Making Ontological Sense of Hardware and Software

By William Duncan, University of Buffalo
wdduncan@buffalo.edu
December 31, 2009

Abstract

The distinction between hardware and software is an ongoing topic in philosophy of computer science. We use the terms “hardware” and “software” to refer to distinct entities, but upon examination it becomes unclear exactly what distinguishes the two. Furthermore, two authors, James Moor and Peter Suber have cast doubt on the idea that there is a distinction. James Moor has argued that the distinction between hardware and software should not be given much ontological significance. Peter Suber has argued that hardware is software. I find both these positions implausible. For, they ignore more general ontological distinctions that exist between hardware and software. In this paper I examine the arguments of Moor and Suber, and show that although their arguments may be valid, the arguments draw implausible conclusions because they are based on narrow ontological perspectives. The ontological distinctions used to motivate their arguments are not applicable to reality in general. I then argue that by using a more general ontology framework distinctions do emerge between hardware and software. That is, when hardware and software are considered within an ontology which has wide applicability to reality, hardware and software are very different things. The ontology I have used for this purpose is the Basic Formal Ontology (BFO). The paper concludes by demonstrating that in BFO hardware is an independent continuant, whereas software is a generically dependent continuant.

Keywords: Ontology, Philosophy of Computer Science, Hardware, Software

1. Introduction

How do we distinguish hardware from software? Answering this question is more difficult than you might think. Often, we distinguish the two by asserting that software is modifiable whereas hardware is not. But this cannot be right. With the right equipment and expertise, hardware is modifiable also. Perhaps you may wish to qualify this further and assert that software is intended to be modified whereas hardware is not. However, this is also wrong. Many commercial software products are not intended to be modified. What about distinguishing the two by asserting that software is portable whereas hardware is not? Again, we run into difficulties. Many of the components in a computer system can be removed and placed in another computer system.

Given the difficulty in distinguishing hardware and software, some have doubted that there is a distinction. James Moor has argued that the distinction between hardware and software should not be given much ontological significance. Peter Suber has argued that hardware is software. I find both of these positions implausible. For, they ignore more general ontological distinctions that exist between hardware and software. That is, Moor and Suber in their arguments do not consider the fundamental categories which describe reality in general. Rather, by focusing solely on these certain aspects of hardware and software, they draw implausible conclusions. However, when we consider hardware and software from a general ontological perspective, distinctions do emerge between hardware and software. That is, when hardware and software are considered within an ontology which has wide applicability to reality, hardware and software are very different things.

The layout of this paper is as follows. First, I will outline and discuss the arguments of Moor and Suber. Second, I will discuss what an ontology is. Third, I will give a brief description of the ontology I am going to use: Basic Formal Ontology (BFO). Last, I will demonstrate how when using BFO hardware and software are distinguished. Namely, in BFO, hardware is an independent continuant, whereas software is a generically dependent continuant.

2. Moor's argument: The distinction is pragmatic.

In his article "Three Myths of Computer Science", James Moor argues that the distinction between hardware and software should not be given much ontological significance. Instead he thinks that we should take a "pragmatic view of the software/hardware distinction."[1] What Moor means by this will be explained below.

His argument is based on two assertions. The first is that a computer program, is "a set of instructions which a computer can follow (or at least there is an acknowledged effective procedure for putting them into a form which the computer can follow) to perform an activity."[2] The second is that computers programs are to be understood on two different levels:[3]

1. The symbolic level which consists of the instructions to a computer. That is, the symbols we use to represent the kinds of actions a computer is to take when given an instruction.
2. The physical level which consists of the various media on which computer instructions are physically stored (such as floppy disks, CD's, magnetic tape, and so on).

For example, on the symbolic level a flow chart may be a computer program if it represents some set instructions a computer can follow. It does not matter if a computer can, at present time, actually read the flow chart. It is what the flow chart represents which makes it a computer program. The physical level of this program would then be result of burning the instructions to a CD as series of pits and lands.

[1] James H. Moor, "Three Myths of Computer Science", *British Journal for the Philosophy of Science* 29(3) (September 1978): 215.
[2] Moor, 214.
[3] Moor, 213.

When we distinguish between hardware and software, though, either the physical level or symbolic level computer programs is often overlooked. The physical level of computer programs is overlooked when we focus only on the on the symbolic level of computer programs. At the symbolic level, computer programs consist of instructions a computer can follow. These instructions are thought be part of software, and not a part of hardware. Hardware is taken to be "the physical units making up a computer system", while software (i.e., computer programs) is something separate from the computer system.[4] This characterization of hardware and software, however, misses two important facts about computer programs. First, many early computers were programmed by throwing switches or setting wires. The computer programs of these early computers were part of the hardware; not separate. Second, modern digital computers usually store computer programs internally, and, thus, are part of the physical structure of the hardware.[5]

The symbolic level of computer programs, on the other hand, is overlooked when we consider only the physical level of computer programs. At the physical level, software is taken to be the part of computer system which we can change. However, this distinction between hardware and software cannot be consistently maintained. For, depending on the context, the part of a computer system that a person can change may vary. For example, a person with expertise in circuit design may be able to modify a computer system's mother board or CPU. Components normally considered to be hardware (i.e., not modifiable) by others.

