An Ontological Analysis from Algorithm to Computer Capability

José M Parente de Oliveira

Divisão de Ciência da Computação – Instituto Tecnológico de Aeronática (ITA)
Pça Mal Eduardo Gomes, 50, CEP 12.228-900 – São José dos Campos – SP – Brazil

parente@ita.br

Abstract. *In software or program ontologies described in the literature, the separation between abstract and implementation views is evident. On the other hand, the ontological entities involved in such views are not very well defined in the context of programs life-cycle, having as parts conception, construction plan, construction process and verification. In many papers the ontological entities described are not very well grounded and they are only assumed as premises without top ontological background. To provide answers to some raised questions, this paper presents a computer program ontology based on the Basic Formal Ontology (BFO) and an interpretation of such an ontology. The ontology and its interpretation show how a computer program ontology is complex and need to advance to attend some theoretical or practical needs.*

1. Introduction

In software or program ontologies described in the literature, the separation between abstract and implementation views is evident. On the other hand, the ontological entities involved in such views are not very well defined in the context of programs life-cycle, having as parts conception, construction plan, construction process and verification.

In many papers the ontological entities described are not very well grounded and such entities are only assumed as premises without top ontological background, without providing an ontological account of the entity's types of computer programs nor a clear meaning for program abstraction and implementation. Based on these general issues, the main questions taken into account here are the following:

- Though some works in the literature [Lando et al., 2007; Oberle, 2009; Duarte et al, 2018] provide a grounded characterization of computer program ontological entities, another important issue is what kind of entities are algorithms, source codes, machine codes, and machine code executions?
- How can we organize an ontology taking into account a life-cycle view of programs?
- What does a machine code installed in a computer mean to such a computer?
- What is the importance of a program implementation in a computer?

To provide answers to such questions, this paper presents a computer program ontology based on the Basic Formal Ontology (BFO) and an interpretation of such an ontology. The ontology and its interpretation show how a computer program ontology is complex and need to advance to attend some theoretical or practical needs.

The paper is organized as follows. In Section 2, there is a presentation on Algorithms, Computer Programs and Ontologies described in the literature. In Section 3 we present the most important elements of Basic Formal Ontology (BFO) used here. In Section 4, we present a proposal of a Computer Program Ontology and its interpretation. Finally, in Section 5, we present the concluding remarks.

2. Algorithms, Computer Programs and Ontologies

The literature on algorithms and computer programs is vast, but the same is not true for program ontologies. Thus in this section, we intend to highlight some fundamental points about algorithms, computer programs and computer ontologies.

According to Rapaport [2019] view, it is interesting to notice that information processing is nothing but symbol manipulation. Arguably, however, it is interpreted symbol manipulation; moreover, not all symbol manipulation is necessarily information in some sense. So, perhaps, although computers are nothing but symbol manipulators, it is as information processors that they will have (are having) an impact.

For Donald Knuth [Knuth, 1973], an effective method (or procedure) is a mechanism that reduces the solution of some class of problems to a series of mechanical steps which is bound to:

- Always give some answers rather than ever give no answer;
- Always give the right answer and never give a wrong answer;
- Always be completed in a finite number of steps, rather than in an infinite number of steps;
- Work for all instances of problems of the class.

For Knuth, an algorithm is an effective method expressed as a finite list of well-defined instructions for calculating a function. Such a method must possess the following properties:

- Finiteness: The algorithm must always terminate after a finite number of steps.
- Definiteness: Each step must be precisely defined; the actions to be carried out must be rigorously and unambiguously specified for each case.
- Input: An algorithm has zero or more inputs, taken from a specified set of objects.
- Output: An algorithm has one or more outputs, which have a specified relation to the inputs.
- Effectiveness: All operations to be performed must be sufficiently basic so that they can be done exactly and in a finite length of time.
- For each problem or class of problems, there may be many different algorithms.
- For each algorithm, there may be many different implementations (programs).

