INFRINGING SOFTWARE PROPERTY RIGHTS. ONTOLOGICAL, METHODOLOGICAL, AND ETHICAL QUESTIONS

Abstract

This paper contributes to the computer ethics debate on software ownership protection by examining the ontological, methodological, and ethical problems related to property rights infringement that should come prior to any legal discussion. The ontological problem consists in determining precisely what it is for a computer program to be a copy of another one, a largely neglected problem in computer ethics. The methodological problem is defined as the difficulty of deciding whether a given software system is a copy of another system. And the ethical problem corresponds to establishing when a copy constitutes, or does not constitute, a property rights infringement. The ontological problem is solved on the logical analysis of abstract machines, and the latter are argued to be the appropriate level of abstraction for software at which the methodological and the ethical problems can be successfully addressed.

Keywords Computer Ethics; Software Ownership; Intellectual Property Rights; Software Copyright; Software Patent.

1. Introduction

The identification of property rights related to software systems is one of the most discussed topics in computer ethics and it is still to be addressed in full by policy makers and legislators (Johnson 2009). On the one hand, the debate focuses on the problem of determining whether, and to which extent, intellectual property rights should be exerted on software *qua* abstract entity. On the other hand, the choice of which legal framework among copyright, patent, and trade secrets better grants intellectual property rights protection for software owners, without hindering innovation in the software industry, is still open. These debates involve different philosophical issues that ought to be addressed in advance, as they determine possible answers to the above problems. In particular, we argue, an *ontological problem*, a *methodological problem*, and an *ethical problem* to software ownership should be clearly articulated. The philosophical analysis of software systems and their relations here proposed is aimed to pave the way to a legally oriented discussion.

The main problem around which the present philosophical analysis is developed is that of property right *infringement* for software. In general terms, an infringement concerning software may take place in case a copy is used in the development of a distinct software system. However, determining precisely what it is for a computer program to be a copy of another one is a largely neglected problem in the current debates in computer ethics. We refer to this as the *ontological problem for software copies.* This problem has been tackled formally in Angius and Primiero (2018), by proposing a taxonomy of the logical relations for software systems distinguishing among exact, inexact, and approximate copies. This paper will develop upon such taxonomy to address the fundamental ontological problem of identifying the most appropriate level of abstraction (LoA) (Floridi 2008) at which software should be legally protected.

Once the ontology of copies is determined with some precision, it is essential to provide a decision method for establishing whether a piece of software is an exact, inexact or approximate copy of another piece of software. We refer to this as the *methodological problem for software copies,* which constitutes one main difficulty of the copyright scheme for software code. The methodological problem is the problem of determining whether a piece of software copies the functionalities of another piece of software. By functionality of a software system one normally intends any operation that can be performed by the said system. In this sense, a software copy is a system which has the same functionalities of another software system. Section 2 will show how functionalities cannot be subject to protection at this level of definition. More technically, a functionality can be expressed in terms of the behaviour of a system which allows it to satisfy a given requirement. In this more precise sense, a software copy is a system which has the same behaviour of another software system.

Understanding when property right infringement occurs comes prior to the problem of setting out the legal framework under which programs should be protected. As it will be extensively shown, exploring the methodological problem will offer additional arguments supporting or opposing the advanced legal schemes.

Focusing on the ontological and methodological problems first, allows one to engage in the examination of the *ethical problem for software copies*: the violation of intellectual property rights, as expressed by the article 1.5 of the ACM Code of Ethics[1], can be addressed in legal

[1] See https://www.acm.org/code-of-ethics: “Developing new ideas, inventions, creative works, and computing artifacts creates value for society, and those who expend this effort should expect to gain value from their work. Computing professionals should therefore credit the creators of ideas, inventions, work, and artifacts, and respect copyrights, patents, trade secrets, license agreements, and other methods of protecting authors’ works. Both custom and the law recognize that some exceptions to a creator’s control of a work are necessary for the public good. Computing professionals should not unduly oppose reasonable uses of their intellectual works. Efforts to help others by contributing time and energy to projects that help society illustrate a positive aspect of this principle. Such efforts include free and open source software and work put into the public domain. Computing professionals should not claim private ownership of work that they or others have shared as public resources”.

terms by the identification of whether an artefact is an exact, inexact, or approximate copy. Most legislations allow software systems' ownership to be protected through copyright of the *high-level language program*. The high-level programming language is the set of instructions that linguistically implement the intended functionalities of the system, and hence its behaviour. Copyright protection at the level of computer programs has been opposed mainly for two distinct reasons: first, the copyright system offers limited capabilities to protect from copying; second, it hinders innovation in software developments (Samuelson et al 1989). Some have argued that patenting *algorithms* is a more straightforward way of protecting software property rights (Chisum 1986). Allen Newell (1986) opposed the patentability of algorithms arguing that our understanding of them is insufficient to define when they satisfy patent eligibility criteria. Intellectual property laws specify that ideas can never be protected nor their ownership be claimed; only expressions of those ideas into a physical medium, be it a written text or a physical process, are amenable to protection. According to Newell, it is not always feasible to distinguish ideas from processes in algorithms. Others have argued that an *ad hoc* legal framework for intellectual property rights protection should be envisaged to protect *observable executions* of implemented programs (Samuelson et al 1994).

This paper argues that this debate is better reconsidered in the light of the different LoAs that each approach chooses for protection, and it highlights some of the difficulties associated to each. Section 2 recalls the computer ethics debate on legal protection of software ownership focusing on LoAs. Section 3 challenges Newell's thesis by providing an ontological analysis of algorithms distinguishing as many as three software LoAs with which algorithms can be identified. Section 4 addresses the methodological problem of determining when a given software system S' is a copy of a system S. Except the elementary case of code duplication, frequent in the context of piracy but mostly uncommon in software development, establishing whether S' is an exact, inexact, or approximate copy of S is quite a hard but crucial task. The problem, which has been underestimated by the software property rights debate, is here examined for each LoA. Section 5 reviews the literature on intellectual property rights applied to software systems and considers infringements at the different LoAs. It is argued that approximate copies may not violate software intellectual property rights and should therefore be allowed to a certain extent. Section 6 shows how our ontological, methodological, and ethical analysis applies to a case study. Section 7 concludes by highlighting future applications of the present analysis, in particular to the problem of identifying properties shared by copied computational artefacts.

