

Towards Defining Computer Capability*

José M Parente de Oliveira^{1,*}

¹*Aeronautics Institute of Technology, Pça Mal Do Ar Eduardo Gomes 50, São José dos Campos, 12228-900, Brazil*

Abstract

Computer program and algorithm are terms that are becoming more and more pervasive in almost every activity in society. On the other hand, as many people use them interchangeably, it is important to have a well-grounded understanding of such terms and their differences. Also, in such a realm, a term not well discussed is capability, which is a well known term used in different contexts, but which is not very well characterized in the context of computers and programs. Capability can be seen as a potentiality that under normal circumstances brings benefits to its bearer, user, or owner, or, in other words, capabilities are what an entity is able to do in virtue of its material constitution as something positive, describing their realizations in terms of achievement and success whose realization someone has or had an interest. Thus, as abstract entities such as algorithms and programs have no capabilities, then it makes sense to talk about computer capabilities, instead of program capabilities. Thus, the main questions taken into account here are the following: (i) what does computer capability related to programs mean?, (ii) how can we sketch computer capability related to programs? Accordingly, this paper presents a domain ontology to answer the above questions.

Keywords

Computer Program, Program Ontology, Computer Capability, Basic Formal Ontology (BFO)

1. Introduction

There is nothing new in saying that software, computer program or algorithm are terms that are becoming more and more pervasive in almost every activity in society. On the other hand, as people in general use them interchangeably, it is important to have a well-grounded understanding of such terms and their differences.

From the computer science standpoint, algorithm manifestation refers to program execution. But for its execution, there are several information representations and transformations. Ontologically speaking, a well-grounded description provides an account of all the aspects involved in algorithms and programs.

As discussed in [1], in software or program ontologies described in the literature, the separation between abstract and implementation views is usual [2, 3, 4, 5]. On the other hand, the ontological entities involved in such views are not very well defined for all entities, nor are the relationships and processes involved in the characterization of a program.

FOUST VII:7th Workshop on Foundational Ontology, 9th Joint Ontology Workshops (JOWO 2023), co-located with FOIS 2023, 19-20 July, 2023, Sherbrooke, Québec, Canada

*Corresponding author.


✉ parente@ita.br (J. M. P. d. Oliveira)

🌐 <http://www.ita.br/~parente> (J. M. P. d. Oliveira)

🆔 00000-0002-7803-1718 (J. M. P. d. Oliveira)



© 2023 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

 CEUR Workshop Proceedings (CEUR-WS.org)

A well known term used in different contexts is capability, but which is not very well characterized in the context of computers and programs. We can say that capability is a potentiality that under normal circumstances brings benefits to its bearer, user, or owner [6]. We can also say that capabilities are what an entity is able to do in virtue of its material constitution as something positive, describing their realizations in terms of achievement and success whose realization someone has or had an interest [7]. Thus abstract entities such as algorithms and programs have no capabilities.

Following this view, then it makes sense to talk about computer capabilities, instead of program capabilities. Thus, the main questions taken into account here are the following:

- What does computer capability mean?
- How can we sketch computer capability?

By now, we can say that computers have the function of manipulating symbols, in particular binary ones, as a way to process information, whilst computer capabilities are the potentials computer have to process information as stated in executable codes derived from source codes written in programming languages, as well as the level of realization of such potentials.

Thus, this paper presents a domain ontology that uses the Basic Formal Ontology (BFO) as the foundational ontology, which aims at providing some grounds to answer the above questions. The choice for the use of BFO is due to its possibilities for representing information content and structure entities, as well as the possibilities for representing the concretization of different types of information. The paper is organized as follows. Section 2 presents key aspects described in the literature. Section 3 presents the most important elements of Basic Formal Ontology (BFO) used here. Section 4 presents the proposed ontology. Section 5 presents an informal assessment of the competency questions and a comparison with related works. Finally, Section 6 presents some concluding remarks.

2. Key Aspects in the Literature

It is important to start this section presenting the context of computer programs: software and hardware. According to the norm ISO/IEC/IEEE 24765 2017-09, a software is composed of “computer programs, procedures and possibly associated documentation and data pertaining to the operation of a computer system.” A computer program is a “combination of computer instructions and data definitions that enable computer hardware to perform computational or control functions.” An operating system is system software that manages computer hardware, software resources, and provides common services for computer programs.¹

The literature on algorithms and computer programs is vast, but the same cannot be said for program ontologies. Thus in this section, we intend to highlight some fundamental points about how algorithms, computer programs, and other aspects are described in computer program ontologies, as well as point out that the notion of computer capability is missing in computer program ontologies.