For Moschoyakis [1988], Turing machines capture the notion of mechanical computability of a number of theoretic functions, by the Church-Turing Thesis, but they do not present a mechanical computation to be realized by physical machines. This

means that important aspects of the complexity of computations are not captured by Turing machines. In addition, for Moschoyakis [2001], algorithms are generally identified with *abstract machines,* mathematical models of computers, sometimes idealized by allowing access to "unbounded memory". But for him such a view does not provide a clear way to define algorithms correctly. Moschoyakis [2001] argues that a plausible solution is to see algorithms as defined by recursive functions while physical machines model *implementations,* a special kind of algorithms. In practical terms, it is not clear what an implementation is and how algorithms and implementations are ontologically connected.

In defining what a program is, Suber [1988] argues that a program source code has two forms of representation. The first one is the text-based representation organized according to a high-level programming language, which is readable by programmers. The other form is the representation of the program in the high-level programming language in a standard binary pattern stored in the computer memory or saved on an auxiliary memory device, which is not executable, in the sense that the code does not instruct the machine to do something.

In other words, a computer program can be in three states. The first one is the pattern of a source code. The second state is the pattern of a machine code. Both of them are static stages. The third stage is when a machine code is in a position of instructing the computer to do something. In any of these stages, the algorithm is the guiding entity that is incorporated and represented by them. Nevertheless, despite providing an interpretation of what a computer program is, Suber [1988] view requires an ontological separation of interpretation for the states he described, as is done in the present paper.

Another important aspect in the context of computer programs is the implementation of computational models. Rescorla [2014] argues that computational models are abstract entities which are not located in space or time and do not participate in causal interactions. Under certain circumstances, a physical system *realizes* or *implements* an abstract computational model.

A computational model is an abstract description of a system that reliably conforms to a finite set of mechanical instructions. The instructions dictate how to transit between states. A physical system implements the computational model when the system reliably conforms to the instructions. Thus, a Physical system P realizes/implements a computational model M just in case the computational model M accurately describes the physical system P. In other words, a physical system implements a computational model only if the system can instantiate states belonging to the state space description induced by the model.

Aiming at characterizing a software as a social artifact, Wang et al. [2014] argue that a program is not identical to a code. So, when a code is in the circumstances that somebody intends to produce certain effects on a computer, then a new entity emerges, constituted by the code, which is a computer program. If the code does not actually produce such effects, it is the program that is faulty, not the code. In conclusion, a program is constituted by a code, but it is not identical to a code. A code can be

changed without altering the identity of its program, which is anchored to the program's essential property, which is its intended specification.

In terms of ontology, Lando et al. [2007] proposed a general ontology of programs and software, aiming at using it to conceptualize a sub-domain of computer programs, namely that of image processing tools. Such a proposal used DOLCE as the foundational ontology. With the same intention, Oberle et al. [2009] proposed a reference ontology for software, called the Core Software Ontology, which formalizes common concepts in the software engineering realm, such as data, software with its different shades of meaning, classes, methods, etc. The reference nature of such an ontology aims at clarifying the intended meanings of its concepts and associations. In both works, we do not see how a program evolves from requirement to its implementation,

On the basis of an ontology of software which accounts for the possibility for software to change while maintaining its identity, Wang et al. [2014] defend the view that we need to explicitly account for the identity criteria of various software-related artifacts that are implicitly adopted in everyday software engineering practice. In conclusion, a syntactic structure could be used as an identity criterion of a code, and a program specification along with the intentional creation act could be used as the identity criteria of a program.

Extending their view on program and its identity criteria needs, Wang et al. [2014] propose the following structure:

- A program is constituted by some code intended to determine a specific behavior inside the computer. Such behavior is specified by a program specification.
- A software system is constituted by a program intended to determine a specific external behavior of the machine (at its interface with the environment). Such external behavior is specified by a software system specification.
- A software product is constituted by a software system designed to determine specific effects in the environment as a result of the machine behavior, under given domain assumptions. Such effects are specified by the high level requirements.

