Forthcoming in *Grazer Philosophische Studien.*

SOFTWARE IS AN ABSTRACT ARTIFACT[1]

Nurbay IRMAK
University of Miami

Summary

Software is a ubiquitous artifact, yet not much has been done to understand its ontological nature. There are a few accounts offered so far about the nature of software. I argue that none of those accounts give a plausible picture of the nature of software. I draw attention to the striking similarities between software and musical works. These similarities motivate to look more closely on the discussions regarding the nature of the musical works. With the lessons drawn from the ontology of musical works I offer a novel account of the nature of software. In this account, software is an abstract artifact. I elaborate the conditions under which software comes into existence; how it persists; how and on which entities its existence depends.

Keywords: Software, computer programs, musical works, ontology, artifacts, abstract objects

Software has become one of the most important parts of our everyday life. For most of us our personal computers are indispensable. For many people it is hard to imagine how their lives would be without them. We use e-mail for communication, pay our taxes using tax software, pay our bills online, we even do the grocery shopping online. It is not only computers that work with software, phones, cars, fridges, cameras, etc., that is, every electronic device uses software. In this paper I shall discuss the nature of this ubiquitous artifact. I will make two assumptions. The first, controversial one, is that I will assume that there are such things as artifacts.[2] Second, I will

[1] I would like to thank Gordon Bearn, Patrick Chu, Ertürk Demirel, Zeynep Savaş and anonymous referees at *Grazer Philosophische Studien* for their valuable comments on an earlier draft of this paper. I am especially grateful to Amie Thomasson for her critical remarks and helpful suggestions.

[2] So I am ignoring here all the arguments against the existence of artifacts provided by philosophers like Peter van Inwagen (1990) and Trenton Merricks (2001).

assume that an object is an artifact if and only if it is an intentional product of human activity.[3] Given the second assumption, it follows that software is an artifact.[4]

In this paper I offer a novel account of the nature of software. I will argue that although it is an artifact, software cannot be identified with any concrete object, that is, any object having spatio-temporal location. I shall argue that software is a kind of abstract artifact which fails to fit the classical Platonic picture of abstract objects. The kind of abstract object defended here is not Platonic because it is not an eternal and mind-independent entity but is created by human beings with certain intentions. Things that are created (or are/can be destroyed), whether concrete or abstract, have temporal properties; that is, they begin (or cease) to exist at a certain time. I take the category of abstract objects to cover all non-spatial entities which may or may not have some temporal properties. Platonic entities, in this view, fall under this ontological category as they lack spatio-temporal properties.

The paper has three parts. In (I) I will very briefly argue that most of the proposals offered in the philosophy of computer science fail to capture the nature of software as an artifact. It is important to distinguish, or so I will argue, different concepts that are closely related to software such as algorithm, text, copy and execution of software, since (a) people have tried to identify software with one or the other and (b) the differences between them and software and the differences among them may help us to understand the nature of software. I argue that all of them are distinct objects and none of them can be identified with software. In (II) I will draw attention to certain similarities between software and musical works. Granting the similarities, I shall argue that the discussions regarding the nature of the musical work can and do help us to understand what software is. I will briefly discuss one of the most plausible views about musical works and see the consequences for software. After addressing the main problems with it I shall argue for an improved approach to musical works and software. In (III) I lay out my positive account of software. I explain in what conditions software comes into existence; how it persists; how and on which entities its existence depends.

Before I get to the ontology of software, I need to say a bit about the methodology I follow in this paper. The goal of the present study is to develop an ontology of software which is in accordance with the common beliefs and practices of computer programmers, software users, or any competent speaker who uses the related concepts. Throughout this paper, I will argue against the kinds of ontology that require serious revisions in our common beliefs and our language regarding the objects in question. I suggest that an adequate and successful ontology of musical works and software is a theory that takes the common beliefs and practices of listeners, composers, programmers, computer users or any competent speaker who use the related concepts seriously. "Seriously" is to be understood in its strongest sense. Our theory of software or musical works, if we accept that there are such things, must be coherent with the way people talk about them, with the things they believe about them, with their practices that involve those

[3] This is a widely accepted definition of artifacts. For further discussion on what an artifact is see Hilpinen (2004).