2. The ontological problem

Property rights laws are conceived on the basis of the type of objects whose property is claimed, such as real estates, chattels, or intellectual objects. The latter category covers an extended range of entities: ideas, processes, artefacts, machines, textual expressions, drawings, artworks, or marks. Distinct schemes for the legal protection of intellectual property have been formulated for different types of objects, provided that no legal protection can be afforded for ideas. These schemes may take the form of patent, copyright, trade secret, or registered trademark laws (Moore and Himma 2018).[2]

It follows that the ontological problem of identifying what kind of object is software foreruns both the problem of inquiring whether software systems are eligible for intellectual property rights protection, and the problem of identifying the legal scheme that better supplies such protection (Koepsell 2003). Indeed, most of the ongoing legal and ethical debates on copyright and patents for software struggles to understand which ontological status should be assigned to software systems: machines, processes, or textual expressions. The choice is notoriously difficult. The debate presents several possible positions on what is the right object to protect, because software systems can be analysed at several LoAs. A first step in an attempt at reconsidering this vexing problem seems therefore to require an ontological analysis which has often been dismissed or not considered in sufficient depth in the debate (Koepsell 2003). One position that stresses the need for an analysis of the LoA in the description of the ontology and epistemology of software system is presented in Primiero (2016) and it distinguishes among:

- Intention: at this level one reflects on, and expresses, the computational problem to be solved;
- Specification: at this level the set of requirements needed by the solution of the problem is stated;
- Algorithm: at this level a procedure is formulated which satisfies the requirements and provides a solution to the problem;

[2] These distinctions broadly refer to the Anglo-American legislations, but they can well be applied to most western legislations. Since the aim of this paper is to carry out a philosophical analysis of software systems that precedes the legal debate, reference to specific State legislations is avoided.

- High-level programming language instructions: at this level the task resolution provided by the algorithm is implemented in linguistic constructs appropriate for the chosen language;
- Assembly/machine code operations: at this level the operations which need to be performed for the implementation of the high-level programming language instructions are translated into low-level constructs required by the related hardware;
- Execution: at this level, operations are executed by electrical charges and the information flow on the hardware.

Each LoA expresses a control structure for the level below and, at the same time, it constitutes an *implementation* of the level above. The execution level is the only *physical* implementation level, in that it makes reference to the physical hardware executing the machine code operations. If the control structure is preserved throughout each level, the resulting software system is *in principle* correct. According to this stratified ontology, no LoA taken in isolation reflects the full nature of a software system. Instead, the whole hierarchy in which each layer can be multiply realized by many different lower levels is required. From the point of view of the computer ethics debate, the problem concerning the appropriate legal schemes for software ownership protection should be classified depending on the LoA chosen for protection.

First, neither intention nor specification can be elected for legal protection. This stems from the fact that ideas and other abstract entities, like mathematical statements or the formulation of natural laws, are denied ownership claims. Neither copyright nor patent protection allows one to cover ideas. The principle for such a restriction on what is eligible for intellectual property rights protection is that ideas pertain to the public domain, while expressions of those ideas in a written text or picture, or realizations of those ideas in a process or artefact, can be protected in that those ideas are conveyed to others through human labour. Written texts, musical recordings, artworks, and other creative, non-utilitarian, expressions can be protected by copyright.[3] Copyright gives owners the “right to copy”, sell, distribute, translate, work on derivative versions thereof, and to make public performance based on a published text. Processes, machines, manufacturers, and other artefacts carrying out specific tasks, and which can be considered “inventions” in that they are new or are new improvements of existing articles, can be patented. Patents give inventors the

[3] Articles of utility, non-creative expressions such as automatically generated texts, and public domain texts are usually out of the scope of copyright protection.

monopoly over their inventions, that is, only patent holders are allowed to implement and sell their inventions; independent discoveries of an already patented machine or process cannot be patented, nor traded. Hence, software should not be investigated at either the intention or the specification level for the purposes of protection.

Second, we can consider software at the level of algorithms and their implementations in high-level programming language instructions and low-level machine-code operations. The legal debate on software property rights and their protection started in the 70's and it was mostly concerned with the question of whether the ownership of software systems should be defended through copyright or patent. Most of the software released today is copyrighted; however, some difficulties related to the copyrighting of high-level language programs keep the debate still going.[4] This debate mainly concerns the ontological problem of whether the algorithm (as a process) or the code (as a text expressed either in a high-level programming language or in a machine code language) should be subject to legal protection. Among the first ones in doing so, Calvin Mooers (1975) argued that copyright is the best available legal scheme for software property rights protection. Framed in the model of LoAs, the argument for copyright reflects the principle that ideas conveyed by the intention and specification LoAs, which as such are not covered by intellectual property laws, can instead be protected once fixed in any tangible medium of expression one finds at lower LoAs. Tangible expressions of this sort include, according to Mooers, expressions of algorithms, either in the form of diagrams or of other textual expressions, as well as source codes and object codes of developed programs, i.e. high-level programming language instructions or machine code operations. Copyright is to be preferred in that any text can be copyrighted without restrictions, and it prevents not only from copying, but also from translating or making modified versions of it, thereby covering the case of copied programs which are expressed in a different high-level programming language.

One difficulty associated with the copyright protection of the high-level language set of instructions is that of assessing the so-called *non-literal* infringement. With this term one refers to the infringement of copyright concerning elements that go beyond the actual text. Consider a published novel: plot, incident, characters, and setting are all non-literal expressions that cannot be copied by a distinct novel under copyright laws. Different novels

[4] Just after the implementation of programmable computing machines, computer programs were developed and freely exchanged among programmers. As a result of the marketing of general purpose personal computers, software was sold and thus its ownership protected. In the US, software was initially patented until 1976, when an amendment of the Copyright Act extended copyright protection to software code (see Koepsell 2003, ch. 5 for an historical overview of the copyright/patent legal debate for software).

are nonetheless allowed to develop upon the very same idea (e.g. a man has been killed and the police looks for the guilty). In the case of software, non-literal elements include data, data types, models, design processes and so on. In the US, in 1992 (Computer Associates *vs* Altai) it was ruled that non-literal structures of computer programs are protected by copyright. Distinguishing the unprotectable ideas from non-literal protectable elements of a released software is not an easy task when the source or object code is copyrighted.

Samuelson (2016) identifies as many as five ways in which US Courts carry out such a task in software copyright infringement lawsuits. The most widely used, often called the 'abstraction-filtration-comparison' test, breaks down the high-level program into parts, such as routines and subroutines, which are then analysed to establish whether it contains non-protectable ideas and public-domain material, in order to filter them out. Whatever remains is compared for infringement. The main result of the applications of this method is that copyright may extend even to non-literal elements of a computer program expressed in a high-level language, but it does not cover its structural elements, expressed by the implemented abstract machine.