Given that computer programs are best understood as having both symbolic and physical levels, Moor advocates that we view the distinction between software and hardware as being "pragmatic":[6]

[4] Moor, 215.
[5] Moor, 215.
[6] Moor, 215.

> [Since] programming can occur on many different levels, it is useful to understand the software/hardware distinction as a pragmatic distinction. For a given person and computer system the software will be those programs which can be run on the computer system and which contain instructions the person can change, and the hardware will be that part of the computer system which is not software. At one extreme if at the factory a person who replaces circuits in the computer understands the activity as giving instructions, then for him a considerable portion of the computer may be software. For the systems programmer who programs the computer in machine language much of the circuitry will be hardware. For the average user who programs in an applications language, such as Fortran, Basic, or Algol, the machine language programs become hardware. For the person running an applications program an even larger portion of the computer is hardware.

In stating that the software/hardware distinction should be understood as "pragmatic", Moor is focusing on the practical activity of computer programming. That is, the activity of computer programming as based on practice, not theory. The ways in which people perform this activity, however, is not uniform. What counts as a computer instruction or which instructions can be changed is dependent upon the person doing the programming and the type of programming being performed. This dependency relationship between a computer instruction and the activity of computer programming, then, makes distinguishing software and hardware difficult. For, at the symbolic level, computer instructions play a role in programming activities related to both software and hardware. At the physical level, the computer instructions in both hardware and software can both be modified. Thus, Moor concludes we should not read too much into the software/hardware distinction. For, as he states, "What is considered hardware by one person can be regarded as software by another."[7] It is better to understand the hardware/software distinction as, in his term, a pragmatic matter.[8]

[7] Moor, 216.
[8] To be precise, Moor states, "This pragmatic view of the hardware/software distinction makes the distinction both understandable and useful" (Moor 215). I take this statement to imply that Moor intends that this pragmatic distinction is better.

2.1 Discussion of Moor's argument

Moor's argument consists of an analysis of software and hardware at both the physical and symbolic level. Thus, my discussion of him will proceed in the same manner. First, let us consider the physical level. Moor's observation that, on the physical level, the ability to modify a computer instruction is dependent on the computer programmer and type of programming activity is insightful. For, in certain contexts, the ability to modify something can depend on the person who performs the activity. Consider two home owners. One is skilled in the art of home construction. The second is not. For the first home owner, certain aspects of the house may be modifiable, while for the second these may be fixed. For example, the first homeowner may be able to add another bathroom. In this particular context, then, the distinction between what is and what is not modifiable depends on the homeowner who is performing the activity.

I will grant that Moor may be correct about this aspect computer programs. However, even with this concession, his overall position has a flaw. He asserts that since the modifiability of computer programs is dependent upon the computer programmer, the distinction between entities which contain computer programs (i.e., software and hardware) is also dependent upon the computer programmer. I believe this is a mistake. For, although activities and their outcomes are intimately linked, activities which depend upon the people who perform them do not necessarily produce entities which are distinguished by how people perform the activities. As a counterexample, consider, again, the activity of house construction. As shown above, there are elements to this activity which depend upon the person performing it. However, this aspect of the activity does not necessitate that the distinction among all houses is dependent on those who build the houses. A wigwam is a very different dwelling than a castle although both result from the activity of house construction (in the broad sense). The distinction between them is not based on the practical activities involved in their construction. Each dwelling has properties which are not shared by the other. Similarly, although the activity of computer programming (taken in Moor's broad sense) may be involved in creating hardware and software, this may not necessitate that the distinction between the two is based on the practical aspects of the activities which create them. There may be other properties germane to hardware and software which differentiate them.

Next, let us consider the symbolic level. Recall, Moore asserts that at the symbolic of computer programs what counts as a computer instruction is also dependent upon the practical activity of computer programming. Thus, even when understood at the symbolic level, the distinction between software and hardware is dependent upon the practical activity of computer programming. My counterexample of house building also applies to the symbolic level as well. For, houses also have a symbolic level of understanding in the form of blueprints. Blueprints, like computer programs, consist of a set of instructions. That is, a set of instructions for building a house. These instructions, of course, have different interpretations for each person involved in the construction of a house. For example, an electrician is concerned with how to install an electrical system, while a brick mason is concerned with building a foundation. However, this does not mean that the distinction between electrical systems and foundations is based upon the practical activities of the electrician and brick mason. Similarly, although computer instructions may be interpreted differently in contexts pertaining to software and hardware, this does not necessitate that the distinction between software and hardware is based upon how those who carry out the instructions interpret the instructions.