[4] This is not free from controversy, of course. According to some people software is not necessarily a product of human activity. See Suber (1998). More on this will follow.

objects. The desiderata for such theory are to be based on those beliefs and practices. The ontology of musical works or the ontology of software we need is not a kind of ontology in which one searches for the facts about musical works and software which are independent of human practices and beliefs. Ordinary beliefs and common practices guide our understanding of the existence and persistence conditions of the objects in question. However, those beliefs and practices are sometimes not as clear or as precise as we would like them to be, and the ontology that underlies them may be implicit; making it explicit requires philosophical work such as is here undertaken. I will not defend this methodological approach here, though others have done so elsewhere.[5]

(I) Ontology of software

Many philosophers and computer scientists share the intuition that software has a dual nature (Moor 1978, Colburn 2000). It appears that software is *both* an algorithm, a set of instructions, *and* a concrete object or a physical causal process. On the symbolical level software is some sort of abstract mathematical object that can be implemented in a machine. Many take an algorithm to be a kind of recipe, that is, a finite sequence of instructions intended to achieve a goal. I will follow that view and use “algorithm” in that sense. On the physical level software involves a realization of the algorithm in the hardware; in the central processor, memory elements, input/output devices, etc. The idea is that it would be a mistake to reduce software to an entity that belongs exclusively to one or the other levels described above. Timothy Colburn argues that the duality can be explained by distinguishing the medium of description from the medium of execution. The medium of description is a text that is constructed by one of the many possible levels of formal language, whereas the medium of execution consists of concrete computer parts like circuits, semiconductors, etc. Though helpful, this doesn’t address the question of what the nature of software is. Trying to save the above intuition, Colburn claims that software is a concrete abstraction. It is something that is “at once concrete and abstract (Colburn 2000, 205).” It is far from clear what a concrete abstraction means, however, and Colburn doesn’t attempt to explain it (Colburn 2000, 205, 208-209).

The dual nature view is quite confusing. The idea of concrete abstraction needs serious philosophical work and in the absence of such work it doesn’t seem to be an acceptable characterization. However it would be unfair to dismiss the dual nature view for that reason. Perhaps what it means for software to have a dual nature is just that it has concreteness and abstractness as its essential properties. But if concreteness is being in space and time and abstractness is being non-spatio-temporal, then having both as the essential properties is amount to saying that software has contradictory essential properties. In any case, in due course I will provide arguments against the idea that software can be identified with a concrete or spatio-temporal object and thus it can’t have concreteness as one side of this ‘dual nature’, so I will postpone this issue until then.

I want to discuss briefly another view which is a bit extreme. It is extreme simply because in this view everything turns out to be software. Peter Suber (1998) argues that software is nothing but a pattern *per se.* What he means by pattern is important for his account. “A

[5] See Thomasson (2004), (2005).

pattern," he writes, "is taken in a broad sense to signify any definite structure, not in the narrow sense that requires some recurrence, regularity, or symmetry" (Suber 1998, 90). In this view, anything that has a pattern qualifies to be software. All the questions and concerns about whether a computer program exists only when it is being executed or if it is also present when it is being stored, copied, erased or advertised (Turner & Eden 2008), as well as the dispute over the distinction between hardware and software is rendered trivial (Moor 1978, Colburn 2000). "Hardware is also software," Suber concludes, "but only because everything is" (Suber 1998, 102).

But it is one thing to *have* a pattern and another thing to *be* a pattern. It appears that Suber doesn't make such a distinction. So he is unable to distinguish the truth that clouds have patterns from the falsehood that they *are* patterns. Even if we ignore that distinction, Suber's account fails for another serious reason. His account of software is too broad and thus it misses the artifactual nature of software. We still need a description of how clouds, rocks and trees are different from Microsoft Word or MacOs X or Windows 7. If he is right then what follows is that Microsoft Word, a skin of a coral snake, the flag of Angola, behaviors of the Zodiac Killer are all of the same kind; namely they are all software just because they are or have a certain pattern. It seems Suber takes "software" to be a very broad category which applies to anything that has a definite structure. But what we need to explain is the nature of software and what makes it different from other kinds of things like human beings or mountains. Software as pattern *per se* fails to explain all the distinctions we make among those different kinds of things.