Moon (2015) reviews the case Karum vs. Fisher & Paykel Finance from 2004, in which syntactically and structurally different high-level code was observed in the legacy evolution and adaptation of a software platform and treated as non-literal infringement of software copyright. He suggests the necessity of identifying the levels of abstraction undergone by the product at the level of design: while this choice is methodologically close in spirit to our identification of LoAs, identifying levels of abstraction within the design process makes the 'abstraction-filtration-comparison' test dependent on the contextual and accidental development process followed, and therefore it presents limited generalizability.[5] Moreover, with current distributed large software, the identification of a precise design method becomes an extremely complex task. Courts have already ruled against the copyright protectability of the non-literal abstractions represented by functionalities of the whole system, functional characteristics of individual software elements and business logic, business rules or any algorithm. Moon (2015) then suggests that the problem of infringement should be accounted for code as performing the corresponding functionalities, which means that text is merged with the idea. In particular, the data flow diagram could be considered for infringement

[5] Software development methods, such as the Waterfall, Spiral, or Agile models, focus, in principle, on three LoAs, namely the levels of requirements, design of algorithms and their relations, and implementation of algorithms in high-level language programs (Turner 2018). However, actual applications of those methods may not refer to all of those LoAs in practice; for instance, Agile methods often move from the elicitation of the software requirements directly to the implementation of a high-level program.

analysis. We argue that an identification of LoAs independent of a specific development process is a more appropriate method and that, in particular, the identification of code as performing functionalities can indeed be obtained under a very specific reading of algorithms as abstract state machines, as widely shown in the next section.

Also Koespell (2003) highlighted how the idea/expression dichotomy, which feasibly fits with the protection of ordinary literary works, is far from being clearly applicable in the context of software systems and their stratified ontology. Specifications themselves, once written in a physical medium, can be considered textual expressions of intentions, and algorithms can as well be considered expressions of specifications (see also Rapaport W. J. 2018, ch. 13). Protecting algorithms, in particular, concerns the larger and more fundamental problem of the way in which algorithms and specifications are defined and are related to each other. An analysis which distinguishes algorithms as specifications and their formal translation is therefore essential in determining the difference between abstract and implemented processes.

The distinction introduced above between abstract formulation of processes and their implementation reflects the distinction between algorithms and their displayed behaviour at running time. As behaviour is largely independent from text, it is a good candidate to be protected in order to defend software property rights (Samuelson et. al. 1994). Moreover, most of the arguments that are usually carried in front of the Court to prove a copyright infringement levers on the observational equivalence between the two involved pieces of software.[6] At the same time, there is no way of establishing, with a satisfiable level of precision, the sameness of what has been called the “look and feel” of software (Samuelson et. al. 1994). One way out of this impasse may be that of considering sameness of *prescribed behaviours*, rather than of *observed behaviours*. Prescribed behaviours are computations described by algorithms, i.e. what a machine running a program that implements an algorithm *should* do according to that program. Two algorithms are the same if and only if they prescribe the very same set of behaviours. Provided that any software LoA is a correct implementation of its upper LoA, the sameness of prescribed behaviours is inherited down throughout the cascade of abstraction levels. This means that any two high-level language programs P and Q prescribe the same set of behaviours if they implement the same algorithm A, independently of the high-level language chosen to implement A. And two machine-code programs X and Y prescribe the very same set of

[6] Samuelson *et. al.* (1994) offer many concrete examples of actual US lawsuit cases of this sort.

executions if they are two correct implementations of the same high-level language program P, independently of the architecture chosen for compiling P.

The execution level is defined by observable behaviours and no further prescriptions are made, being the lowest LoA in the hierarchy. Even assuming that each LoA is a correct implementation of the upper levels, the sameness of prescribed behaviours may not result in a sameness of observed behaviours. This may be due to malfunctioning software (Floridi et al. 2015): a malfunctioning program which was developed as a copy of another program may nonetheless carry out some different executions.

Hence, the crux of the problem appears to be the precise identification of algorithms and their implementations. Choosing algorithms as candidates for legal protection has given rise to a long debate on the patentability of algorithms (Samuelson 1990). Allen Newell (1985) highlighted some of the difficulties that are connected with the legal protection of algorithms. In particular, available models of what an algorithm is are of no help in determining whether algorithms can be identified with patentable processes. Newell starts with the definition of algorithms as step-by-step procedures for the solution of a specified class of problems;[7] as such, algorithms could be patented. However, it is not uncommon to see, in the development of a given software, algorithms expressed in terms of informal and non-procedural specifications, leaving to the interpreter the duty of including the steps to be executed to implement the specified behaviour. And programs written in high-level programming languages can be considered algorithms themselves, which are nonetheless subject to yet another different form of legal protection, namely copyright. *“Fixing the models is an important intellectual task”* (Newell 1985, p. 1035) to understand whether algorithms can be the object of legal protection for software. Indeed, what algorithms are is an important question still to be answered (Hill 2016, Vardi 2012). *“It is a job for lawyers and, importantly, theoretical computer scientists. It could also use some philosophers of computation, if we could ever grow some”* (Newell 1985, p. 1035).

The problem is conceptual: there are different readings of the term ‘algorithm’ denoting them (again) at different levels of abstraction. The next section offers a model of the ontology of algorithms that aims at clarifying in which form they can be understood as candidates for legal protection.

[7] Seemingly, this definition would exclude non-terminating procedures such as those characterizing reactive concurrent systems. It should be nonetheless noted that finite fragments of non-terminating procedures, and carrying out a specific computational task, do satisfy Newell’s starting definition of algorithms.

3. Algorithms as layered artefacts

Most of the discussion on the eligibility of algorithms to legal protection for software ownership started with the famous 1972 Benson lawsuit case in the US. Benson submitted to the US patent office an algorithm for transforming binary coded numerals in pure binary numerals. The patent office denied granting patent to Benson's algorithm and the Supreme Court agreed in stating that the converting algorithm is to be considered in the same manner as a set of mathematical statements. Here a first observation comes to the fore: under a certain interpretation, algorithms are manifestation of ideas, and as such are not patentable. Chisum (1986) criticised the Supreme Court decision on the Benson case arguing that a categorical mistake was made in failing to distinguish the mathematical problem, i.e. converting decimals to binary numerals, from the algorithm conceived to solve the problem. Algorithms are step-by-step effective methods which can be understood as processes and, as such, they can in fact be subject to patent protection.