There are a number of responses to my counterexample. First, it may be argued that I have been too broad in my examples of house construction. However, I reply that the same criticism can be applied to Moor’s examples of computer programming. He, after all, includes both the activities of replacing circuits and typing lines of code under the umbrella of computer programming. Thus, Moor’s description of computer programming suffers from the same deficiency as my example of house construction. In order to enquire further, a more detailed description of computer programming is needed. However, Moor does not provide one.

Second, one can just respond that my counterexample is just plain wrong. By comparing house construction to computer programming, I am comparing apples to oranges. I agree that most if not all analogies break down at certain point. However, Moor has provided a very general description of an activity and the relation of the activity to its instances. Likewise, I have provided a very general description of the activity of house construction. If my analogy breaks down, it does so in the details. But, again, Moor has not provided adequate details in order to determine where my analogy breaks down.

There are, of course, other responses, but it is not the purpose of this essay to address them all. Rather, I suggest that the crucial mistake in Moor’s position is that his ontological perspective is too narrow. That is, by considering only the symbolic and physical aspects of computer programs, he is not able to find any significant ontological differences between software and hardware. This, for me, is an implausible result. In my counterexample I have shown that when Moor’s implicit ontology is applied to other aspects of reality it leads to difficulties. What is needed, then, is a general ontological framework which describe software and hardware as well other aspects of reality.

3. Suber's argument: Hardware is software.

In Peter Suber's article "What is Software", one conclusion he reaches is that hardware is software. Suber's argument is more detailed than Moor's, and, thus, for the sake of clarity, I will present his argument by means of the following premises (P1 – P4) and conclusions (C1 – C3):[9]

(P1) Software is pattern that is readable and executable.
(P2) All patterns meet the physical and grammatical requirements for readability and executability.
(P3) All concrete objects display patterns.
(C1) Therefore, all concrete objects can be read and executed as software.
(C2) Therefore, all concrete objects are software.
(P4) Hardware is a concrete object.
(C3) Therefore, hardware is software.

Suber's first premise (P1) is based on his assertion that software at its most basic level must be represented as some type of pattern. In this assertion, it is important to understand that Suber is using pattern "in a broad sense to signify any definite structure, [and] not in the narrow sense that requires some recurrence, regularity, or symmetry."[10] Thus, whether in the form of magnetic oxide on a disk, pits and lands on CD, or some other form, it is the pattern which represents the instructions contained in software.

This characterization of software, however, is not complete. For, it does not adequately distinguish software from data or noise. Software gives instructions to a machine, but the requirement that software is pattern does not offer any criteria determining what types of patterns can do this. Thus, Suber adds the requirement that software "must in principle be capable of meeting the physical and grammatical conditions of readability and the requirement of

[9] Peter Suber, "What is Software", http://www.earlham.edu/~peters/writing/software.htm (accessed November 23, 2009).
[10] Peter Suber, "What is Software".

executability."[11] By, "physical and grammatical conditions", here, Suber means first that the pattern must be in some physical form that a machine can read, and second that the pattern must have "certain syntactic structures within the pattern" [12] that can act as instructions to the machine.

The second premise (P2) follows from two principles Suber calls the Sensible and Digital Principles. The Sensible Principle states that "any pattern can in principle be physically embodied."[13] That is, as Suber states, "[all] patterns that can be imagined can be drawn. Patterns that are conceivable but not imaginable (like Descartes' chiliagon or 1000-sided polygon) can be described in a notation that provides a complete recipe for conception; and the notation can be drawn."[14] Thus, since all patterns can be physically embodied, all patterns meet the physical requirement for software.

The Digital Principle states that any pattern in principle can be represented as a digital pattern.[15] For example, any analog pattern (e.g., a painting) can be digitized. Once digitized, though, the grammatical requirement for software is met. For, within a digital pattern, the necessary distinctions are present for constructing syntactic structures. This then makes it possible the digital pattern to be read and executed. That is, the syntactic structures enable the pattern to give instructions to machine. Thus, since all patterns can be digitized, all patterns meet the requirements for readability and executability. From the Sensible and Digital Principles, then, it follows that all patterns meet the physical and grammatical requirements for readability and executability.

[11] Peter Suber, "What is Software".
[12] Peter Suber, "What is Software".
[13] Peter Suber, "What is Software".
[14] Peter Suber, "What is Software".
[15] Peter Suber, "What is Software".