Duarte et al. [2018] presented three related domain ontologies: the Software Ontology, an ontology about software nature and execution, and the Reference Software Requirements Ontology, which addresse what requirements are and types of requirements, and the Runtime Requirements Ontology, with the purpose of extending the previous ontologies to represent the nature and context of Runtime Requirements Ontology. They characterized the concept of software, requirement and runtime requirement ontology without going further into the specific concepts related to computer programs.

In discussing problems related to computer program ontologies, Eden and Turner [2007] present a top-level ontology which consists of three categories which can be roughly characterized as follows:

- Metaprograms - contain statements describing programs, such as algorithms, abstract automata, and software design specifications, which consist of

constraints imposed on the structure or behavior of programs such as software metrics and, in particular, descriptions in the literature on software design, including architectural styles, design patterns, and abstract data types. They also include informal descriptions.

- A program is divided into Program Scripts and Program Processes.
  - Program Script - A-temporal entities which consist of well-formed instructions to a given class of digital computing machines, commonly represented as inscriptions or text files.
  - Program Process - Temporal entities that are created by a process of executing (running) a particular program-script in a particular physical setting, also known as operating system processes or 'threads'.
- Program-Hardware contains digital computing machines.

We observe two categories of program-scripts: scripts encoded in a machine language, the category of which is generally referred to as machine code, and scripts encoded in a high-level programming language, the category of which is generally referred to as source code.

In our view, the ontologies described in the literature have the main purpose of clarifying the semantics of the elements of computer programs, as well as the semantic of programs as whole. On the other hand, such ontologies do not provide an account of the semantic of the programs in the context of their life-cycle, once they do not include the entities that represent program conception, construction plan, construction process, and verification. Thus, we argue here that taking into account the life-cycle view as the context for ontologies of program, we open up new possibilities of theoretical and practical studies.

3. Basic Formal Ontology (BFO)

BFO is grounded in the Aristotelian tradition. It is an upper-level ontology, designed to be very small and aiming at consistently representing those upper level categories common to domain ontologies of different fields. BFO is grounded in two broad categories of entities: continuant and occurrent. Figure 1 depicts the entities of BFO here considered. In what follows, we describe such elements taking Arp et al. [2015] as reference.

Continuant entities are those entities that continue or persist through time, preserving their identity through changes, and have no temporal parts.

An **Independent Continuant** is a Continuant entity that is the bearer of qualities. If a continuant entity *a* is the bearer of quality *b*, then we also say that *b* inheres in *a*. Independent continuants are such that their identity and existence can be maintained through gain and loss of parts, and also through changes in their qualities, through gain and loss of dispositions, and of roles.

Material entities have some portion of matter as part. It is thus an independent continuant that is spatially extended in three dimensions and that continues to exist through some time interval.

An **Object** is a Material Entity that is:

- Spatially extended in three dimensions.
- Causally unified, meaning its parts are tied together by relations of connection in such a way that if one part of the object is moved in space then its other parts will likely be also moved.
- Maximally self-connected, which means the different parts of the object are tied together in a certain way and that anything that is tied to these parts in the same way is itself part of the object.