In fact, we demand or should demand of a theory of software that accounts for not only the differences between software and, say, tables and buildings, but also the differences between a particular piece of software (say Windows 7), its algorithm, its text or code, its copies and its executions. Let us put aside the question what software is for a moment and see if we can understand the rest of the list. The *algorithm*, as I have explained above, is a sequence of instructions to the computer in order to accomplish a certain goal. It is a kind of recipe for computer, a kind of to-do list. Many take it to be a kind of mathematical language-independent object. I don't have an argument for the claim that algorithms are eternal mathematical objects; I shall simply assume that they are. Therefore, (assuming we accept a Platonistic view of mathematical entities) an algorithm should be understood as an abstract object in the platonic sense. That is, it lacks spatio-temporal properties and unlike software, cannot be created. The fact that an algorithm is this abstract object devoid of spatio-temporal properties doesn't entail that it cannot have concrete particular instances. The type/token distinction might help us to understand the relation between an algorithm and its particular instance. The distinction is typically taken to be a distinction between a general kind of thing and its particular concrete instance. Type is abstract whereas token is often taken to be concrete or has spatio-temporal properties.[6] So the

[6] Nothing crucial in this paper hinges on what kind of things types are or whether or not all tokens are concrete. One common view is that types are universals. I will follow the crowd and assume that types are universals. Tokens are often taken to be particular concrete instances of types, but one might argue that there are abstract tokens. For instance, the token of a type 'string quartet' seems to lack spatial properties and thus an abstract object (I would like to thank anonymous referee for this example). See Wetzel (2008) for different theories about the nature of types.

distinction between an algorithm and its concrete instance, say, on a piece of paper is a distinction between a type and its token.

The *text* or *code* of software is a text written in a particular programming language. It is also abstract, but unlike algorithm, it is language-dependent. The type/token distinction can also be drawn for the text of software. A text as a type or a text type is abstract whereas its token copies are concrete objects. Consider very simple software that adds two values you enter and displays the sum. Here is a possible text of that software written in the programming language C++:

```
copy      a, reg1
add       b, reg1
copy      reg1,c.
```

A *copy* of software involves certain physical dispositions of particular components in a computer to do certain things. In this sense, having a copy of, say, Windows 7 in a computer is having a computer some of whose components (like hard drive) have certain dispositions to do certain things. Physical copies of a computer program like the one above on CD-ROMs, flash disks, etc. are tokens of the text of the program.

What hasn’t been explained yet is the *execution* of software (i.e. running a program), which is a kind of physical process, a kind of event. It is the physical manifestation of those dispositions in a computer.

I hope the differences between some of the things or stages I have listed and explained above (algorithm, text, copy and execution) are clear already. I do not think any of the things in that list can be identified with software. I will start with concrete things or stages and move on to abstract ones. Let me start with the question whether a piece of software is identical with an execution or set of such executions of a certain program. I think it is clear that they are not the same thing since they have different persistence conditions or if you like different modal properties. When I terminate the execution of Windows 7 by, say, turning the computer off, the execution of Windows 7 ceases to exist whereas nobody would accept the claim that by terminating an execution of certain software you destroy software itself. For similar reasons a particular copy of a piece of software is not identical with the software itself. Assume that I destroy a copy of Windows 7, say by breaking a particular Windows 7 DVD-ROM into pieces. In doing so, it is obvious that I destroy a copy of Windows 7, but I cannot destroy the program by merely destroying a particular copy of it. Identifying software with its particular copy is similar to a confusion caused by overlooking the type/token distinction. A text type is not identical to its text token. Even though they have an intimate relation, they still are distinct entities. Thus destroying a particular text token (a Windows 7 DVD-ROM) you do not thereby destroy the corresponding text type (the text of Windows 7). The existence of a type does not depend on its particular token. In other words, types survive the destruction of their particular tokens. In order to avoid repetition the discussion below on whether software is identical with its text or algorithm is merely concerned with algorithm and text types unless stated otherwise.

The text is not identical with software, because the same software might have different texts. You may write the same software with different programming languages and get different software texts. A good example would be software which is written with certain programming language that is compatible with a particular operating system (OS) and then translated into other languages so that other operating systems can run that very software. Think of a web browser, say Firefox 3, we use all the time. It can run both in Windows OSs and Mac OSs, but it has different texts for Windows and Mac.