The third premise (P3) is rather uncontroversial since Suber uses the term "pattern" in a broad sense (i.e., anything with a definite structure). From this, then, it follows that all concrete objects can be read and executed as software (C1). For, from the Sensible and Digital Principles, it follows that the patterns displayed by concrete objects meet the physical and grammatical requirements for readability and executability. Thus, since all concrete objects meet these requirements, they can be read and executed as software.

This, however, does not get us to the conclusion that all concrete objects are software (C2), only that all concrete objects can be software. In order to infer (C2) it must be noticed that regardless of whether a pattern is taken to have intrinsic meaning or not, it is possible to give a pattern meaning. This meaning may take the form of giving a pattern a meaningful reading, or it may take the form of constructing a formalism which expresses a certain meaning. The former, Suber calls the "freedom of interpretation", and the latter he calls the "freedom of formalization".[16] As an example, consider Suber's remarks:[17]

> The freedom of interpretation enables us to read the pattern of commas in the first edition of the *Critique of Pure Reason* as a definite structure holding an indefinitely various body of information, depending on what language conventions we will use to interpret it. The freedom of formalization allows us to construct the pattern that, when physically expressed as hardware, will execute the patterns we have chosen as software.

Thus, from the freedom of interpretation and freedom of formalization, it follows that not only can concrete objects be read and executed as software, but also that they are in fact software. For, all concrete objects display patterns, and, moreover, these patterns are readable and executable. But, this conclusion reaffirms (P1). That is, software is pattern that is readable and executable. Hence, all concrete objects are software. Given this conclusion (C2), the last

[16] Peter Suber, "What is Software".
[17] Peter Suber, "What is Software".

premise (P4) and final conclusion (C3) are somewhat trivial. As Suber states, “[hardware], in short, is also software, but only because everything is.”[18]

3.1 Discussion of Suber’s argument

For ease of reference, here is Suber’s argument presented in the last section:

(P1) Software is pattern that is readable and executable.
(P2) All patterns meet the physical and grammatical requirements for readability and executability.
(P3) All concrete objects display patterns.
(C1) Therefore, all concrete objects can be read and executed as software.
(C2) Therefore, all concrete objects are software.
(P4) Hardware is a concrete object.
(C3) Therefore, hardware is software.

One reasonable inquiry to make at this point is whether Suber’s argument is valid. However, I am not going to challenge the validity of his argument. Much of Suber’s article is devoted to spelling out his reasoning, and in order to question the validity of Suber’s argument much effort would have to be devoted to spelling out Suber’s reasoning in more detail than I already have. This would be a time consuming, and I think ultimately unnecessary project. Thus, I will simply grant that Suber’s argument is valid.

We are now left the question of the plausibility argument’s premises. There are a number of questions one could raise, but I will forego doing this. Rather, I will consider whether Suber’s argument leads to an implausible conclusion. If it does, then, assuming the argument is valid, there must be problem with one of the premises. For this purpose, consider the following premise and conclusion:

(P5) A peanut butter sandwich is a concrete object.
(C4) Therefore, a peanut butter sandwich is software.

[18] Peter Suber, “What is Software”.

Given that Suber's argument asserts that everything is software, (C4) is a valid inference. But, (C4) strikes me as implausible. For, aside from being somewhat humorous, it ignores an important distinction between peanut butter sandwiches and software. Namely, peanut butter sandwiches are things we eat. This is not to deny that in the future we may have edible software. But, if this were to happen, would we start considering all food to be software? Perhaps. Perhaps not. To engage in serious inquiry about this question would be too far off task. It would require a detailed investigation into types of things we consider to be food.

So, how might one respond to this? First, of course, one could charge that I have not adequately represented Suber's argument. To this I respond that even if I have misrepresented some of the finer points of the argument, Suber's conclusion that "everything determinate is software"[19] is clear. Thus, (C5) is still a valid inference. That is, from the assertion that everything is software, it still follows that a peanut butter sandwich is software.

Second, one could hold that although (C5) sounds implausible, the argument is still, in fact, sound. This may be the case. For, I imagine that food in some sense can be considered as software for the body. But, if one really wishes to hold to the view that everything is software, the question must be raised as to how to distinguish the various types of software in the world. For, Suber's argument not only asserts that peanut butter sandwiches are software, but so are automobiles, shopping malls, and roller coasters. I find this implausible, but not because Suber's argument is invalid. Rather, I find it implausible because Suber's assertion that "everything is software" does not account for other distinctions we make. For, by focusing solely on the nature of pattern and how it defines software, Suber, like Moor, presents too narrow of an ontological perspective to make these distinctions.