This divergence of views highlights the complex nature of algorithms, object of inquiry both in formal (Moschovakis 2001, Gurevich 2012) and in philosophical discussion (Hill 2016). This distinction has explicitly emerged in the debate under investigation, for example in Newell (1986) where more than one algorithmic level is identified in the definition of software. But a major limitation in the analysis of software protection is the understanding available to lawyers and lawmakers of the layered ontology of algorithms at the basis of software. In this section we review such issue.

From the point of view of the software development process, a first high-level sense of the notion of algorithm is that of an *informal description of an input/output relation*. For a standard example, consider the informal specification of a sorting algorithm in natural language:

"Given an unordered list L, order each element and the successive one according to the less-than relation, and repeat the operation between pairs until you obtain an ordered list L' containing all and only the elements of L".

Note how this specification for a program is not an algorithm from a technical point of view, as it does not describe precisely all procedural steps required to proceed with the ordering. When a process (to be realised algorithmically) is understood in this sense, it subsumes a denotational reading of processes: two distinct processes which denote the same Inupt/Output (I/O) relation are, to all effects and purposes, indistinguishable from each other at this level. Obviously, from a legal viewpoint this is highly unsatisfactory. One would want to be able to identify when one instance of a common I/O relation has been formulated which is (not) original with respect to other existing ones.[8] In order to restore such a characterization which reflects more closely the step-by-step nature of algorithms at lower LoAs, it is essential to focus on two important properties: linguistic formulation and implementation. Adding the former characteristic means to provide a structural or procedural way to instantiate those I/O relations in ways that allow to describe how the algorithm should be executed. This sense of the notion of algorithm corresponds to a *procedural description in a given informal language*. *Mergesort*, for example, will specify the input data type as a list, use a cardinality condition on the sorting process, specify what and how to merge the head and the tail of the list given as input through a recursive step:

"Given an array of elements as input, divide it in two halves, and proceed dividing until the subarray is reduced to a single item. Then each subarray is by definition sorted (because it has only one element). Then take two sorted subarrays and start merging by respecting the less-than relation. Continue merging until all subarrays have been included and an order array obtained".

This level of definition has the advantage of allowing to identify equivalence classes of algorithms: in this sense, two different procedures of the same I/O relation can be identified if they reflect distinct recursive constructs.

It should be noted here that only algorithms intended as specifications or, at best, as their linguistic constructions may fall under the common objection that, as being identifiable with ideas or mathematical statements, they cannot be subject to legal protection. Chisum's (1986) contention that the definition of the computational problem should be clearly distinguished from the algorithm for the solution of the defined computational problem can be now addressed by further specifying the layered nature of algorithms. The way algorithms are classified is by their complexity properties, and while certain complexity measures can be already identified at the level of their (in)formal linguistic construction, essential algorithmic aspects require us to add the second characteristic mentioned above, namely implementation. The highest level at which implementation can be reflected is by considering algorithms as descriptions of the execution of a given program *P* on an abstract machine *M* for the process corresponding to an I/O relation. An abstract machine can be translated into a State Transition System (TS) defined by a set of states, state transitions, initial and final

[8] The general problem of determining whether a semantic property has a corresponding I/O procedure satisfying it is in general undecidable by Rice's Theorem (Rice 1953). The current formulation is decidable, as it compares to an existing algorithm.

states. At this level, *algorithms are implementations of mathematical statements by abstract machines*. The abstract machine implementation of a given algorithm *specifies all the prescribed behaviours of any program implementing that algorithm.* At this LoA, two algorithms as abstract machines A and B are the same algorithm if and only if they prescribe the very same set of computations. Mergesort can, for example, be implemented by an abstract machine of the following kind:

```
Algorithm MergeSort(A)

Input : true
Output : (Permutation A) ∧ (A sorted)
Method : if |A| > 1 then
            B ← A[0..|A|/2];
            C ← A[|A|/2..|A|];
            MergeSort(B);
            MergeSort(C);
            Merge(A, B, C)
```

As different machines can implement the MergeSort algorithm as a procedure, here the problem of legal protection seems to be addressable. Note that we did not reach yet the level of high-level programming language instructions executable by *physical* machines: abstract machines specify all the processes that any implementing physical machine is allowed to carry out, thereby satisfying the need of focusing on the system execution level. On the other hand, those processes are specified independently of any chosen high-level programming language instructions to be executed by the physical machines, thereby dodging the underlying problems associated with code protection. The abstract machine for MergeSort can, for example, be implemented by the following Java high-level language instructions:

```
public static void mergeSort(int[] a, int n) {
    if (n < 2) {
        Return;
    }
    int mid = n / 2;
    int[] l = new int[mid];
    int[] r = new int[n - mid];
    for (int i = 0; i < mid; i++) {
        l[i] = a[i];
    }
    for (int i = mid; i < n; i++) {
        r[i - mid] = a[i];
    }
    mergeSort(l, mid);
    mergeSort(r, n - mid);
merge(a, l, r, mid, n - mid);
}
```

In conclusion, the ontological analysis of algorithms and their layers referred to in this section requires that algorithms be understood at three LoAs, distinguishing among:

- Algorithms as specifications;
- Algorithms as procedures;
- Algorithms as abstract machines.

If algorithms are themselves layered entities, the ontological analysis of software system into a hierarchy of LoAs, assumed in the philosophy of computer science and summarised in the previous section, is to be completed as follows:

- Intention: at this level one reflects on and expresses the computational problem to be solved;
- Algorithm
    - as a specification: at this level the set of requirements needed by the solution of the problem is specified in terms of an informal description of an input/output relation;
    - as a procedure: at this level a procedural description in a given informal language is provided that implements the input/output relation provided by the upper level;
    - as an abstract machine: at this level procedures are implemented as *prescriptions* of the executions of a given program *P* on a machine *M* for the process corresponding to an I/O relation;

- High-level programming language instructions: at this level the task resolution provided by the algorithm as an abstract machine is implemented in linguistic constructs for the chosen language;
- Assembly/machine code operations: at this level the operations which need to be performed for the realization of the programming language instructions are implemented into the low-level constructs required by the hardware;
- Execution: at this level, operations are physically implemented, e.i. executed by electrical charges, and thus *observable*; the information flow on the hardware.

The reformulation of the hierarchy of software LoAs just provided, on the one hand makes sense of Newell's remark that the term 'algorithm' is polysemous and may refer to a variety of computational entities. On the other hand it allows one to distinguish unprotectable from protectable expressions of algorithms. Algorithms as specifications and algorithms as procedures, by being identifiable with ideas and mathematical statements, cannot be granted legal protection. Claiming ownership protection of the high-level language instructions, or even of the execution LoA, is limited by the difficulties discussed in the computer ethics literature recalled in the previous section.