According to [8], information processing is nothing but symbol manipulation. However, not all symbol manipulation is necessarily information processing in some sense. So, perhaps,

¹https://en.wikipedia.org/wiki/Operating_system

although computers are nothing but symbol manipulators, it is as information processors that they have an impact on several sectors of society. Nevertheless, to play the role of information processor, computers need to run programs that have an algorithm implicit in the instructions stated in a specific program language.

In describing the notion of algorithms, Knuth [9] argues that an effective method (or procedure) is a mechanism that reduces the solution of some class of problems to a series of mechanical steps. Then on the basis of such steps, Knuth defines an algorithm as an effective method that is expressed as a finite list of well-defined instructions for calculating a function.

In analyzing algorithms in terms of Turing machines, Moschovakis [10] says 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, algorithms are generally identified with abstract machines, which can be seen as mathematical models of computers, which are sometimes idealized by allowing access to "unbounded memory" [11]. Thus, as for him such a view does not provide a clear way to define algorithms correctly, it is necessary to have a clear separation and representation of algorithms and their implementations.

Anyway, as algorithms are the foundation of computer programs, they deserve a place in computer program ontologies, although it is not a common rule.

Along these lines, we can say that a program source code or high level code has two forms of representation [12]. 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. The source code by its turn is translated into a machine code, which is in a position of instructing the computer to do something. Although it is a necessary discussion, it does not bring new insight on how algorithm, source code, and machine codes can provide a deeper view of computer programs.

Another relevant aspect in the context of computer programs is the implementation of computational models. Such models are abstract entities that are not located in space or time and do not participate in causal interactions [13]. Under certain circumstances, a physical system realizes or implements an abstract computational model. Inevitably, some questions arise. What does it mean to implement an abstract computational model in ontological terms? Can we say that such an implementation is a program or software?

Also, we can argue that a software is a social artifact and that a program is not identical to a code [5]. 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 this view, 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. A fundamental

question here is what are the ontological grounds to see a program as a social construction?

In a discussion about problems related to computer program ontologies [14], we found a top-level ontology which consists of three categories which can be roughly characterized as follows: (i) 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; (ii) A program is divided into Program Scripts and Program Processes. Program Script consists of well-formed instructions to a given class of digital computing machines, commonly represented as inscriptions or text files. Program Process is a temporal entity that is created by a process of executing (running) a particular program-script in a particular physical setting, also known as operating system processes or 'threads'; (iii) Program-Hardware contains digital computing machines, which allow the realization of program processes.

Also, in ontological terms [2], we found 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. Along the same line, there is 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. [3].

Thus, such ontologies have the purpose of clarifying the intended meanings of important concepts and associations, but they do not offer a clear notion of what a program is in its most fundamental nature.

We see an attempt of expanding the concept of program and software [4], when three related domain ontologies are proposed: the Software Ontology, an ontology about software nature and execution, the Reference Software Requirements Ontology, which addresses 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. Despite the advancements, a discussion on concepts related to computer programs is missing.

It is important to say that the notion of Disposition used in [4] is different from ours. In their view, Program Copy Execution is the event of the physical manifestation represented as the complex disposition Loaded Program Copy that inheres in the Machine. In addition, though not called disposition, some of the aspects related to non-functional requirements, which are considered as qualities, in our view, could be seen as capabilities, as we do here, and the notion of Monitoring Runtime Requirement could be seen as a means for checking the realization of capabilities.

In a previous work [1], we presented a computer program ontology based on the Basic Formal Ontology (BFO) and an interpretation of such an ontology. The idea was to show how a computer program ontology is complex and the need to advance it to attend some theoretical or practical needs. Though the term computer capability was mentioned, we did not provide a deeper clarification on that.

Thus, to sum up this section, we can say that the ontologies described in the literature have the main purpose of clarifying the meaning of the elements of computer programs, as well as the meaning of programs as a whole. On the other hand, such ontologies do not provide a clear ontological account of computer capabilities. The main motivation to advance this concept is to

support the meaning of computer program and capability, as well as point out possible practical use of them.

3. Basic Formal Ontology (BFO)

BFO is an upper-level ontology, 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. In what follows, we describe the entities of BFO that are relevant for the present work, taking [15] as the main 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. 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 are Independent Continuant Entities that have some portion of matter as part. They are spatially extended in three dimensions and continue to exist through some time interval.

An **Information Bearing Entity** (IBE) is a material entity that has been created to serve as a bearer of information. IBEs are either self-sufficient material wholes, or proper material parts of such wholes. An example of a self-sufficient material whole is the memory unity of a computer [16, 17].

Generically Dependent Continuants are Continuant entities that depend on one or more 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. Each pattern exists only if it is materialized in some counterpart specifically dependent continuant, more specifically a quality, which will be defined further.

Information Content Entities (ICEs) are Generically Dependent Continuants that provide information about some portion of reality. An ICE is thus conceived as an entity which is about something in reality and which can migrate or be transmitted (for example through copying) from one entity to another [17]. In other words, ICEs inform us about something [18, 16].