[19] Suber, "What is Software".

This lack of ontological perspective is not necessarily damning for Suber's argument. He may still be correct. However, it does present us with an option. We can hold to Suber's conclusion that everything is software, or we can consider another ontology which will allow us to make distinctions that are more plausible. I will now turn to this latter option.

4. What is an ontology?

In order to demonstrate that there is an ontological distinction to be made between software and hardware, it is necessary to say a little about what ontology is. Historically, ontology is a branch of philosophy concerned with the nature of what exists. To be more precise, this means the "science of what is, of the kinds and structures of objects, properties, events, processes and relations in every area of reality."[20] When creating an ontology (singular), then, one seeks to provide a detailed account these objects, properties, events, processes, and relations. That is, an account of what distinguishes one entity from another, and how distinct entities relate to each other.

There are different kinds of ontologies. The type of ontology I will be concerned with is "realist ontology". In realist ontology, the terms within an ontology should represent reality itself. This is juxtaposed with what I shall call "conceptualist ontology". In conceptualist ontology the terms in the ontology represent concepts. What is meant by the term "concept" varies depending on the ontology. For, they can be mental representations, abstract ideas, or some other product of human cognition.[21]

[20] Barry Smith, "Ontology", *The Blackwell Guide to the Philosophy of Computing and Information*, (Victoria, Australia: Blackwell Publishing, Ltd., 2004), 155.
[21] Barry Smith, "Beyond Concepts: Ontology as Reality Representation", http://ontology.buffalo.edu/bfo/BeyondConcepts.pdf, 3-4 (accessed November 29, 2009).

The difference between realist and conceptualist ontologies can be seen more clearly when we consider how each are evaluated. In realist ontology the criterion for evaluation is how well the ontology portrays reality. This mandates that realist ontology is informed by science or some other form of empirical investigation. For example, an ontology of chemical processes draws upon the expertise of chemists and the body of knowledge associated with the field of Chemistry. It is important to note that realist ontologies are not immutable. Rather, as new discoveries are made, the ontology must be updated to reflect the new advances.

Conceptualist ontologies, on the other hand, are not necessarily evaluated by how well they portray reality. Rather, the creation of a conceptualist ontology is a pragmatic enterprise in which the objects of the domain are "conceived as nothing more than nodes or elements of data models devised with specific practical purposes in mind"[22], and not an enterprise necessarily concerned with accurately describing reality (in the mind-independent sense). This emphasis on practicality can have unintended consequences. For example, an ontology of alchemy can be deemed just as good (or perhaps even better) than an ontology of chemistry.

5. Which ontology to use?

Realist ontology is best suited for investigating hardware and software. For, these entities are the result of computational investigation and technological advances. We are interested in the entities as they exist in reality, and not the concepts of hardware and software. More specifically, the realist ontology I shall use is the Basic Formal Ontology (BFO).

My reasons for choosing BFO are straight forward. First, it has a large number of users.[23] Thus, BFO is constantly being tested, and deficiencies addressed. Second, much

[22] Smith, "Ontology", 162.
[23] http://www.ifomis.org/bfo/users

thought has been given into its categories and relations. This removes the burden of having to "reinvent the wheel" (so to speak) as to which categories and relations need to be fundamental. For the interested reader, there is a wealth of literature freely available on the internet which justifies BFO's classifications.[24] Finally, BFO is domain neutral. That is, "it is narrowly focused on the task of providing a genuine upper ontology which can be used in support of domain ontologies developed for scientific research … [The BFO] does not contain physical, chemical, biological or other terms which would properly fall within the special sciences domains."[25] Thus, I am not committed to terms or categories which may only be applicable in other specific domains of inquiry.

5.1 A brief introduction to BFO

BFO, as mentioned, is a realist ontology. This commitment to realism is brought out by how BFO relates universals and particulars. Universals (often called natural kinds) are the entities "in reality to which the general terms used in making scientific assertions correspond."[26] For example, the general term "human being" signifies a universal because there are particular instances of human beings. The entities signified by universal terms, thus, have a spatiotemporal existence in that they signify what their corresponding particular instances have in common.[27]

This assertion that universals have spatiotemporal existence may strike you as odd. If so, I offer two quick replies. First, it is important to remember that universals and their particular instance have a symbiotic relationship: one cannot exist without the other.[28] This is important

[24] For examples, see "Barry Smith, National Center for Ontological Research" (http://ontology.buffalo.edu/smith/), and "Basic Formal Ontology (BFO)" (http://www.ifomis.org/bfo).
[25] "Basic Formal Ontology (BFO)", http://www.ifomis.org/bfo (accessed September 17, 2009).
[26] Smith, "Beyond Concepts: Ontology as Reality Representation", 6 (accessed July 17, 2009).
[27] Smith, "Beyond Concepts: Ontology as Reality Representation", 6 (accessed September 22, 2009).
[28] Smith, "Beyond Concepts: Ontology as Reality Representation", 6 (accessed September 22, 2009).

for avoiding the mistake of confusing universals with concepts. Universals must have particular instances in objective reality, concepts do not.