The only possibility left on the list and probably the best candidate for being identical with software is the algorithm. I think the question whether software can be identified with an algorithm deserves a more detailed and careful answer. There are at least two reasons why a piece of software and the algorithm it employs are two different things. First, they are different because you can have two algorithms that are exactly the same but this doesn't guarantee that they are the algorithms of the same software. Second, you may have the same software containing different algorithms. I will start with the first one. One might argue that software doesn't have attributes over and above the properties of the algorithm that it contains and thus if you have the same algorithm then you will have the same software. Suppose two different programmers, *x* and *y* independently of each other come up with exactly the same algorithm at exactly the same time. Further, assume that *x* works for a big software company whereas *y* is just a computer science student. I think it would be reasonable to claim that the program that *x* wrote will be installed by millions of people, whereas the program that *y* wrote will have just one instance which is installed in his personal computer. *x*'s program will have many social, cultural and economical attributes that *y*'s program will lack. For instance the former will have properties like being expensive, user friendly, popular among young people, pretentious, cool etc, whereas the latter, not being a commodity, lacks all those properties. Or suppose that the same algorithm, say the algorithm that Windows 7 contains now, were used in the mid 1950's. Would it be the same software as the one we are using now? Not really. First of all there were no computers that could run the software back then. All the properties that we attribute to Windows 7 right now, like that it is more stable than Windows Vista or that it has a better user-interface than Windows XP couldn't be attributed to the algorithm that was indicated in the mid 1950's. If true, this difference between software and algorithm shows that unlike algorithm identity, software identity requires a historical continuity or sameness in origin.

The other reason that software is not algorithm is that the same software might have different algorithms. Consider Windows 7 again. Almost every day the software installs new updates, and its algorithm changes with every update. We normally say that it is the same software that survives all the changes. Furthermore, we want to be able to say that the same program could have a different algorithm than it actually has. In fact there are cases in which the same software written for different operating systems has different algorithms. For instance, Microsoft Word's algorithms for PC and for Macintosh are different and yet it seems we think that they are the same software.

But perhaps, the claim that software survives changes in its algorithm is too quick.[7] One might, for example, argue that the same software couldn't have different algorithms just as the same musical works couldn't contain different sound structures. So, according to this view, any change in the algorithm results in a numerically distinct piece of software. In fact this might not

[7] I thank an anonymous referee for pressing me on this point.

be as implausible as it seems. Consider the Microsoft Word example again. One could insist that the Microsoft Word for Mac and the Microsoft Word for PC are two different programs, and that is why they have different names. The latest Word for PC is called Microsoft Word 2010 and the one for Mac is called Word for Mac 2011. In order to understand and clarify this problem a distinction between a piece of software and its versions is to be made. This particular distinction has also important consequences for the proper understanding of the identity and persistence conditions of software. A version of a piece of software denotes one of its unique states. For instance, the version of the Microsoft Word processor that I use for typing this paper is called Microsoft Word 2011. There is no single method for software versioning. Different software companies, software engineers use different methods.[8] The decision to use a particular version numbering scheme is almost entirely a pragmatic decision; it is a matter of convenience (both for programmers and end-users). To illustrate the importance of software versioning for our discussion about the identity of software I will use “Semantic Versioning”[9], which provides a clear guideline for version numbering. Here are two important rules from that guideline:

- A normal version number must take the form X.Y.Z where X, Y, and Z are integers. X is the major version, Y is the minor version, and Z is the patch version. Each element must increase numerically. For instance: 1.9.0 < 1.10.0 < 1.11.0.
- Once a versioned package has been released, the contents of that version must not be modified. Any modifications must be released as a new version.

The order of numbering system is determined by the significance of the change made on the algorithm of a piece of software. Therefore, a change in the major version implies an important change in the computer program. Whereas, a change in the patch version mostly indicates fixing some bugs. Every change in software version, even a change in the patch version requires a change in the algorithm. This means that if software is identical with algorithm, then every version with a different number (including the minor and the patch number) is itself a different piece of software and not a different version of the same software. This conclusion is at odds with the common practice of software engineers, companies and end-users. Even if one might hesitate to claim that, say Firefox 2.2.2 is the same software as Firefox 6.0.2 due to the major changes in its algorithm, text and user-interface (the medium where users communicate with software or more generally the way software looks and feels to users) there is no doubt that Firefox 6.0.1 is the same software as Firefox 6.0.2; in the later release only two bugs are fixed.[10] Think of more complex and messy computer programs like operating systems available today. Take Windows 7, for instance. Almost every day the program installs updates altering its text and algorithm yet Windows 7 survives those changes. A quick look on the information about your PC will confirm this.