An **Information Structure Entity** (ISE) is the structural part of an ICE. An important example here of ISE is the syntactic structure of a programming language, which governs the structure of instructions in that language in a program. ISEs thus capture part of what is involved when we talk about the 'format' of an information content entity [16].

A **Specifically Dependent Continuant** is a continuant entity that depends on one or more specific independent continuants for its existence [17]. There are two types of Specifically Dependent Continuants: Quality and Realizable Entity.

Qualities are what things are by virtue of the way they are qualified [19]. Qualities inhere in independent continuants, which means that for a quality to exist some other independent continuants must also exist. Examples of qualities include the memory capacity and the processing speed of this computer.

An **Information Quality Entity** (IQE) is a Quality that is the concretization of some Information Content Entity (ICE) [16]. An IQE is a quality of an Information Bearing Entity which is created when a physical artifact is deliberately created or modified to support it [16]. For example, computers are created to be bearers of binary codes that represent programs written in a specific language.

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

Occurrences are those entities that occur, happen, unfold, or develop in time, usually referred to as events, processes or happenings. Occurrences 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. Examples of processes include the elaboration of an algorithm, the compilation of the source code in a programming language, and the execution of a machine code installed in a computer.

4. Computer Program Ontology

Thus to provide the grounds to answer the two previously mentioned questions, we elaborated a computer program ontology which is a refinement and to some extent an extension of the ontology proposed in [1], but now with focus on capability.

A considerable part of the rationale for the ontology comes from the view that some phenomena in reality are perceived and represented, and certain invariants are described in terms of some concepts [20, 21]. Such concepts are used in the definition of an algorithm, which is translated into a source code written in a programming language, which then is compiled to produce the machine code that can run on a computer. Many aspects about such concepts that were discussed in Section 3 along with the concepts presented in Section 2 played an important role in the elaboration of the ontology presented here.

We also defend the view that a computer executing a machine code has a set of possible capability dimensions, which can be put on scales of realizations. We argue that seeing some computer capabilities as the potentials computers have to process information as stated in executable codes, as well as the level of realization of such potentials, are useful information either for program correction or improvement, or for end-users.

In what follows, the proposed ontology that supports this view is presented in two parts for the sake of clarity. We present the entity types as being a subclass of a superior class and with the corresponding “differentia” for their characterization.

4.1. From Requirements to Machine Code

This part of the ontology represents the entity types from the requirement definition to the machine code, as shown in Figure 1. As previously mentioned, Information Content Entities (ICE) are patterns that exist only if they are materialized in some counterpart specifically

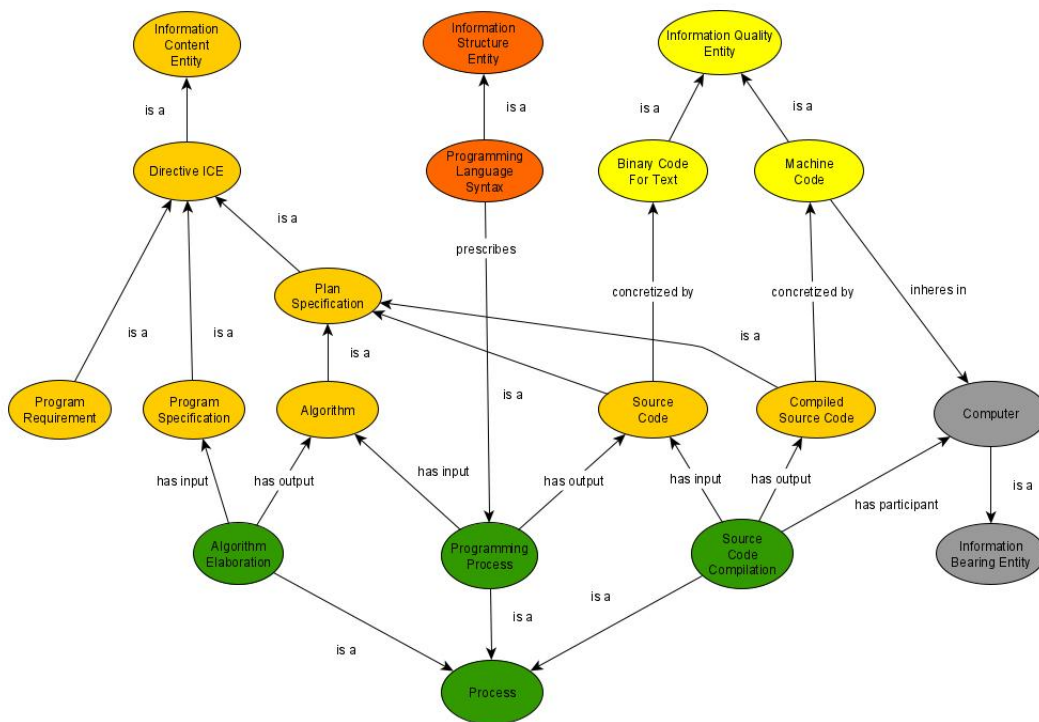


Figure 1: From Requirements to Machine Code Part of the Ontology.

dependent continuants. ICE has two major types: Directive ICE and Descriptive ICE, but only Directive is of interest in this section.