Second, "[it] is the existence universals which allows us to describe multiple particulars using one and the same general term, and, thus, makes science possible."[29] That is, science is concerned with understanding the principles which can be applied to the plurality of particular entities existing at different places and times.[30] Such an understanding could not be possible if the general features describing the particulars did not exist.

Since BFO is an upper level ontology, its purpose is to describe reality in general. To do this, BFO's most general term is "entity". Simply put, "[everything] which exists in the spatio-temporal world is an entity."[31] This high degree of generality allows the universal term "entity" to apply to the entities of any scientific domain.[32] The generality entailed by the universal term "entity", however, is not sufficient for the building an adequate ontology. It would be an ontology consisting of one category with everything in it, and no distinguishing criteria. The first important criterion for categorizing entities is the continuant/occurrent distinction. BFO defines these universal terms as follows:[33]

1. **continuant**: An entity that exists in full at any time in which it exists at all, persists through time while maintaining its identity and has no temporal parts.
   Examples: a heart, a person, the color of a tomato, the mass of a cloud

2. **occurrent**: An entity that has temporal parts and that happens, unfolds or develops through time. Sometimes also called perdurants.
   Examples: the life of an organism, the most interesting part of Van Gogh's life, the spatiotemporal region occupied by the development of a cancer tumor

[29] Barry Smith, "Chapter 4: New Desiderata for Biomedical Terminologies", in *Applied Ontology: An Introduction*, ed. Katherine Munn and Barry Smith (Piscataway, NJ: Transaction Books, Rutgers University, 2008), 92.
[30] Barry Smith, "Chapter 4: New Desiderata for Biomedical Terminologies", 92.
[31] Grenon and Smith, 75.
[32] Spear, 28.
[33] "jOWL - Basic Formal Ontology Browser", http://jowl.ontologyonline.org/bfo.html (accessed September 19, 2009).

The continuant and occurrent categories contain a number of important subcategories. However, I will only be dealing with the continuant branch of BFO. For, hardware and software are entities which maintain their identity through time. The occurrent branch is important, but it is beyond the scope of this paper to deal with all aspects of BFO. A complete description of BFO is found at the web site www.ifomis.org/bfo.

6. Continuants

When considering the terms "hardware" and "software", we are confronted with problems. First, consider the term "hardware". Does it refer to some specific piece of hardware or an aggregate of components which form a computing system? Thus, rather than using the term "hardware", I will use the general term "piece of computing hardware", to refer to physical entities contained in a computing system that are recognized as being a unified whole.

Next, consider the term "software". Again, this term is ambiguous. Sometimes we use the term "software" to refer to specific physical objects, such as CDs or floppy disks, that we can load onto multiple computer systems. Other times we use the term "software" to refer to the computer programs that are contained on these physical objects. Thus, to be clear, I will use the term "software application" to refer to software in the first sense of the term. That is, an entity that has been designed to be loaded onto multiple computer systems.

6.1 Piece of computing hardware

With the ambiguities of the term "hardware" addressed, what does it mean for an entity to be a piece of computing hardware? To answer this, I must bring in a distinction among continuants: the distinction between independent and dependent continuants:[34]

1. **independent continuant**: A continuant that is a bearer of quality and realizable entity entities, in which other entities inhere and which itself cannot inhere in anything.

2. **dependent continuant**: A continuant that is either dependent on one or other independent continuant bearers or inheres in or is borne by other entities.

As a simple example of how to distinguish between independent and dependent continuants, consider your hair. The hair, itself, is an instance of the independent continuant, but the color of your hair is an instance of a dependent continuant. For, in order for the color of your hair to exist, it has to inhere in some object (i.e., your hair). Again, it is important to note the symbiotic nature between the instances of independent and dependent continuants. That is, all instances of independent continuants have instances of dependent continuants that inhere in them, and all instances of dependent continuants inhere in some in some instance of an independent continuant.[35]

Given the independent/dependent continuant distinction, a piece of computing hardware, then, is an independent continuant. For, a piece of computing hardware bears certain qualities and realizes certain functions that are necessary for computation. For example, a hard drive is designed to store information in a magnetic medium, and some set of logic gates on a CPU is designed to add integers. In each example, there is some computational function (i.e., the storing

[34] "jOWL – Basic Formal Ontology Browser", http://jowl.ontologyonline.org/bfo.html (accessed September 25, 2009).
[35] Barry Smith, "A Formal Theory of Substances, Qualities, and Universals", http://ontology.buffalo.edu/bfo/SQU.pdf (accessed September 25, 2009).

of information and the adding of integers) that is involved in computation. By computational function, here, I am referring to some complex set of phenomena (or some set of end directed activities) that are involved in computation, and not to some mathematically rigorous definition. This may not satisfy some readers, and, while a more formal description of computational function is an important undertaking, it is more detail than in necessary at this point. However, since it is playing a crucial part in defining computing hardware, it is still important to investigate the ontological status of a computational function in order to gain some understanding as to why it plays this critical role.