[8] For a list of some versioning methods see: http://en.wikipedia.org/wiki/Software_versioning (Retrieved September 2011).

[9] http://semver.org/ (Retrieved September 2011).

[10] See http://www.mozilla.org/en-US/firefox/6.0.2/releasenotes/ (Retrieved September 2011).

A different argument against the view that software is identical with its algorithm is motivated by the literature on the digital rights management. When one looks at the copyright infringement lawsuits for software, leaving the legal and moral issues aside it seems clear that computer programs are not taken to be identical to their algorithms by the courts. One interesting example is *Lotus vs. Paperback.* In June 1990, the federal district court of Boston, Massachusetts decided that the user-interface (or the look and the feel) of Paperback Software’s spreadsheet software VP-Planner infringed copyrights of the user-interface of the Lotus 1-2-3. The court held that the copyright protection extends beyond the literal source of the computer program, namely its algorithm and text.[11] It also covers to some extent the way software looks and feels. U.S courts gave similar decisions about extending the scope of software copyright so that it covers certain non-literal elements of computer programs.[12] One of the main issues in all those trials was to decide what counts as substantial similarity for the user-interface of computer programs. This is a hard question and fortunately we don’t need to settle the issue here. The importance of these court decisions for our purpose, namely deciding whether software is identical with its algorithm, is that they show that the identity of software is not only about its algorithm but also, for instance, about its interface; the way it looks and feels to end-users. The underlying idea here is that software survives changes in its algorithm to a certain extent so that we can trace copyright infringements. I conclude that identifying software with its algorithm contradicts with our practices and beliefs about software, therefore it should be denied.

To sum up what I have done so far, I first assumed that software is a kind of artifact and committed to a methodology of some sort of common sense ontology**.** Second, I have briefly argued that none of the accounts that have been given so far successfully captures the artifactual nature of software. Third, I have distinguished four different sorts of entity that are closely connected with software and argued that none of them can be identified with it. Now I want to conclude from the discussion so far that any successful ontology of software

(1) has to do justice to the artifactual nature of software (that it is created and is a production of human activity with appropriate intentions),
(2) must be able to distinguish software from its algorithm, its text, its copies and its execution,
(3) must account for the fact that software identity requires historical continuity or the sameness of origin, and
(4) must account for the identity of software over change in its algorithm or its text.

In the rest of the paper I shall argue that we can find traces for a successful theory of software in the ontology of music. I will briefly examine one of the most popular theories of musical works that is proposed by Jerrold Levinson (1980). I shall argue that although it suffers from serious problems it is a reasonable view and is on the right track. After applying the view to the subject matter of the present paper, I will argue that the view in question is problematic because it fails to satisfy (4) above; one of the desiderata for any successful ontology of

[11] See Lotus v. Paperback (1990).

[12] See, for instance, Computer Associates v. Altai (1992), Lotus v. Borland International (1996).

software. Finally, I will provide my own view in which software is a kind of historical abstract entity that is created by computer programmers with certain intentions.

(II) Software and musical works

The reason why I want to look closer to the ontology of music is that it seems software is very much like musical works in many respects. First of all both are historical abstract entities that are human creations. Second, we see a set of concepts that are closely related yet not identical to musical works as we have seen with software. In the case of software, there were closely related entities including the algorithm, text, copy and execution, whereas for musical works the concepts that are deeply connected and yet are difficult to identify with it are sound structure, score, copy of score and performance.

I will not discuss in details why musical works shouldn't be identified with any of the entities named above.[13] Instead I will make a few remarks that might motivate this view.[14] A musical work has many instances, none of which, it seems, can be identified with the work itself. The instances of a musical work are performances which are physical processes or events. A musical work can also not be identified with the original score written by its composer, since, as Jerrold Levinson argues (Levinson 1980, 5), many listeners may be familiar with a work who have no idea about or no contact with the original score. It seems that the musical work has a deep relation, if not identity, with its sound structure. A sound structure (like an algorithm) is a Platonic abstract object; "a structure, sequence, or pattern of sounds, pure and simple (Levinson 1980, 6)."