A **Directive ICE** is an Information Content Entity which has the purpose of specifying a plan or method for achieving something.

A **Program Requirement** is a Directive Information Content Entity that describes what a program should do, the functionalities it provides and the constraints on its operation. According to the IEEE Standard Glossary of Software Engineering Terminology, a requirement is “A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document.” As will be discussed later on, the term Capability is central in the proposed ontology.

A **Program Specification** is a Directive Information Content Entity that describes the effects a program is expected to produce. There is no commitment to how the effects will be produced. The program specification derives from the Program Requirement.

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

A **Plan Specification** is a Directive Information Content Entity with the specification of actions and objective as parts. In other words, a Plan Specification prescribes the expected endpoints to be achieved by means of the prescribed actions.

An **Algorithm** is a Plan Specification for some sort of calculation that terminates after a

finite number of well-defined steps, whose actions to be carried out must be rigorously and unambiguously specified so that they can be done exactly and in a finite length of time, as well as has zero or more inputs and one or more outputs. This definition of Algorithm is fully in accordance with Knuth's view [9].

A **Programming Process** is a Process that has as input an Algorithm and a Programming Language Syntax, and as output a Source Code, having a programmer as a participant in the process.

A **Programming Language Syntax** is an Information Structure Entity (ISE) which describes the syntactic structure of a programming language and governs the structure of instructions in programs written in that language.

A **Source Code** is a Plan Specification of an algorithm written according to a programming language syntax. A source code based on an imperative language, such as C, is composed of functions, which contain instructions and variables. A source Code can be read by programmers, but it is internally saved in a computer as a binary code.

For the sake of clarity, the next three definitions are not depicted in Figure 1.

A **Program Function** is part of a Source Code composed of a sequence of instructions grouped as a unit, responsible for performing a computation related to some specified effects that a program is expected to produce.

An **Instruction** is part of a function responsible for specific actions. The main instruction types are attribution, selection (if...then...else), and iteration such as "for" and "while".

As part of an Instruction, a **Variable** is a container for different types of data, such as integer, float, String, etc. The variable name is used to refer to the variable itself and to the stored value. This separation of name and content allows the name to be used independently of the exact information it represents. The value of a variable may change during the course of program execution.²

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 a Source Code before its translation to Machine Code by means of Compilation Process. Though not represented in Figure 1, but there is a link *inheres in* between Binary Code and Computer.

Source Code Compilation is a Process that has as input a Source Code, a Programmer and a Computer as participants, and a Compiled Source Code as the output.

A **Compiled Source Code** is a Plan Specification of a Source Code that can be expressed in a machine-interpretable form ready to be executed by the computer. The Compiled Source Code is concretized by a Machine Code.

A **Machine Code** is an Information Quality Entity that inheres in the computer and is ready to be executed by the computer.

A **Computer** is an Information Bearing Entity, in which a Binary Code or a Machine Code can inhere, and is the bearer of Data Processing Function and Program Execution Capability. A Computer has several qualities, such as physical features. Inheres in and bearer of are inverse relations. The specific use depends on the circumstances that offer the best clarity.

²[https://en.wikipedia.org/wiki/Variable_\(computer_science\)](https://en.wikipedia.org/wiki/Variable_(computer_science))