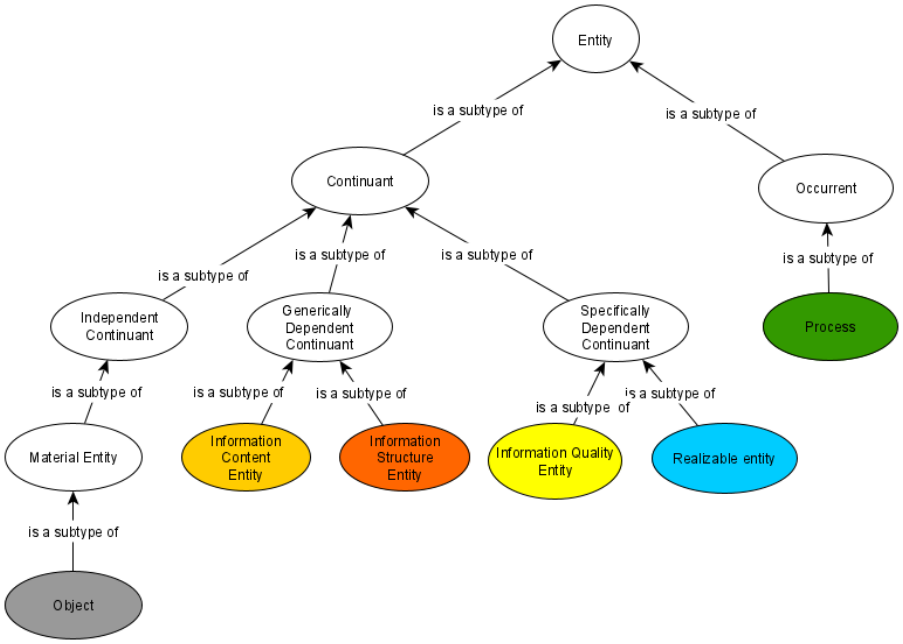

Figure 1 – BFO Entities.

Generically Dependent Continuants are entities that depend on one or other independent continuants that can serve as its bearer. We can think of the generically dependent continuants as complex continuant patterns of the sort created by authors or designers or, in the case of DNA, through the process of evolution. Examples include the Facebook trademark, the pattern that is your own signature, and a square arrangement of sixty-four alternating black and white squares. Each such pattern exists only if it is materialized in some counterpart specifically dependent continuant. Such a relation in BFO is technically called "concretized by".

Information Content Entities (ICEs) provide information about something in reality. They have this something as a subject; they represent, mention or describe this something; they inform us about this something [Smith et al., 2013]. There are physical entities that are created or modified to serve as bearer of certain patterned arrangements of ICE. For example, ink or other chemicals, electromagnetic excitations [Smith et al., 2013].

Information Structure Entity (ISE). An ISE is a structural part of an ICE; speaking metaphorically, it is an ICE with removed content: for example, an empty cell in a spread-sheet; a blank Microsoft Word file. ISEs thus capture part of what is involved when we talk about the 'format' of an information artifact [Smith et al., 2013].

A **Specifically Dependent Continuant** is a continuant entity that depends on one or more specific independent continuants for its existence. Specifically dependent continuants are said to inhere in other independent continuants, their bearers, and they cannot migrate from one bearer to another. Examples of specifically dependent continuants include the color of a given tomato, and the pain in my left elbow.

An **Information Quality Entity (IQE)** is the pattern on an IBE in virtue of which it is a bearer of some information. An IQE is a quality of an IBE which exists in virtue of such patterned arrangements and which is interpretable as an ICE or ISE. Such an IQE is created when a physical artifact is deliberately created or modified to support it (patterned to serve as its bearer) [Smith et al., 2013].

A **Realizable Entity** is defined as a specifically dependent continuant that has at least one independent continuant entity as its bearer, and whose instances can be realized, in the sense of being manifested, actualized, executed, in associated processes in which the bearer participates.

Occurrents are those entities that occur, happen, unfold, or develop in time, usually referred to as events, processes or happenings. Occurrents are either processes that unfold in successive phases, or they are the instantaneous boundaries of processes, such as their beginnings or ends, or even the temporal and spatiotemporal regions that such entities occupy.

A **Process** is an occurrent entity that exists in time by occurring or happening, has temporal parts and it always depends on some material entity. The dependence here is analogous to that between a specifically dependent continuant and its independent continuant bearers. Examples of processes include the life of a given organism, the course of a disease, the process of cell division, and the fall of water down a waterfall.

4. A Computer Program Ontology Interpretation

In this section we present the proposed ontology used as reference to interpret computer programs and a section to discuss such interpretation, due to the important aspects to be discussed.

4.1. The Proposed Ontology

In software or program ontologies described in the literature, the separation between abstract and implementation views is evident. On the other hand, some of the ontological entities involved in such views are not very well defined.

For Duncan [2017], who used BFO for reasoning about the distinction between software and hardware, “a software program is a specification that consists of one or more programming language instructions and whose concretization is embodied by an artifact that is designed so that a physical machine may read the concretized instructions, whereas hardware is an artifact whose functions are realized in processes that directly or indirectly bring about the result of some calculation.” Such a reasoning was an important source of inspiration for the present paper.