The classical view about the nature of musical works is a kind platonism according to which musical works are abstract objects; namely sound structures or types. In this view, the relationship between a musical work and its performance is a relation between a type and its token (Kivy 1983, 92; Deutsch 1991, 209; Dodd 2000, 424). Performances, then, are tokens of a musical work which is itself a structural type or a sound structure.

This classical view, however, meets serious objections. The decisive one, as Jerrold Levinson argues, is that the classical view fails to satisfy a criterion that any adequate theory of musical works should meet called creatability:

> Musical works must be such that they do *not* exist prior to the composer's compositional activity, but are *brought into* existence *by* that activity (Levinson 1980, 9).

The Platonic theory roughly sketched above fails, because according to standard Platonism abstract objects are eternal or atemporal[15] entities and thus cannot be created. Levinson gives two

[13] See Roman Ingarden (1989) for such discussion.

[14] For a detailed discussion on this question see Levinson (1980), Kivy (1983), Deutsch (1991), and Dodd (2000).

[15] Being eternal and being atemporal are two different things. x might be eternal but it is still in time, it just always was and always will be. On the other hand if x is atemporal it is not placed in temporal dimension and thus it would be a mistake to say that x always will be, since x has no temporal properties.

more requirements but for the purpose of this paper let's assume that the creatability requirement gives us enough grounds to reject the classical Platonism. What he proposes, instead, is a view in which musical works are still abstract objects, but unlike the classical Platonic view, he argues that those abstract entities can be created. Levinson argues that a musical work is a sound structure indicated by a composer at a certain time. According to him, a composer x by indicating a pre-existing sound structure ψ, at time t creates a new object called an 'indicated structure.' For instance, Beethoven's Symphony No. 7 in A Major is not to be identified with the sound structure Ψ, but instead, Levinson claims, it is *Ψ-as-indicated-by-Beethoven-in-1811-12.*

Let us look at how Levinson's theory could be applied to software. Let us assume that x is all the engineers who worked on coding the Windows 7 and Ψ is the algorithm x wrote for Windows 7. Then, Windows 7 is *Ψ-as-indicated-by-x-in-2009*. As an indicated type, Windows 7 was created in 2009 and thus it didn't exist, say, before the invention of the first computer. Software as an indicated algorithm satisfies the creatability requirement and so is in accordance with the assumptions I made before that software is a kind of artifact.

I find Levinson's view quite plausible. The application of his theory to software seems to satisfy most of the requirements that I set above; namely (1), (2) and (3). However, I think it fails to satisfy (4): the identity of software through change. Similarly I don't think that the theory can allow a change or a revision on a musical work once composed. If a musical work is a sound structure indicated by a composer at a certain time then changing the indicated sound structure (say by revising the score) would bring a distinct musical work into existence. But we ought rather to say that, assuming that changes are not radical and extensive, it is the same musical work that has survived the changes. Levinson might respond by saying that the sound structure that is indicated by the composer is somewhat vague and thus it might accept a revision and retain its identity. But this doesn't solve the problem for the following reason. Let us say there are two sound structures, S_1 and S_2, the former is the one that the composer indicated first and then he revised it and got S_2. In order to say that we have the same musical work, Levinson might suggest that it is vague that which sound structure is indicated by the composer. So the musical work is a vague indicated sound structure S instead of a precise indicated structure S_1 or S_2. This seems to solve the problem of identity but it does so with the cost of denying that musical works can change. That is, S hasn't been revised or changed; it was already there in the first place, so nothing was changed.

When it comes to software the problem of change gets even more serious. I have already discussed above the differences between an algorithm and software and argued that identifying the software with the algorithm will not work since the software may retain its identity even when it comes to employ a different algorithm. Changes in software happen all the time; programs are updated (meaning their text and algorithm are changed) quite often due to the security reasons, or cleaning the bugs in the program, or adding new features and tools, etc. Therefore, although software installs new updates, and its algorithm changes with every update, we want to be able to say that it is still the same software.

The objection above indicates that Levinson's theory of musical works and its application to software, though on the right track, fails to provide successful answers to the demands that we have set so far.