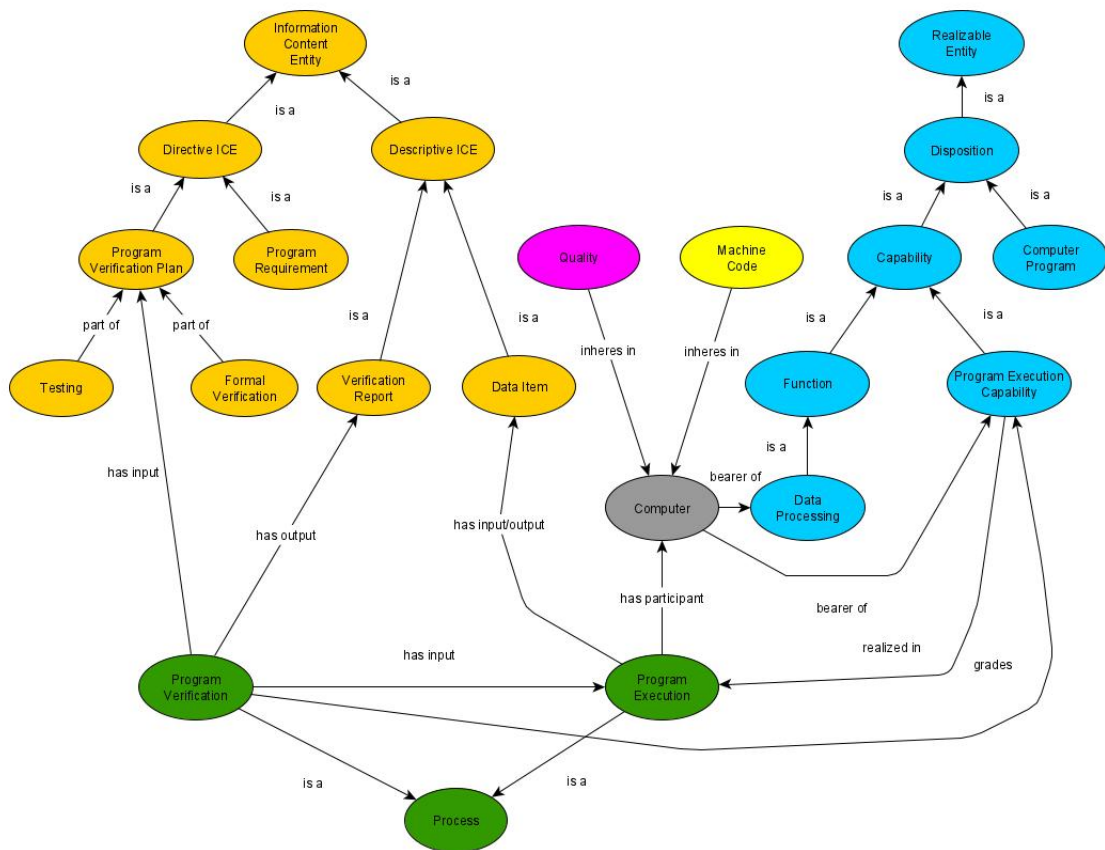


Figure 2: From Program Verification Plan to Capability Part of the Ontology.

4.2. From Program Verification Plan to Capability

This part of the ontology represents the entity types from the Program Verification Plan to the Capability entity, as shown in Figure 2.

A **Program Verification Plan** is a Directive Information Content Entity that prescribes how a program should be verified, taking into account the program requirement and specification. It is important to mention here that the Program Verification Plan establishes the baseline to assert the capabilities to be provided by a computer in relation to a program.

Testing is a Program Verification Plan intended to verify whether the program satisfies the stated requirements and to identify possible defects. The program is executed with test case data to analyze the program response to the test data.

Formal Verification is a Program Verification Plan intended to prove program correctness through the use of formal methods. Formal methods are a set of techniques based on logic, mathematics, and theoretical computer science, which are used for specifying, developing, and verifying programs and hardware [22]. Deductive verification and formal static analysis are some of the approaches that can be applied in formal program verification.

Descriptive ICE is an Information Content Entity that consists of a set of propositions that describe some entity.

A **Verification Report** is a Descriptive Information Content Entity that presents the program function calculations and variables with the corresponding values obtained during a program execution to indicate code correctness.

A **Data Item** is a Descriptive Information Content Entity that presents the program variables with the corresponding values inserted or obtained from a program execution.

Program Execution is a Process that has as input Data Items and a Computer as participant, and Data Items as output. As said before, the Machine Code is an Information Quality Entity that inheres in the Computer, and such an inheritance corresponds to its installation on a computer.

Program Verification is a Process that has a Program Verification Plan and a Program Execution as input, and a Verification Report as output. The Program Verification Process encompasses Testing and Formal Verification processes. Program Verification is an area that deserves further ontological investigation.

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. Such a 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 is a potential that exists because of certain features of the physical make-up of its bearer [7]. For example, a computer has the disposition to execute different machine codes installed in it.

A preliminary definition for **Computer Program**, which requires further investigation, is a disposition of the sort of “total complex of dispositions” which involve combinations of roles and dispositions on the part of individuals and the communities in which it applies [18]. In this view, for instance, translating an algorithm into a code in a program language needs competence (disposition) on the part of a programmer and also a disposition in a wider community of programmers to interpret, correct or improve the code. Expanding this example, we can identify other individual and community based dispositions along the sequence of representations of a program, which are assumed to preserve its identity based on historical continuity and sameness in origin [23]. Furthermore, we can say that each person’s linguistic competence is a disposition and that a language itself is something like the sum total of the linguistic competences of all its users [18].