Thus on the basis of gaps identified in the literature, Duncan work, the author’s experience in computer science, his participation in the Industrial Ontological Foundry, and his use of BFO, we came up with the ontology depicted in Figure 2. Such ontology

extends the ontology presented in Figure 1, now extending the lower levels types of entities. In what follows, the several entities are described.

Program Requirement Statement. A Directive Information content entity that prescribes a process or the product needs an agent has.

Specification. A Directive Information Content Entity that prescribes some parts, features, or some outcomes of a planned process.

A Plan Specification is a specification with action specifications and objective specifications as parts. When concretized, it may be realized in a process performed by an agent to achieve the prescribed process endpoints by taking the prescribed actions.

Program Specification. A Plan Specification entity that prescribes some parts, features, or some outcomes of a program. The program specification derives from the Program Requirement Statement.

A Verification Report is an Information Content Entity that presents the program variables with the corresponding values obtained during a program execution.

A Data Item is an information content entity that presents the program variables with the corresponding values inserted or obtained during a program execution.

An Algorithm is a Plan Specification entity that prescribes some parts, features, or some outcomes of a Planned Process. An Algorithm is a Machine-Independent Plan Specification.

A **Source Code** is a Plan Specification entity that prescribes some parts, features, or some outcomes of a planned process formulated according to a programming language syntax. A Source Code is used by programmers in a language syntax, and internally saved in a computer as a binary code.

A **Program Verification Plan** is a Plan Specification entity that prescribes how a program should be verified, taking into account the program specifications.

Algorithm Elaboration is a Process that has as input a Program Specification and as output an Algorithm.

A **Programming Process** is a Process that has as input an Algorithm and a programming Language Syntax, and as output a Source Code.

Source Code Compilation is a Process that has as input a Binary Code, a Computer as a participant, and as an output a Machine Code.

Program Execution is a Process that has as input a Data Item, a Computer as a participant which inheres a Machine Code, and it has as an output a Data Item.

Program Verification is a Process that has as input a Program Verification Plan, a Program Execution, and a Machine Code, and as output a Verification Report.

A **Binary Code** is an Information Quality Entity (IQE), which is a quality of an information bearing entity, which exists in virtue of such patterned arrangements and which is interpretable as an information content entity. Binary code is the binary representation of the source code before its derivation to machine code by means of the Compilation Process.

A Machine Code is an Information Quality Entity (IQE), which is a quality of an information bearing entity, which exists in virtue of such patterned arrangements and which is interpretable as an information content entity. When a Machine Code inheres in the computer, the computer gains the Capability of executing the Machine Code. Such a Capability is realized during the Machine Code Execution. Another aspect that deserves a comment is the extreme case of specifying a program directly in binary code or machine code. Though this is possible in reality, in practical terms it is not usual. Because of that, to keep the consistency of the proposed ontology, this extreme case is not considered as possible.