6.2 Computational function

Despite the informal character of my use of the term "computational function", there is still quite a bit more which can be said about its ontological status. Among dependent continuants, BFO distinguishes between generically dependent continuants and specifically dependent continuants:[36]

1. **generically dependent continuant**: A continuant that is dependent on one or other independent continuant bearers. For every instance of A requires some instance of (an independent continuant type) B but which instance of B serves can change from time to time.

2. **specifically dependent continuant**: A continuant that inheres in or is borne by other entities. Every instance of A requires some specific instance of B which must always be the same.

As a simple example, consider a PDF file. It is an instance of a generically dependent continuant because the PDF file can exist on multiple computers at multiple times. However, a quality like

[36] "jOWL - Basic Formal Ontology Browser", http://jowl.ontologyonline.org/bfo.html (accessed September 23, 2009).

the redness of a particular tomato is an instance of a specifically dependent continuant because if that particular tomato did not exist, its redness would not exist.[37]

Applying the generic/specific distinction to my notion of computational function, it is tempting to classify computational function as a generically dependent continuant. For example, adding integers is a computational function, and instances of this computational function are found on multiple computers. Thus, instances of computational functions are instances of generically dependent continuants.

This, however, commits the following mistake: it fails to consider the nature of an instance of computational function. An instance of a computational function is actualized (or manifested) in a computational process. For example, consider this simple example of program code (written in Java) the execution of which adds two integers (i.e., int i, int j):

```
public int addTwoIntegers(int i, int j) {
        return i + j;
}
```

This fragment of Java text is an instance of a generically dependent continuant. It can be copied, and exist in multiple places. But, the function that it describes (i.e., the function of adding of two integers) is not the same as the program text. For, the function could have been written in one of many programming languages. It is this entity (i.e., the computational function of adding two integers) which is specifically dependent. For, any instance of this function is actualized in the running of some particular computational process, and the particular computational process could not exist without the instance of a piece of computing hardware on which the process runs.

Given this description of a computational function as being specifically dependent, a point of clarification is needed. Since I described the instances of computational functions as

[37] Examples are taken from: "jOWL – Basic Formal Ontology Browser", http://jowl.ontologyonline.org/bfo.html (accessed September 25, 2009).

being actualized in processes, would not this make computational functions occurrents and not continuants? Again, this would be a mistake. It confuses the natures of the entities being described. The computational function is an entity, the actualization of which (speaking loosely) "does" something. The "doing" is, of course, an occurrent. It unfolds in time. However, the function, itself, is not a temporal entity. It exists even when the function is not being actualized.

This distinction between the function and the process in which it is manifested is more clearly seen in a non-computational example. Consider a claw hammer. One function of the hammer is to drive nails (pulling nails in another). This function does not cease to exist when the hammer is lying on a work bench. Rather, the function to drive nails is something the hammer always posses and this function is actualized when the hammer is involved in the process of driving nails. This relation between the function of the hammer to drive nails and the actualization of driving nails is analogous to the function of adding numbers described above. In each case, the function is non-temporal, but it is actualized in a temporal process.

Entities that are actualized in a temporal process are classified in BFO as realizable entities, and, in the case of computation functions, more specifically as functions (i.e., a function is a subclass of a realizable entity):[38]

1. **realizable entity**: A specifically dependent continuant that inheres in continuant entities and are not exhibited in full at every time in which it inheres in an entity or group of entities. The exhibition or actualization of a realizable entity is a particular manifestation, functioning or process that occurs under certain circumstances.

2. **function**: A realizable entity the manifestation of which is an essentially end-directed activity of a continuant entity in virtue of that continuant entity being a specific kind of entity in the kind or kinds of contexts that it is made for.

[38] "jOWL – Basic Formal Ontology Browser", http://jowl.ontologyonline.org/bfo.html (accessed September 25, 2009).