Capability is a Disposition that under normal circumstances brings benefits to its bearer, user, or owner [6]. Capabilities are what an entity is able to do in virtue of its material constitution as something positive, describing its realizations in terms of achievement and success whose realization someone (or some organism, or some group of organisms) has or had an interest. Thus abstract entities such as algorithms have no capabilities in the sense intended here [7]. We go further saying that source and machines codes have no capabilities as well. Capabilities come with dimensions in a way we can grade their realizations on a scale from zero to positive, reflecting the degree of benefit its exercise brings. 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” [6]. When a Machine Code inheres in a computer,

the computer gains the Capability of executing the Machine Code. Such a Capability is realized during the Machine Code Execution.

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. In other words, a Function is a capability that an object has because it was designed or selected to have it. For instance, the function of a computer is to process data. Thus, artifacts are entities which exist because they were designed intentionally to realize a certain disposition. This disposition that it was designed to realize is that artifact's function [7].

Data Processing is a Function which is nothing but symbol manipulation. Moreover, not all symbol manipulations are necessarily information processing in some sense. Although computers are nothing but symbol manipulators, it is as information processors that they will have an impact. As a sub type 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.

Program Execution Capability is a Capability borne by a computer due to a Machine Code inhered in such a computer. Here, a Machine Code inhering in a computer means the installation of such a code in a computer equipped with all the required software and hardware for its correct functioning. With the installation of the Machine Code, the computer acquires capabilities related to the code, named here Program Execution Capability dimensions, whose realizations during the code execution can be graded on a scale from zero to positive. That means to say that when a capability is outside its range, it ceases to be a capability and its benefits turn into "disbenefits", which implies in a different disposition. Thus, as part of the sketch of Program Execution Capabilities, the following dimensions are proposed here:

D1 - Execute Machine Code. For a machine code to run, the computer must have the appropriate software and hardware configuration. Thus, *D1* is driven by the run time verification of the required computer qualities. The point here is that the machine code needs to know if the computer has the required supporting hardware and software qualities. We defend the view that the program should incorporate such an information in it, in much the same way proposed in [4].

D2 - Comply with Program Specifications. This dimension has the purpose of informing if the computer running the machine code is doing what it is expected to do. This is driven by the program testing stated in the Verification Report which contains the results of integration testing carried out. This dimension allows a program monitor to check the degree of achievement of the program specifications, in other words, it is a run time verification of program specification.

D3 - Execute Program Functions. This dimension indicates the degree of correctness of the program's functions. It is based on the program's functions testing, and source code formal verification. The Verification Report contains the results of testing and formal verification of the program functions.

D4 - Produce Accurate Output. This dimension has the purpose of informing how accurate is the output of the running machine code. For that, the machine code has to be tested under distinct circumstances. An example of inaccurate output is given by a sport watch when it displays different measures for the same walked distance under distinct weather conditions.

Based on such dimensions, we sketched the definition of computer capability produced by a machine code installed in it. For that, we refer to Figures 1 and 2, in which we see that program (p) requirements are translated into program specifications (s), which describe the effects the program is expected to produce. The source (sc) and machine (mc) codes are derived from the specifications and installed in the computer (c) that is supposed to have the required software and hardware qualities (q); the source and machine codes contain the defined program functions (f). Thus, for D1-Execute Program Machine Code, we have a vector of pairs (quality item, capability value) that encompasses qualities related to hardware and software of a computer c running a machine code mc:

$$\mathbf{q} : \{(q_1, qv_1), (q_2, qv_2), \dots, (q_l, qv_l)\} \quad (1)$$

where each q_i represents a computer quality whose values are described as follows:

$$qv_i = x, 0 \leq x \leq max, 1 \leq i \leq l \quad (2)$$

where max is the greatest value in the capability scale. Thus the capability related to D1 can be given by the average of the individual quality values:

$$Cap - D1 = Avg(qv_i). \quad (3)$$

Now we can describe the capability related to D2 - Comply with Program Specifications. For that, we have a vector of pairs (functional specification item, capability value), in which each functional specification s_i is verified by means of functional tests:

$$\mathbf{s} : \{(s_1, sv_1), (s_2, sv_2), \dots, (s_m, sv_m)\} \quad (4)$$

Accordingly, each functional specification has a specific capability value described as follows:

$$sv_i = x, 0 \leq x \leq max, 1 \leq i \leq m. \quad (5)$$

Thus, the capability related to D2 is given by the average of capability values sv_i :

$$Cap - D2 = Avg(sv_i). \quad (6)$$

With respect to D3 - Execute Program Functions, we have a vector of pairs (program function item, capability value), in which each program function f_i is verified by means of functional tests and source code formal verification:

$$\mathbf{f} : \{(f_1, fv_1), (f_2, fv_2), \dots, (f_n, fv_n)\}. \quad (7)$$

As for the previous dimensions, each program function has a specific capability value described as follows:

$$fv_i = x, 0 \leq x \leq max, 1 \leq i \leq n. \quad (8)$$

Thus, the capability related to D3 is given by

$$Cap - D3 = Avg(fv_i). \quad (9)$$

Finally, in addition to the capability D3 related to the degree of correctness of program functions, it is important that such program functions produce accurate output. This is the object of Capability D4 - Produce Accurate Output, in whose representation we have a vector of pairs (program function item, accuracy capability value), in which each function has a related accuracy value (fa):

$$\mathbf{fa} : \{(f_1, fa_1), (f_2, fa_2), \dots, (f_o, fa_o)\} \quad (10)$$

The function accuracy values are described in the same way as before:

$$fa_i = x, 0 \leq x \leq \max, 1 \leq i \leq o. \quad (11)$$

Therefore, the capability related to D4 is given by

$$Cap - D4 = Avg(fa_i). \quad (12)$$

Thus, we finish the proposed sketch for defining the above mention capability dimensions. For us, it is an important step for future refinement and improvement of such a proposal.

5. Informal Assessment of Competency Questions and Comparison with Related Works

In this section, we present a informal assessment of the posed competency questions, as well as a general comparison of the proposed ontology with other related works. But before starting such discussions, it is important to highlight that the proposed ontology is a support for reasoning about the notion of capability, without the need of instantiating it, which is something we plan to do in future works for better representation of programs and the numerical values of the considered capability dimensions.

5.1. Answering the Posed Competency Questions

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 to permit the definition of computer capability based on programs. Thus a computer with a machine code installed in it acquires the capabilities defined here in terms of four dimensions. In our view such dimensions open up new possibilities for further work to extend the used dimensions. Thus, with the ontology and the sketch presented, we answer the two posed questions about the meaning of computer capability related to programs and a sketch of capability.

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 capabilities.

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

the maximum capability in each dimension. On the other hand, capabilities can vary from zero to certain intermediate values, which means lower levels of capability realizations.

It worths emphasizing that the entity type "Program" is seen here as a "total complex of dispositions" which involve combinations of roles and dispositions on the part of individuals and the communities in which it grow up and live. This view accommodate the program's identity along its lifetime. This definition sheds new light on the concept of computer program.

From the practical point of view, if the dimensions of capabilities and their corresponding values were made available previously or at runtime, different users could benefit from that. At previous time of use, users would have information to decide to use or not a program, in consequence of a sense of trustworthiness or not on the program. At runtime, depending on the surrounding circumstances, users could assess if they could trust or not the results provided by the program. Thus, it is pretty feasible to say that knowing about the involved capabilities can make a difference under certain circumstances.

5.2. General Comparison of the Proposed Ontology with Related Works

As previously mentioned, the ontologies described in the literature have the main purpose of clarifying the meaning of the elements of computer programs, as well as the meaning of programs as a whole. On the other hand, such ontologies do not provide a clearer ontological account of computer capabilities.

From the works analyzed, the only one that mentions a term related to capability, the term disposition, is [4]. On the other hand, in their view, a Program Copy Execution is the event of the physical manifestation represented as the complex disposition Loaded Program Copy, that inheres in the Machine. Therefore, from a broader perspective, the term capability has not been used for computer capabilities.

6. Conclusion

In this paper, we aimed at answering the following two questions: (i) what does computer capability related to programs mean?, (ii) how can we sketch computer capability related to programs? To do that we developed a domain ontology based on BFO.

The presented ontology and its interpretation provide a grounded and organized way for answering the questions, shedding light on 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 important elements involved in a computer program ontology and also related to computer capabilities related to program that implies in informing the program users not only about what a program produces as an output, but also how well it produces such an output and if the computer is appropriate or not for running the program.

We can also say that computer capabilities related to programs should be communicated in a clear way, something that is missing in program ontologies. We could also say that the lack of defining computer capability related to programs can lead to serious consequences, specially for critical programs. As a final comment, the ontology presented 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.

Though the proposed ontology is based on BFO, we believe that the notions of programs as social constructions and capabilities as something computers acquire in consequence of having machine codes installed in them can be adapted to other foundational ontologies. As UFO and DOLCE offer wide options for representing abstract and information types of entities, as well as social entities, the notions exploited in the present work can be further investigated with the use of these other foundational ontologies.

As future works, we intend to give a thorough discussion of the view of programs as total complex of dispositions, improve the capability dimensions, mainly those related to verification processes, be it testing or formal verification, and carry out some experiments. Another line of future work is to develop a capability model for semantic web services and exploit its application of capability to database and machine learning, including different kinds of mining tasks and algorithms.

Acknowledgments

We particularly thank the support provided by the National Council for Scientific and Technological Development (CNPq), the Institute of Airspace Control (ICEA) and Department of Airspace Control (DECEA) through the SWIM Project, as well as the invaluable comments and suggestions made by FOIS and FOUST reviewers.