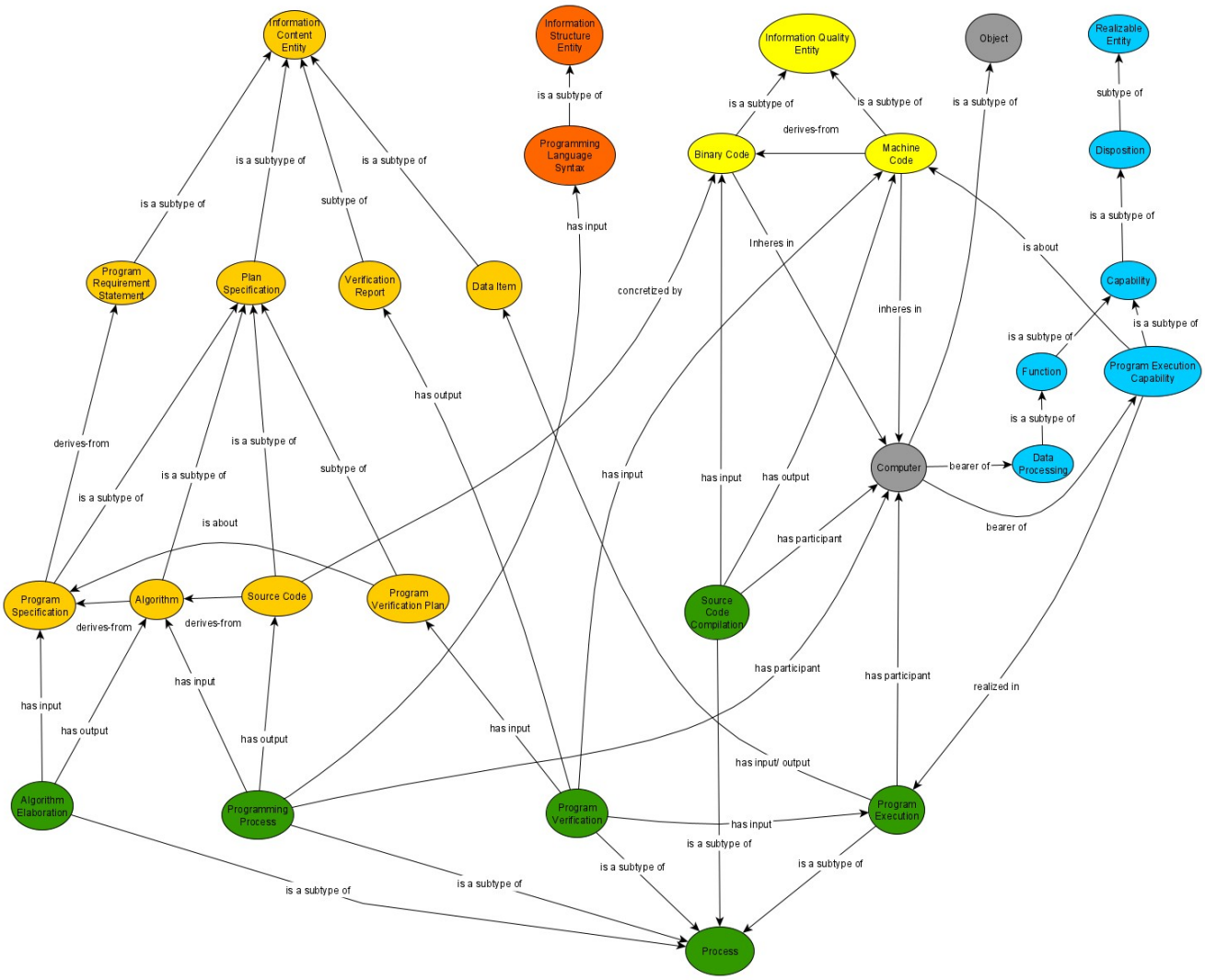

Figure 2 – A Computer Program Ontology.

A **Computer** is an Object, which can inhere a Binary Code or a Machine Code, and it is the bearer of Data Processing and Program Execution Capability. A Computer has physical features and abstract machines. Inheres in and bearer of are inverse relations. Their use depends on the circumstances to offer the best clarity.

A **Disposition** is a Realizable Entity in virtue of which a process of a certain kind occurs, it can occur or it is likely to occur, given appropriate triggers, in the

independent continuant in which the disposition inheres. This process is called the realization of the disposition. The trigger might consist in the objects being placed in a certain environment or being subjected to certain external influence, or it may be some internal event within the object itself. A disposition ceases to exist when its bearer is physically changed. In other words, a disposition exists because of certain features of the physical make-up of its bearer.

Capability is a Disposition that under normal circumstances brings benefits to its bearer, user, or owner [Smith, 2019]. Capabilities come with dimensions along which we can grade their realizations on a scale from zero to positive. In the normal range realizations bring more or less benefits in proportion to their grade on the scale, but there is often a normal range outside which benefits turn into disbenefits.

Program Execution Capability is a Capability borne by a computer about a Machine Code inhered in the Computer. Each Program installed with all the required software or hardware in the Computer provides the Program Execution Capability. Without having the program installed in it, the computer does not hold such capability.