We argue that the algorithm as abstract machine is the only LoA at which legal protection is feasible, independently from the legal framework chosen for protection. We claim that such approach covers all properties in the software development process which should be covered by protection, including optimization properties of programs which can be expressed by refinement operations on abstract machines.[9] Protecting software property rights at the abstract machine level allows one to take advantage of formal tools, such as process algebra, used in theoretical computer science to examine the formal relations holding between two or more distinct algorithms as abstract machines. Angius and Primiero (2018) provide a taxonomy of identity and copy relations between two software systems S and S' in terms of the formal relations holding between the abstract machines realized by S and S'. In particular a distinction is made among *exact*, *inexac*t and *approximate* copies, providing a formal analysis of the potential copy relations considered in (Samuelson et al 1994). This shows two decisive advantages of protecting algorithms as abstract machines: on the one

[9] For the translation of abstract machines into TSs, given a relation of equivalence over (initial and final) states, an abstract machine M' is called a correct refinement of an abstract machine M if and only if for each execution of M' there is an execution of M such that initial states of each are equivalent, there are reachable equivalent states, both executions terminate and their final states are the last pair of equivalent states; or both executions are infinite.

hand, they provide a fair solution to the methodological problem of establishing whether a software system S' is a copy of a system S, which we turn to analyse in the next section. On the other hand, distinguishing inexact and approximate copies from exact copies allows to address the ethical problem of establishing when, and to which extent, one is authorised to reuse protected algorithms as abstract machines in the development of distinct software systems.

4. The methodological problem

The debate on the software intellectual property rights almost unconditionally starts from the assumption of two software systems of which one is a copy of the other. But the difficulty of the methodological problem of establishing under which conditions this is the case, seems an underestimated issue (Samuelson 2016). This limitation appears even more troublesome if one considers that the notion of copy requires a qualitative and quantitative specification: on the one hand, one needs to understand what type of copy one is observing; on the other hand, for some of such types, it is crucial to know how much of a given system has been copied. This latter aspect is especially important: how many of the functionalities have been copied? Being able to answer this question means to offer a solution to the methodological problem, which in turn provides an additional argument in favour of the legal protection of algorithms as abstract machines. Let us consider what types of copy relations for two software systems S and S' can be identified as soon as one considers a functional analysis in terms of their abstract machines.

According to the taxonomy of the copy relations provided in (Angius and Primiero 2018), the notion of exact copy captures the most common sense under which S' is a duplication, or replica (Carrara and Soavi 2010), of S. This is for example the case in which a piece of software is copied from one medium (say a flash drive) to another medium (such as a hard drive). At the algorithm as abstract machine LoA, S' is an exact copy of S in case S' prescribes the very same set of behaviours prescribed by S. Exact copies at the high-level programming language instructions LoA should be characterized by the very same code, that is, expressed in the same programming language. When considering the programming language LoA an implementation of the algorithm as abstract machine LoA, then it is possible to verify the exact copy relation also for different codes expressed in different languages, even pertaining to distinct programming paradigms. In other words: given two programs, one encoded using, say, a procedural high-level language and the other a functional language, if they are in an exact copy relation then they prescribe the very same set of behaviours. This is the case since, by being exact copies, they correctly implement the same algorithm as abstract machine. At the execution LoA, S' is said to be an exact copy of S only in case S' is able to perform the same set of observable executions. Note however that the latter case can be induced even by structurally different programs.

A weaker relation is instantiated by *inexact copies.* S' is an *inexact copy* of S if S' copies all the functionalities of S, but implements additional functionalities not realized by S. At the algorithm as abstract machine LoA, if the abstract machine for S is given by a set-theoretic structure like a TS, then the corresponding abstract machine for S' will be a superset of the abstract machine for S. In other words, states and transitions of the latter are in the set of states and allowed transitions of the former. Provided that lower LoAs of S and S' implement correctly the two algorithms as abstract machines, the inexact copy relation is inherited down throughout the abstraction hierarchy. In particular, if abstract machines for S and S' are implemented using the same high-level programming language the instruction set of S' contains the instruction set of S as a subset. In case implementations involve different high-level languages, even choosing different programming language paradigms, the two programs are in a inexact copy relations in that they correctly implement two algorithms as abstract machines which are in a inexact copy relation (and there must exist a translation between the different chosen languages). The same holds for the execution LoA: here S' is capable of executing all the behaviours of S together with a set of executions not executable by S.

By contrast, when S' is an *approximate copy* of S, S' is said to copy only *some* of the functionalities of S, thereby allowing, for both S' and S, to have functionalities not implemented by the other system. At the algorithm as abstract machine LoA, this can be expressed by requiring that the intersection of the prescribed behaviour sets of S' and S is not empty. The programming language used to implement the two abstract machines in a set of instructions being or not the same, the two high-level instruction sets inherit the approximate copy relation. Observationally, the approximate copy relation is given by the fact that some executions characterise both S' and S while some executions are performed only by S' and some others only by S.[10]

This paper takes the methodological problem in the software property rights debate to be the problem of establishing whether two different software systems, independently developed

[10] See (Angius and Primiero 2018) for the corresponding formal treatment of the exact, inexact, and approximate copy relations.

and released to the market, are in an exact, inexact, or approximate copy relation. Except the case of exact copies, this is far from being an easy task.

A bitwise operation at the source code level may be able to reveal whether S' is an exact copy of S by comparing bit patterns of S' with bit patterns of S.[11] Nevertheless, bitwise operations of this sort do not allow to evaluate whether S' is an exact copy of S in case the two systems are implemented within two different programming languages. Assuming that exact copies are characterized by observational equivalence of executions, one may argue that observers could detect whether two running systems are implementations of the same high-level language program. Again, observational equivalence is the kind of evidence that is usually carried in lawsuits regarding software copyright or patent infringements (Samuelson *et. al.* 1994). However, the methodological limitations affecting observational equivalence, analysed in details below in this section, may prevent one from distinguishing exact, from inexact and approximate copies at the execution LoA.

Even in case S' and S are encoded in the same high-level programming language, neither bitwise operation, nor observational equivalence, may be sufficient to evaluate whether S' is an inexact or an approximate copy of S. For the purpose of the present discussion, this means that the legal protection of the source code, or even of the software execution set, do not safeguard developers and owners from property rights infringements.