References

- [1] J. M. P. de Oliveira, An ontological analysis from algorithm to computer capability, in: Proceedings of the XIII Seminar on Ontology Research in Brazil and IV Doctoral and Masters Consortium on Ontologies (ONTOBRAS 2020), CEUR, 2020, pp. 48–60.
- [2] P. Lando, A. Lapujade, G. Kassel, F. Fürst, Towards a general ontology of computer programs, in: Proceedings of the Second International Conference on Software and Data Technologies - PL/DPS/KE/WsMUSE, SciTePress, 2007, pp. 163–170.
- [3] D. Oberle, S. Grimm, S. S. Staab, An ontology for software, in: Handbook on Ontologies, International Handbooks on Information Systems, Springer-Verlag, Berlin, 2009.
- [4] B. B. Duarte, A. L. de Castro Leal, R. de Almeida Falbo, G. Guizzardi, R. S. S. Guizzardi, V. Souza, Ontological foundations for software requirements with a focus on requirements at runtime, IOS Press (2018) 73–105.
- [5] X. Wang, N. Guarino, G. Guizzardi, J. Mylopoulos, Towards an ontology of software: a requirements engineering perspective, in: Proceedings of the Conference Formal Ontology in Information Systems, Rio de Janeiro, 2014, pp. 317–329.
- [6] B. Smith, Slides on the ontology of capabilities, 2019. Presented in an Industrial Ontology Foundry Initiative Meeting.
- [7] E. Merrell, D. Limbaugh, P. Koch, B. Smith, Capabilities, PhilPapers (2022). <https://philpapers.org/rec/MERC-14>, obtained on January 15th, 2023.

- [8] W. J. Rapaport, Philosophy of computer science, 2019. [Http://www.cse.buffalo.edu/~rapaport/Papers/phics.pd](http://www.cse.buffalo.edu/~rapaport/Papers/phics.pd).
- [9] D. E. Knuth, The Art of Computer Programming: Fundamental Algorithms, volume 1, 2nd ed., Addison-Wesley, Reading, 1973.
- [10] Y. N. Moschovakis, On founding the theory of algorithms, in: Truth in mathematics, Clarendon Press, Oxford, 1998, pp. 71–104.
- [11] Y. N. Moschovakis, What is an algorithm?, in: Mathematics Unlimited - 2001 and Beyond, Springer-Verlag, Berlin, 2001.
- [12] P. Suber, What is software?, Journal of Speculative Philosophy 2 (1988) 89–119.
- [13] M. Rescorla, A theory of computational implementation, Synthese (2014) 1277–1307.
- [14] A. H. Eden, R. Turner, Problems in the ontology of computer programs, IOS Press, Applied Ontology 2 (2007) 13–36.
- [15] R. Arp, B. Smith, A. D. Spear, Building Ontologies with Basic Formal Ontology, MIT Press, Cambridge, MA, 2015.
- [16] B. Smith, T. Malyuta, R. Rudnicki, W. Mandrick, D. Salmen, P. Morosoff, D. K. Duff, J. R. Schoening, K. Parent, 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 (STIDS 2013), volume 1097, CEUR, Fairfax, 2013, pp. 33–40.
- [17] B. Smith, W. Ceusters, Aboutness: Towards foundations for the information artifact ontology, in: Proceedings of the Sixth International Conference on Biomedical Ontology (ICBO), volume 1515, CEUR, Lisbon, 2015, pp. 1–5.
- [18] B. Smith, W. Kusnierczyk, D. Schober, W. Ceusters, Towards a reference terminology for ontology research and development in the biomedical domain, in: Proceedings of the Conference on Knowledge Representation in Medicine 2006, volume 222, CEUR, Baltimore, 2006, pp. 57–66.
- [19] Aristotle, Organon - Translation from Greek, additional texts and notes by Edson Bini, 3rd ed., Edipro, São Paulo, SP, 2019. In Portuguese.
- [20] N. Guarino, Formal ontology and information systems, in: Proceedings of Formal Ontology and Information Systems (FOIS'98), IOS Press, Trento, Italy, 1998, pp. 3–15.
- [21] N. Guarino, Slides on ontological analysis and conceptual modelling, 2014. 2nd IAOA Ontology Summer School, Vitória, Brazil.
- [22] P. Baudin, F. Bobot, D. Bühler, L. Correnson, F. Kirchner, N. Kosmatov, A. Maroneze, V. Perrelle, V. Prevosto, J. Signoles, N. Williams, The dogged pursuit of bug-free c programs: The frama-c software analysis platform, Communications of the ACM 64 (2021) 56–68.
- [23] N. Irmak, Software is an abstract artifact, Grazer Philosophische Studien (2012) 5–72.