A **Function** is a Capability that exists in virtue of the bearer's physical make-up, and this physical make-up is something the bearer possesses because of how it came into being either through natural selection or intentional design. That means to say that these entities in question came into being to perform activities of a certain sort, called functioning. A Function is a capability that an object has because it was designed or selected to have it: the function of a heart = to pump blood; the function of a pump = to pump; the function of a screwdriver = to drive screws. Every function is associated with a type of process whose instances are realizations of that function (pumping, heating, etc.).

As already mentioned, **Data Processing** is a Function whose information processing is nothing but symbol manipulation. Arguably, however, it is interpreted symbol manipulation; moreover, not all symbol manipulations are necessarily information in some sense. So, perhaps, although computers are nothing but symbol manipulators, it is as information processors that they will have an impact. As a subtype of function, Data Processing exists in virtue of computers make-up, and this physical make-up is something computers possess because of how it came into being through intentional design.

4.2. The Ontology Interpretation

The presented ontology provides a very general view of the entities involved in a computer program, including Information Content Entities, Processes, Information Structure Entity, Information Quality Entities, Objects and Realizable Entities. For having being put together, they provide a coherent existence of such entities.

A way to analyze such an ontology coherence is to take a look from left to right, which shows the temporal evolution of a program and its corresponding consequences. The right most entity is the Program Execution Capability, whose realization is graded on a scale that reflects the Program Requirement Statement achievements, or the Program Verification Plan verified by the Program Verification Process.

The Source Code does not provide any kind of action to the computer. On the other hand, the Machine Code can be executed by the Computer, meaning the computer

generates the internal machine state transitions. In other words, the Machine Code stored in the Computer provides it with the capabilities which are the Program Requirement Statements that can be realized when the Machine Code is executed.

It is important to mention here that in order to realize a Capability, a Computer has to be the bearer of certain qualities and other additional capabilities that play the role of enablers for the capability of interest of an organism at a certain time. Thus computers with programs installed in them have the capability to run the programs.

Going back to the graded realization on a scale of Program Execution Capability, we noticed it reflects the degree of benefits its exercise brings. The highest value in the scale is achieving all the Program Requirement Statements, while some program failures, not yet identified in program verification processes, can also be seen as impediments for the realization of capabilities during a program execution. The graded realization provides a rich way to interpret program executions, opening up a wide spectrum for analyzing program constructions and executions.

It is also interesting to notice that the entity type "Program" does not appear directly in the ontology, though entity types such as "program verification", "program execution" and "program specification" do. The reason for that is that here a program is seen as a sequence of representations in the context of a life-cycle view: program specification, algorithm, source code, binary code, and machine code. Such representations are made possible by means of a series of associated processes, which realize the inherent function and capability. In other words, a computer program is a sum of these representations, and in BFO context we do not have a type of entity to represent it.

In summary, we can say that the proposed computer program ontology provides a grounded characterization of computer program ontological entities based on the top ontology Basic Formal Ontology (BFO). In addition, the proposed ontology offer a grounded meaning for important entities such as algorithms, source codes, machine codes, and machine code executions, and how they are related. Finally, the ontology brings the meanings for a machine code installed in a computer and the importance of a program implementation in a computer. As all these aspects have not been clearly discussed in the literature, the proposed ontology opens up the possibility of new discussions on several aspects of computer program.

5. Conclusion

Computer Program Ontology has been a very difficult topic in the literature. The proposed ontologies are very different in terms of content and structure.

The guiding questions used here highlight some aspects to be carefully taken into account when designing a computer ontology, specially when having a very clear purpose and need. When just developing an ontology without considering those the reader can be led to a deadlock. The proposed ontology and its interpretation provide a resourceful way to advance and define more detailed aspects for the use of a Computer Program Ontology.