(III) Software as an abstract artifact

Let us return to the question what kind of objects pieces of software and musical works are. I think that both software and musical works are abstract artifacts which are created by software engineers or composers with certain intentions. As I have argued above they cannot be identified with any physical object like copies of scores, CD-ROMs, circuits, etc. or with any events such as musical performances and software executions. The obvious reason for this is that they have different identity and persistence conditions. Furthermore, I have argued that they are abstract objects in the sense that they can be created and destroyed, but not platonic abstract objects which are eternal non-spatio-temporal objects. It seems to me that the idea that software and musical works are created is so central to our beliefs that we cannot consistently hold those beliefs and also believe the claim that they are eternal and mind-independent entities just like sound structures or algorithms.

If we agree that software is not a platonic entity, that it is created and can be destroyed and that software survives changes in its algorithm or text, then it would be wrong to claim that software falls under the general ontological category of type. Type is usually taken to be a species of universals. Given that it is true, type is not the appropriate ontological kind for software as universals are not usually treated as entities that survive change or can be destroyed. However, I do not have a settled view on the nature of types. If there is a plausible theory of types in which they can change or be created or cease to exist, then I don't see any problem to take software as falling under the category of types on that theory. In the absence of such theory I conclude that computer programs are not types and thus the relation between computer programs and their physical copies cannot be understood in terms of the type/token distinction.

Both software and musical works come into existence when an author or a group of authors indicate a certain algorithm or a sound structure with the right kind of intentions; intentions to create those sorts of things. This act of creation is an act of coming up with the score or the algorithm and by writing them down creating the original copy of it. The existence of software and musical works does not solely depend on a particular copy; even if the original copy is destroyed right after its creation, the software or the musical work does not thereby cease to exist, because the author(s) might create another copy of the same artifact. The existence of these artifacts depends on their authors, copies, performances or executions and memories. The existence of software and musical works depends on the above entities, processes, events, etc. in the sense that necessarily whenever one of them exists at some time software or musical work exist at some time. The existence of software at some time depends on the existence of its author(s) at some time; just like any other artifact musical works and software come into existence by some human being's act of creation. Given that there are copies of a piece of musical work or software at some time, then that musical work or software exists at those times even if their creators ceased to exist, or they are not performed or executed, or no one remembers them, etc. Similarly, if a piece of software is executed at some time, it exists at those times even if there are no remaining copies of it or no one remembers it, etc. If there were no remaining copies, no performances (or executions), but someone remembers the algorithm (or the score) of a piece of software (or a musical work) at some time then they exist at that time. Both musical works and software persist through time by having one of the things on the above list (i.e. copy, the author(s), execution or performance, memory) existing. However, they are not eternal entities. Although they do not have a spatial location, they are created at some time and can be destroyed if and only if all of the following conditions are met:

(1) Their authors ceased to exist.
(2) All of their copies are destroyed.
(3) They are not performed or executed ever again.
(4) There are no memories of them.

Think of a very old computer program, like NIM, a very old computer game created in 1951 by Ferranti Inc. which was played on a computer called NIMROD that was exclusively designed to play NIM. A replica of NIMROD has been built for and exhibited in the *Computerspielemuseum* in Berlin. Let’s assume that every copy, including the one in the replica which is exhibited in the museum in Berlin has been destroyed, and further assume that by some historical accident no one, even the engineers of Ferranti Inc., a UK electricity company, who created it, remembers the NIM or the NIMROD; it has completely been vanished from the memories of people who have seen it displayed at the Festival of Britain in 1951. In that case, I argue that the NIM would cease to exist; it would be destroyed. Perhaps, it is easier to imagine similar scenarios today. There are thousands of new computer programs created every minute today, and most likely some of them have never been made public, forgotten by the very people who created them. It is not hard to assume for some of them that all of their copies are lost and memories of them are vanished. If I am right, then we should say that those computer programs ceased to exist and thus computer programs are not eternal entities such as numbers, mathematical formulas etc. One might argue that simple algorithms, like the algorithm of NIM, could be indicated by a different author at some later time, for instance, after every copy of it is destroyed and no memory of it remains. One might continue to claim that if the same algorithm is indicated at some other time, then why not say that it is the same software, NIM. I argued above that different software might have the same algorithm, and if I am right, then when the same algorithm is indicated again (after the destruction of all the copies of and memories of software that has that algorithm) in a different historical context, by different author(s), with different purposes, etc. a new software is created even if the same name, say NIM, is given to it.

