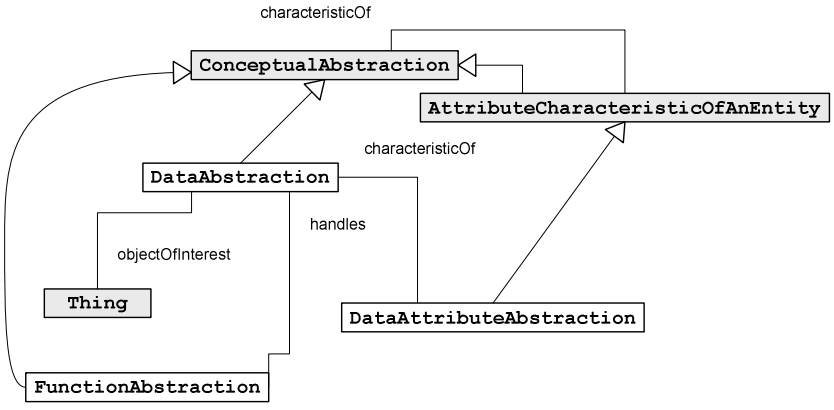
[4] Here we consider concrete versions and not software series.

A straightforward mapping of COSMIC and IFPUG concepts are discussed in what follows, with an analysis of their ontological status.

First of all, entities `COSMIC:OOI` and `IFPUG:Entity` refer to anything that is of interest to users, which encompasses a broad category of things. OpenCyc `Thing` concept encompasses every possible individual entity.

However, the interest in functional specifications is then going to the specific representations of objects of interest. The OpenCyc `ConceptualAbstraction` concept (defined as "general concepts formed by extracting common features from specific examples") captures the main requirements of both `COSMIC:DataGroup` and `IFPUG:File`, even though it does not mandate persistence as a characteristic. Further, its specialization `AttributeCharacteristicOfAnEntity` defined as "abstractions belonging to or characteristics of an entity" subsumes the definitions of `COSMIC:DataAttribute` and `IFPUG:DataElement`. Two additional characteristics of data concepts in IFPUG and COSMIC require additional definition: (1) the fact that data elements/attributes are "unique" or "atomic", and (2) the definition in IFPUG of record elements as parts of files. Both require a definition of a sub-abstraction *part-of* predicate relating parts of the conceptual abstractions.

The dynamic or process related part is considered only in terms of the data elements that are used, communicated and changed. Both methods provide room for expressing the complexity of the computation, but this is kept separate from the other considerations in the method. Then, the important elements in the ontology are Specifications of the input and output parameters, along with the specifications of the data that is consulted and/or modified/created. All of these can be represented through the same conceptual abstraction concepts defined above. The processes in themselves (better, the description of the processes) can also be represented through conceptual abstractions. For preserving their differentiated ontological status, separate concepts for processes and data abstractions can be defined. This will be dealt with later in this paper, but an initial diagram with the main classes is provided in the next Figure.



3 Analyzing conceptual mismatches and fuzzy aspects

Table 2 summarizes the main mappings and the concepts that can be used to underlie them.

Concept	IFPUG concept	COSMIC concept	Shared meaning	Potential fuzziness
`DataAbstraction`	`File`	`DataGroup`	Facets of the facts abstracted from an object of interest.	What is considered an OOI is conventional in relation with an abstraction paradigm.
`DataCharacteristic`	`DataElement`	`DataAttribute`	Components of a higher data level abstraction.	In both cases, the consideration of primitiveness is conventional.
`FunctionAbstraction`	`TransactionalFunction`	`FunctionalProcess`	Movement of data.	Kinds and granularity of functional abstractions. Consideration of processing complexity.