Suppose that a bitwise operation brought to the conclusion that not all bit patterns of S' are bit patterns of S; in other words, S' contains additional code lines. This does not necessarily mean that S' is an inexact copy of S. To evaluate so, one has to show that the additional code lines in S' implement some additional function not realized by S. This can only be done by considering the abstract machines specifying the prescribed behaviours of S' and S.

The problem is even more troublesome for approximate copies. In order to show that S' is an approximate copy of S after a bitwise operation ended with a negative answer, one has to ascertain whether the additional code lines of both S' and S implement some supplementary functionalities not implemented by the other system. Therefore, the methodological problem of establishing whether S' is an inexact or approximate copy of S, independently of the

[11] We are assuming to perform a bitwise operation at the source code level only and not also at machine code level since, assuming correctness of implementations, any result of a bitwise operation at the high-level language instruction set level is inherited at the assembly/machine operation code level but not *vice versa*. Suppose a bitwise operation at the machine code level reveals that bit patterns are different. This does not imply that the source code is different. For instance, two copied and thereby identical high-level instruction sets result into different machine code operation sets in case different compiler implement the source codes into two different architectures.

high-level languages used for encoding, reduces to the problem of establishing the logical relations holding between the algorithms as abstract machines of S and S'. At the basis of such reduction is the choice of defining copy relations on the basis of prescribed behaviours. While the behavioural aspect of our analysis complies with the shared view, in computer ethics, that the software's "look and feel" is what should determine whether a property right infringement occurred, one may argue that the execution LoA better fits to compare the "look and feel" aspect of software, that is, that observable behaviours should be compared.

Comparing the executions of S' with those of S implies testing both systems. Accordingly, the methodological problem of evaluating whether S' is a copy of S at the execution level is constrained by the same impediments of the engineering practice of *software testing* (Ammann and Offutt 2016). In general terms, software testing is the practice of launching a program for a given interval of time to see whether the executed behaviours comply or do not comply with a given (set of) algorithm(s), expressed as specification, procedure, or abstract machine. In the latter case, software testing consists in evaluating whether observed behaviours match with prescribed behaviours.

Determining whether S' is or is not a copy of S on an observational basis would require to observe S running and then testing S' against the observed behaviours of S. In other words, observed behaviours of S act as prescribed behaviours for testing S'. Evaluating whether S' is an exact copy of S turns out to be, in principle, the most difficult case. It implies testing whether all the executions of S are executions of S' and whether all the executions of S' are executions of S. However, it is a well-known fact that testing all the observable executions of a software system is not a feasible task: possible inputs of non-naive software are potentially infinite, testing embedded and reactive software being cases in point. This would require testing the two systems for an infinite interval of time (Ammann and Offutt 2016). Moreover, the path complexity of even minimally non-trivial system makes testing a limited source of certainty for software reliability (Symons and Horner 2014). Testing whether a given system executes a specified behaviour gives rise to a problem of enumerative induction and only probabilistic evaluations on the executable computations of a system are available through software testing (Angius 2014). Since, as it will be extensively shown in the last part of this section, decision methods for the exact copy relation are available at the algorithm as abstract machine level, it is preferable to avoid probabilistic methods to evaluate whether a property right infringement occurred.

The same methodological problem arises when evaluating whether S' is an inexact copy of S, in that one needs to test all the executions of S to evaluate whether they are also executions of S'. Things may be easier, but still not unproblematic, for the approximate copy relation. In this case one needs to test that a non empty set of executions of S are also executions of S'. Even though this may not constitute a property right infringement (see next section), it is formally sufficient, for S' to be an approximate copy of S, to test whether S' is able to perform at least one behaviour observed in the execution of S. Therefore the task reduces to test whether a given behavioural property of S is also a property of S'. The difficulty of such a task depends on the kind of behavioural property that is to be tested.

Temporal properties, specifying that a defined state will eventually be reached or that a specified state will never be reached (or not reached), cannot be easily evaluated through testing (Ammann and Offutt 2016). The former often go under the name of *liveness* properties, when requiring that something good, like a final state, will eventually occur. The latter constitute the so-called *safety* properties, when specifying that something bad, such as a deadlock, won't ever happen. Suppose one is testing a software against a given liveness property; observing a running program for a finite amount of time may reveal that the state has been reached. Assume now that the specified state was not reached during the testing time interval; this does not mean that the system will not eventually reach it. Again, ascertaining that would require running the program for an indefinite amount of time.

This main drawback of software testing is often summarised with Dijkstra's motto: "Program testing can be used to show the presence of bugs, but never to show their absence" (Dijkstra 1970, p.7). In these cases, software engineers only test those executions that may reveal a violation of the behavioural property to be checked. For instance, a way out to test, say, a safety property is to keep on observing those executions that may violate the required property, such as executions going through the undesired state. Those executions are selected by considering *models* of the software system's executions, which may take the forms of data flow graphs, boolean models, input domain models (see Ammann and Offutt 2016), or even TSs for abstract machines (Callahan *et al.* 1996). In case TSs are being used, paths violating the liveness property are isolated in the formal model and the software system is subsequently tested using inputs corresponding to the initial state of the selected paths.[12]

Without going any further in the analysis of the software testing engineering practice, it can be concluded that testing whether S' is a copy of S is either infeasible (for exact and inexact

[12] A path from an initial to a final state of a TS is given as a set of states and a set of transitions such that each state holds a transition relation with its successor state. For a more detailed and formal epistemological analysis of the practice of using abstract machines to improve software testing see (Angius 2013).

copies) or it requires algorithms as abstract machines (for approximate copies). In the case of approximate copies, one should also be careful to notice that testing whether an execution performed by S is also allowed by S' may give rise to false negatives, in case of malfunctioning of S' with respect to such behaviour. Additionally, false positives may be given as well in case the same execution is performed as implementations of two different abstract machines. It follows that the algorithm as abstract machine alone expresses the feasible LoA for checking whether two software systems are in a copy relation and that prescribed, rather than observable, behaviours are those that ought to be legally protected. What follows highlights how also exact and inexact copies can be easily detected at the abstract machine level.