Based on these definitions, then, a computational function is a realizable entity since it fits the necessary criteria for being a realizable. It is actualized in processes that occur in one or more pieces of computing hardware, and, the computational function may not be actualized at all times (e.g., the computer may be turned off). Furthermore, a computational function is a BFO function, since the computational function (i.e., the entity) that inheres in a piece of computing hardware does so because the piece of computing hardware has been intentionally designed for that purpose.

6.3 Software application

Let us now consider how to classify a software application. A software application contains a set of encoded instructions that represents some computational function. There are two important terms, here, of note. First, is the use of the term "encode". Software exists in many forms: on CD(s), on floppy disk(s), on punch cards, etc. In all of these, the instructions are encoded in distinct ways. Second, is the use of the term "representation". The encoded set of instructions is not the computational function. For example, in my "addTwoIntegers" example above, the encoded instructions are not the same as computational function, but a representation of the computational function. I could have represented the computational function is some other programming language.

Assuming this analysis is correct, two results follow. The first is that an instance of a software application is an instance of a generically dependent continuant. For, an instance of an encoded representation of a computational function does not depend on a specific instance of an independent continuant in order to exist; only that it exist in at least one instance of an independent continuant. The second is that the encoded computational function is not actualized in the software application. For example, my CD of my Microsoft Word does not execute Microsoft Word. The execution of Microsoft Word only occurs when my CD is loaded onto my laptop (i.e., a piece of computing hardware).

6.4 Definitions: computing hardware, computational function, and software application

Based on the ontological distinctions above, it is now possible to define the entities "piece of computing hardware", "computational function", and "software application":

1. **piece of computing hardware**: An independent continuant that is intentionally designed for the actualization of one or more computational functions.

2. **computational function**: A realizable entity that inheres in one or more pieces of computing hardware. The actualization or manifestation of a computational function is an essentially end-directed activity in virtue kind or kinds of contexts the computing hardware is made for.

3. **software application**: A generically dependent continuant that encodes a representation of one or more computational functions (realizable functions).

In each these definitions, it is important to note the crucial role that the BFO category "function" is playing. For, being a function (in the BFO sense) entails that computational functions are intentionally designed entities. Thus, not just any function will count as a distinguishing quality for hardware (e.g., the function of my hard drive to be a paper weight), and not just any randomly encoded representation will count as a software application.

7. Conclusion

This paper has yielded the following results. First, using the BFO, the entities "piece of computing hardware", "software application", and "computational function" are distinguished in BFO as follows:

entity

- continuant
    - independent continuant
        - **piece of computing hardware**
    - dependent continuant
        - generically dependent continuant
            - **software application**
        - specifically dependent continuant
            - quality
            - realizable entity
                - function
                    - **computational function**

Thus, in contrast to the positions of Moor and Suber, a piece of computing hardware and a software application are very different entities.

Second, in regards to this ontological categorization, these entities have been given the following definitions:

1. **piece of computing hardware**: An independent continuant that is intentionally designed for the actualization of one or more computational functions.

2. **computational function**: A realizable entity that inheres in one or more pieces of computing hardware. The actualization or manifestation of a computational function is an essentially end-directed activity in virtue kind or kinds of contexts the computing hardware is made for.

3. **software application**: A generically dependent continuant that encodes a representation of one or more computational functions (realizable functions).

The advantage of these definitions is that they clearly distinguish among the three entities, and, thus, helps avoid confusion about their ontological status. The net result, of course, is that hardware is distinguished from software in an ontologically significant way.

8. Acknowledgements

I would to thank professors Barry Smith, Raymond Turner, and Amnon Eden. Without their insightful comments, I could not have completed this paper.

References

1. "Basic Formal Ontology (BFO)", http://www.ifomis.org/bfo (accessed September 17, 2009).

2. Moor, James H. "Three Myths of Computer Science". *British Journal for the Philosophy of Science* 29(3) (September 1978): 213 - 222.

3. "jOWL - Basic Formal Ontology Browser." http://jowl.ontologyonline.org/bfo.html (accessed September 19, 2009).

4. Smith, Barry. "A Formal Theory of Substances, Qualities, and Universals." http://ontology.buffalo.edu/bfo/SQU.pdf (accessed September 25, 2009).

5. Smith, Barry. "Beyond Concepts: Ontology as Reality Representation", http://ontology.buffalo.edu/bfo/BeyondConcepts.pdf (accessed July 17, 2009).

6. Smith, Barry. "Chapter 4: New Desiderata for Biomedical Terminologies", in *Applied Ontology: An Introduction*, ed. Katherine Munn and Barry Smith. Piscataway, NJ: Transaction Books, Rutgers University, 2008.

7. Smith, Barry. "Ontology". *The Blackwell Guide to the Philosophy of Computing and Information*. Victoria, Australia: Blackwell Publishing, Ltd., 2004, 155-166.