Regarding functional abstractions, the abstractions are characterized by data groups involved – Exit, Entry, Read and Write in COSMIC and EI, EO, EQ in the case of IFPUG. This characterizes adequately the functional specification, but only in the case of IFPUG the complexity of the internal processing is used as a weighting in the counting. In both cases, it is important to question that the processing is important in terms of specification, i.e. different processing (that are obviously different in ontological terms) could be expressed in terms of similar data elements moved. One possibility to introduce that propositional information inside the ontology without introducing purely implementation concerns could be that of specifying the functions in terms of pre- and post-conditions on the data moved, as used in techniques as design by contract (Meyer, 1997). Even though that approach does not directly translate into a counting mechanism, it defines function concisely and allows for the identification of data moved. In other direction, the *granularity* of functions is a concern from an ontological viewpoint. The definitions of common method have a fuzzy point in their reference to the object of interest. For example, in COSMIC, data group types are "distinct, non empty, non ordered and non redundant set of data attribute types where each included data attribute type describes a complementary aspect of the same object of interest". This allows an easy recognition of classes or records as data groups, but leave some uncertainty on which is the criteria for a data group being primitive. This is a major ontological issue which can only be solved by building on an ontology of primitive data types (but not in the sense of conventional programming languages). For example, the consideration that a date or a vCard is an attribute or a data group with attributes inside is conventional, and its primitiveness should be either made explicit or inferred.

In other direction, the kinds of functional abstractions require also further inquiry. The key ontological issue is that different terms are required for each type of functional component that entail some difference in engineering terms. For example,

the difference between an Input and a Save in COSMIC is clearly relevant. However, there is also a relevant engineering distinction in updating a data attribute of an entity or updating the relationship between two entities. This in turn is reflected in some paradigms on operations of different complexity (e.g. in OODBMS updating a link is different from updating a data attribute) and it could even affect what is considered a distinct data abstraction, i.e. is the relationship between two entities a distinct OOI or not? This links with the ontological aspects of data abstractions. The ontological representation requires the *explicit* modelling of the paradigmatic information modelling constructs we use, e.g. object-oriented, logics-based, relational. This is compatible with current ontological languages since it can represent different incompatible but internally consistent paradigms. This is in practice implemented in existing tools that compute semi-automatically function points from UML diagrams.

In addition to the key ontological elements pointed out, there are other differences between COSMIC and IFPUG, as those described by Xunmei, Guoxin and Hong (2006), but they affect scope and the introduction of non functional aspects, which are not a concern in our present discussion.

4 Conclusions and outlook

Software functionality as something that is reified in the form of specifications is an integral part of any software engineering endeavor. As such, the measurement of functionality requires a deep understanding of the object measured. This paper has reported an analysis of such conceptual structure, pointing out to the issues that are the source of potential conceptual mismatches.

From the conceptual structure described here, it becomes apparent that it is reasonable to initiate and inquiry on the possibility of bridging or transforming different function measurement methods, since they are all based in a similar core of concepts.

References

1. Abran, A., Cuadrado-Gallego, J.J., García-Barriocanal, E., Mendes, O., Sánchez-Alonso, S. and Sicilia, M.A. (2006). Engineering the ontology for the Swebok: Issues and techniques. In Calero, C., Ruiz, F. and Piattini, M. (Eds.), *Ontologies for software engineering and software technology* (pp. 103-122). New York: Springer.
2. Calero, C., Ruiz, F. and Piattini, M. (Eds.), *Ontologies for software engineering and software technology*. New York: Springer.
3. Colburn, T. (2000). *Philosophy and Computer Science*. M.E. Sharpe, Armonk, NY.
4. Lenat, D. B. Cyc: A Large-Scale Investment in Knowledge Infrastructure. Communications of the ACM 38(11): 33—38 (1995).
5. Meyer, B. (1997) *Object Oriented Software Construction*, Prentice Hall, 1997, pp. 331-410.
6. Xunmei, G., Guoxin, A. and Hong, Z. (2006) The Comparison between FPA and COSMIC-FFP. In *Proceedings of the Software Engineering Measurement Forum* 2006