As already mentioned, computational models are abstract entities, which are not located in space or time, and they do not participate in causal interactions. Under certain circumstances, a physical system aims at realizing or implementing abstract

computational model. Which are those circumstances? What is it for a physical system to realize a computational model? When does a concrete physical entity implement a given computation? All these aspects, as well as all those related to a whole computer program ontology, are not ontologically clearly described in the literature.

The presented ontology and its interpretation provide a grounded and organized way for answering the fundamental questions raised in the introduction. Such questions raise important aspects not taken into account in many ontology works.

In other words, the presented ontology and its interpretation provide a grounded way to identify the most important elements involved in a computer program ontology and also a practical guideline to identify the elements to be taken into account and collect representative data in future works. For instance, the ontology provides an encouraging view to developers to think carefully about the information transformations and processes along the program life-cycle to develop methods and tools to improve program development. Another thing is to use this ontology to improve program verification processes, be it testing or formal verification. Formal program verification is an ongoing project in our group.

Acknowledgments. I particularly thank the important support provided by the *National Council for Scientific and Technological Development* – CNPq.

References

Arp, R.; Smith, B.; Spear, A. D.: Building Ontologies with Basic Formal Ontology. Cambridge, MIT Press, 2015.

Duarte, B. B; de Castro Leal, A. L.; de Almeida Falbo, R.; Guizzardi, G.; Guizzardi, R. S. S., souza, V. E. S. Ontological foundations for software requirements with a focus on requirements at runtime. IOS Press, 2018: 73 – 105.

Duncan, W. D. Ontological distinction between hardware and software. IOS Press, Applied Ontology, 2017, 12, 5-32.

Eden, A. H.; Turner, R. Problems in the Ontology of Computer Programs. Applied Ontology, Vol. 2, No. 1 (2007), pp. 13–36. Amsterdam: IOS Press. Amsterdam, The Netherlands: IOS Press.

Knuth, D. The Art of Computer Programming Fundamental Algorithms, vol. 1, 2nd Ed. Reading: Addison-Wesley, 1973.

Lando, P.; Lapujade, A.; Kassel, G.; Fürst, F. TOWARDS A GENERAL ONTOLOGY OF COMPUTER PROGRAMS. In: ICSOFT (PL/DPS/KE/MUSE), 2007, pp. 163-170.

Moschovakis, Y. N. On founding the theory of algorithms. In: H. G. Dales and G. Oliveri (eds.) Truth in mathematics. Clarendon Press, Oxford 1998, pp. 71-104.

Moschovakis, Y. N. What Is an Algorithm? B. Engquist et al. (eds.), Mathematics Unlimited 2001 and Beyond. Springer-Verlag Berlin Heidelberg, 2001.

Oberle, D.; Grimm, S.; Staab, S. S. An Ontology for Software. Staab and R. Studer (eds.), Handbook on Ontologies, International Handbooks on Information Systems, DOI 10.1007/978-3-540-92673-3, Springer-Verlag Berlin Heidelberg, 2009.

Rapaport, W.J. (2019). Philosophy of computer science. Current draft in progress at http://www.cse. buffalo.edu/~rapaport/Papers/phics.pdf.

Rescorla, M. A theory of computational implementation. Synthese 191, 1277–1307 (2014). https://doi.org/10.1007/s11229-013-0324-y

Smith, B. et al. IAO-Intel - An Ontology of Information Artifacts in the Intelligence Domain. In: Proceedings of the Eighth International Conference on Semantic Technologies for Intelligence, Defense, and Security, Fairfax, VA, (STIDS 2013), CEUR, vol. 1097, 33-40.

Smith, B. The Ontology of Capabilities. Slides presented in an Industrial Ontology Foundry Meeting, September, 2019.

Suber, P. What is Software? The Journal of Speculative Philosophy, New Series, Vol. 2, No. 2 (1988), pp. 89-119Published by: Penn State University PressStable URL: http://www.jstor.org/stable/25668234 .Accessed: 15/06/2014 07:16.

Wang, X.; Guarino, N.; Guizzardi, G.; Mylopoulos, J. Towards an Ontology of Software: a Requirements Engineering Perspective. In: Frontiers in Artificial Intelligence and Applications, September 2014.