Process algebra (Fokkink 2013) provides automatable means to check whether S' is an exact, inexact, or approximate copy of S by considering the logic relations holding among abstract machines. The formal definitions of the three copy relations in Angius and Primiero (2018) are given in terms of the process algebra notions of *bisimulation* and *simulation* between any pair of TSs for the involved abstract machines. Then the three relations of copy receive appropriate formal counterparts as follows:

- The *exact copy* relation between two systems S' and S can be represented by the bisimulation relation ≡ holding between the transition system TS' for S' and TS for S. TS ≡ TS' holds iff TS' simulates all the behaviours of TS and TS simulates all behaviours of TS', that is, iff TS' prescribes all and only the behaviours prescribed by TS.[13]
- The *inexact copy* relation can be represented by the process algebra relation of *simulation* ≤ holding between the copying TS' and the copied TS. TS ≤ TS' holds iff TS' simulates all the behaviours of TS, that is, iff TS' prescribes all, but not necessarily only, the behaviours prescribed by TS.
- The *approximate copy* relation TS ≅ TS', according to which TS' prescribes at least one of the behaviours prescribed by TS, can be represented by a partial simulation relation ≤, requiring that TS be able to simulate at least one path in the simulation quotient system of TS'.[14]

[13] For the bisimulation relation TS ≡ TS' to hold it is also required that TS and TS' have the same branching structure.

[14] Informally, the simulation quotient system of TS' is the set of paths of TS' that TS' itself can simulate. Since every state transition system simulates itself, the simulation quotient system of TS' is tantamount to the set of paths of TS'. If TS simulates at least one path in the simulation quotient

Bisimulation equivalence and simulation preorder can be checked in polynomial time (Baier and Katoen 2008). Accordingly, addressing what has been called here the methodological problem at the algorithm as abstract machine LoA supplies one with automatable means to check, for any two pairs of software systems, whether one is a copy of the other without incurring the difficulties limiting the protection of the source code or of observed executions. Indeed, abstract machines model all the prescribed behaviours of any implementing high-level program and of any consequent implementing physical machine. On the one hand, by comparing abstract machines one is directly aware of whether two software systems realise the same set of computable functions or whether additional functionalities are included. On the other hand, by considering all the behaviours of a software system that are prescribed by its TS one is able to overcome the running-time constraints of testing. Additionally, checking for approximate copies at the algorithm as abstract machine level also allows to examine whether the copying and the copied systems share specified liveness and safety properties.[15]

The distinction among, and the formal definition of, exact, inexact, and approximate copies put forward new questions for the computer ethics debate on software ownership protection. The ethical, and thereupon legal, problem of whether, and under which circumstances, copying violates property rights for software (Nissenbaum 1995), can now be more profitably examined as the problem of whether exact, inexact, and approximate copies constitute infringements of intellectual property rights.

5. The ethical problem

In order to understand when an infringement occurs for software, and therefore when legal protection is needed, we need to preliminary recall when intellectual property rights in general can be claimed.

A "*rule-utilitarian*" argument is usually put forward by consequentialists to contend that both *copying* and *reusability* of programs should not be allowed (Moore 2008). Whereas permitting copying implies producing replicas in the exact copy relation sense provided in the previous section, favouring reusability means, especially in the object-oriented programming paradigm, allowing for objects and methods of those objects to be taken from one developed software and included into another development project. According to the '"rule-utilitarian"

system of TS', than TS' and TS have at least one path in common, that is, they prescribe at least one common behaviour.

[15] Liveness and safety properties correspond to paths in the checked TSs and copies share such properties in case they simulate, among others, exactly those paths.

argument, copying and reusability harm the software industry in that the decrease in income of developers results in minor investments towards innovation with a consequent loss of social utility (Moore 2008, pp. 110-119). Two fallouts foreseen by consequentialists are a decrease in software production, and the following increase in prices (Barwise 1989). But it may also be the case that permitting copying and reusability would cause a decrease in the prices of software insofar as developers can take advantage of available free objects and codes. In other words, developing software in a non-proprietary regime would be economically less appealing for programmers, developers, and software companies.

A deontological argument for intellectual property right is also formulated by levering on John Locke's theory of property contained in the *Second Treatise on Government* (Locke 1690). According to Locke, property of physical objects can be morally claimed provided that

(i) they are the result of one's labour upon some natural good which, as such, belongs to the community, and that

(ii) *"there is enough and as good left for others"* (Locke 1690, section 27), that is, when the ownership claim of a given object does not prevent others from acquiring other objects of the same kind.[16]

Moore (2008) argues that Locke's theory of property can be extended to intellectual objects in so far as condition (ii) is always satisfied: the ownership claim of an intellectual object does not prevent others from owing the very same object (pp. 119-128). Also, abstract entities are infinite and not limited by physical constraints, as it is the case for physical goods whose ownership claim, by their being finite, results in a diminishing of available resources.

This becomes particularly evident if one considers the legal protection of the high-level programming language instructions. One is allowed, according to the Lockean justification argument, to claim ownership protection over a program's code in that

(i) the development of the program required the design and encoding human labour, and

[16] Intellectual property rights may also be defended following the "*personality-based*" argument contained in Hegel's *Philosophy of Right* (Moore 2008, 2011). Hegel claims that individuals possess property rights over products of physical or intellectual labour in that those products are externalizations of the author's personality, feelings, and other features of her subjectivity. As long as the Anglo-American accounts of property rights dominate the software property rights debate, the present discussion will focus on the consequentialist and deontologist arguments only.

(ii) nothing prevents others from developing a different program that nonetheless implements the same algorithm as specification.

Exerting property rights over, say, a text editor is morally sound in that others may exert their rights over different developed text editors implementing different algorithms as abstract machines and data structures.

Others, such as Gordon (1993) and Breakey (2010), contend that condition (ii) is not fully satisfied when applied to intellectual objects. Even though the legal protection of one conceived process or expressed idea does not prevent someone else from the community from envisioning different processes or expressions, new ideas often condition society and culture to the point that new ideas need somehow to rely on the previously introduced intellectual object. This contention is in line with the worries of some consequentialists who stress that protecting programs' code or algorithms (usually as procedures) hinders software innovation in that it does not allow for the reusability of those portions of code (classes, objects, functional structures), or of the corresponding algorithms, that could be fruitfully included in different software developments (Samuelson *et. al*. 1994). Released software often implements innovative algorithms for the solution of crucial computational problems, including safety-critical and computer security problems.

Advocates of the free and open source software point at this problem as one of the main difficulties associated with software ownership, arguing that it is false that it results in higher innovations in the software industry. Quite to the contrary, giving property rights up allows for the liberal circulations of ideas and the consequent increase in software innovation, as it was at the beginning of the software era (Stallman 1985).[17] Rewards for programmers and developers, they contend, may well come from governamental support, as it is done for other forms of intellectual labour, such as scientific research (Shavell and Ypersale 2001).