I conclude that unlike platonic entities, software and musical works depend for their existence on humans. A further conclusion is that they can be destroyed if all their copies, performances, executions and memories about them are destroyed. They, therefore, are abstract only by virtue of lacking spatial location.

(IV) Conclusion

Although software is ubiquitous and a quite indispensible part of human life not much has been done to understand its metaphysical nature. In this paper, I argued that the proposed accounts of the nature of software are inaccurate and wrongheaded. I offer a novel account of software according to which software is an abstract artifact. In this new account I greatly benefited from the discussion on the nature of musical works. Looking at the ontology of musical works is motivated by remarkable similarities between musical works and software. I argued that like musical works, software is an abstract artifact. Both software and musical works are created by an act of human being(s) with the right sort of intentions. One difference between software and musical works might be that there are different kinds of intentions behind the creation of these artifacts. Musical works and works of art in general are not created in order to serve a practical purpose or to have a practical use. However, what is typical about software like a piece of software that is developed for airline flight booking and reservation system is that it is a technical artifact. A technical artifact is an artifact that is intentionally made to serve a given purpose; an object that is designed for achieving practical goals. However, there are many computer programs that seem to lack any practical goal. Some computer games, digital audio and video files might be good examples of such software. A further problem for drawing such a distinction between software and musical works or works of art in general is that there are pieces of software that are intended to be works of art.[16] It seems that the issue of ontological differences between software and musical works is more complicated than it looks and requires a careful discussion. I, therefore, leave the discussion for future investigations.

The existence of software depends on certain other things, processes, and mental states such as physical copies, executions and memories. Software could be destroyed if all the things that its existence depends on are destroyed. Therefore, unlike other abstract objects, like numbers, propositions, concepts, etc. software is creatable and destructible thus it is not eternal entity. The purpose of this paper is to introduce a novel account of software and to contrast it with the accounts proposed so far. Nevertheless, there is still much to say about the ontology of software. For instance more should be said about the kind of dependence relations[17] that hold between software and copies, memories, executions, and so on. Moreover, work still needs to be done on questions such as how software changes, what the identity conditions for software are, and more. However, answering all those questions would require a different paper.

[16] For interesting examples of software art see: http://www.runme.org/ and http://www.year01.com/code/.

[17] For different kinds of dependence relations see Thomasson (1999).

References:

Colburn, Timothy 2000: *Philosophy and Computer Science*. Armonk, NY.: M. E. Sharpe Publications.

Deutsch, Harry 1991: “The Creation Problem”. *Topoi* 10, 209-225.

Dodd, Julian 2000: “Musical Works as Eternal Types”. *British Journal of Aesthetics* 40, 424-440.

Hilpinen, Risto 2004: “Artifacts”. *Stanford Encyclopedia of Philosophy*.

Ingarden, Roman 1989: *The Ontology of the Work of Art*. Athens, Ohio: Ohio University Press.

Kivy, Peter 1983: “Platonism in Music: A Kind of Defense”. In Lamarque & Olsen (eds.), *Aesthetics and the Philosophy of Art*. Oxford: Blackwell Publishing, 92-102.

Levinson, Jerrold 1980: “What a Musical Work is”. *The Journal of Philosophy* 77, 5-28.

Merricks, Trenton 2001: *Objects and Persons*. Oxford: Oxford University Press.

Moor, James H. 1978: “Three Myths of Computer Science”. *The British Journal for the Philosophy of Science* 29, 213-222.

Suber, Peter 1998: “What is Software?” *Journal of Speculative Philosophy* 2, 89-119.

Thomasson, Amie 1999: *Fiction and Metaphysics*. New York: Cambridge University Press.

Thomasson, Amie 2004: “The Ontology of Art”. In Peter Kivy (eds.), *The Blackwell Guide to Aesthetics,* Oxford: Blackwell, 78-92.

Thomasson, Amie 2005: “The Ontology of Art and Knowledge in Aesthetics”. *The Journal of Aesthetics and Art Criticism* 63, 221-229.

Turner, Raymond & Eden, Amnon 2008: “The Philosophy of Computer Science”. *Stanford Encyclopedia of Philosophy*.

Van Inwagen, Peter 1990: *Material Beings*. Ithaca: Cornell University Press.

Wetzel, Linda. 2008: *Types and Tokens: An Essay on Universals*. Cambridge: MIT Press.