Others claim that software innovation can still be fostered even in a proprietary software system in case the legal protection of software is somehow loosened (Koepsell 2003). Both Gordon (1993) and Breakey (2010) conclude that as long as condition (ii) in the Lockean property theory is not fully satisfied, a *strong* property right should not be granted for intellectual objects. And Nissenbaum (1995) agrees in saying that even though the legal

[17] Besides utilitarian arguments, free software promoters also advance "libertarian" and "communitarian" claims in supports of their thesis. Proprietary software denies the liberty of modifying source code thereby impeding users to interact with programs in the way they wish. Mostly, proprietary regulations prevent from sharing the code with the community, harming those who are in need (Rapaport J. 2018).

protection of software is morally justifiable under the Lockean property theory, prohibiting any form of copying dampens the software industry and some form of copying should therefore be allowed. In the computer ethics debate on software property rights, copy is often used as a synonym of duplication, that is, in the sense of exact copy introduced in its precise meaning above. This prevents from articulating such proposal into actionable principles. On the other hand, the distinction among exact, inexact, and approximate copies may allow one to do so, provided that algorithms as abstract machines define the LoA at which property rights are exerted. Indeed, as underlined in the previous section, neither the high-level language instruction set, nor the set of observed executions, permits to detect the three kinds of copy relations.

Software ownership cannot be assured by protecting program executions, in that this would plainly fail to comply, on the other hand, with condition (ii) of the Lockean deontologist argument. Preventing executions from copying precludes the chance of developing a system that is able to perform the same observable behaviours but which implements different algorithms as abstract machines and uses different data structures. Granting legal protection of the observable behaviours of a text editor, for instance, implies denying the right to realize another text editor which does not exploit the design and encoding labor employed to develop the first editor. In Locke's words, there would not be "enough and as good left for others".[18]

We have argued above that algorithms as specifications and algorithms as procedures cannot be object of ownership protection for the same reason ideas and mathematical statements are out of the scope of intellectual property right laws. We now stress how the algorithm as abstract machine is the only LoA that, besides avoiding the objections against the protectability of algorithms as specifications and as procedures, maximises the conditions put forward by both consequentialists and deontologists for software intellectual property rights protection.[19]

First, algorithms as abstract machines do not need to be identified with any implementing high-level language program. Second, condition (ii) of the Lockean justification argument is

[18] It goes without saying that the intention, the algorithm as specification, and the algorithm as procedure LoAs, as being identified with cognitive processes and ideas, are not to be considered as the outcome of human labour, though they may well be creative processes. Accordingly, their legal protection would violate condition (i) of the Lockean justification of intellectual property.
[19] In the computer ethics debate, the rule-utilitarian argument and the Lockean justification are to be considered two distinct moral frameworks wherein to argue in favour, or against, software property rights. This section is highlighting that, despite the fact one is a consequentialist or a deontologist, she would find the algorithm as abstract machine level complying with her eligibility criteria for ownership protection.

satisfied: many different abstract machines can be built which implement the same algorithm as a specification but different algorithms as procedures. In other words, given an algorithm as a specification, that is, an informal description of an input/output relation, owning an algorithm as an abstract machine does not prevent others to develop a distinct abstract machine which satisfies the same input/output relation, together with other important properties including efficiency, but implementing a different algorithm as a procedure.[20]

Furthermore, and most importantly, consequentialists and more in general those, like Gordon (1993) and Breaky (2010), who are worried about the denial of reusability of algorithms, may find the legal protection of abstract machines complying with their conditions under the following circumstances. Section 4 underlined how abstract machines allow for the formal definition of exact, inexact, and approximate copies in terms of process algebra relations holding between any couple of TSs representing those machines. Protecting algorithms as abstract machines does not dampen software innovation provided that approximate copies be authorized. Approximate copies permit the reusability of the simulated computational paths. Clearly, also admitting exact and inexact copying of released software does not impede other developers from taking advantage of any newly discovered solution to critical computational problems. Indeed, this is what is often promoted by advocates of free and open source software. However, allowing exact and inexact copies raises the objections of deontologists who claim that intellectual property would be violated in that one is exploiting the human labor involved in the development of the copied systems with no necessity: the same system could be indeed developed implementing different algorithms as abstract machines.

Approximate copies may or may not violate intellectual property rights exerted on software. The definition of TS' as an approximate copy of TS when TS simulates at least one path in the simulation partitions of TS' allows for many systems of a given kind K to be approximate copies. A path in TS specifies a prescribed behaviour of the implementing software system; it is very likely that two systems of the same kind have at least one behaviour in common, especially when there are behaviours that can be modelled, at the abstract machine level, in very few ways. Consider, to give one simple example, two operating systems and the basic behaviours involved in the volume-up and volume-down operations to fine-tune the volume.

[20] For instance, the sorting algorithm as a specification defined in section 3 can be implemented in a varieties of algorithms as procedures and, in turn, in different algorithms as abstract machines. Besides the MergeSort procedure, abstract machines for the sorting specification can implement the Binary Tree Sort algorithm as a procedure, or the Tournament Sort Procedure, to give a couple of examples, which are nonetheless characterized by the same time complexity.

Approximate copies of this sort should not count as violations of property rights and may be considered as "*fair use*" of the copied abstract machines.

It should be also noted that exact copies are approximate copy limit cases, in which the copied TS simulates all paths in the simulation partition of the copying TS. It should therefore be established a *threshold of paths* that an approximate copy is allowed to have in common with the copied system to count as a "fair use", as it is done in the fair usage of copyrighted material, including written text or musical compositions, wherein one is allowed to quote a text or take a sample of a recorded song. On the one hand, approximate copies so understood do not violate intellectual property rights; on the other hand, they support the protection of ownership rights without hindering software innovation. Functionalities of a software system correspond to paths in a TS; the opportunity to copy computational paths favours the reusability of some of those functionalities that are considered essential in the development of similar or different systems. Accordingly, allowing approximate copies to a specified extent defines when, and under which circumstances, protected algorithms as abstract machines can be reused in other software development projects.

Not only approximate copies should be allowed, there are cases in which it should be requested for a software system S' of kind K to be an approximate copy of a system S of kind K taken as a model. Consider again software involved in safety-critical situations, such as traffic light controllers, nuclear plants, or robotic surgery systems. Safety and liveness properties play a crucial role in the developments of those systems in that they allow to specify, respectively, that something bad will not ever occur, or that something good will eventually occur. It may so be requested that, say, any software used in a nuclear plant should be an approximate copy of a model-software system with respect to selected paths successfully implementing safety and liveness properties of this sort.

6. A Case Study

A well-known case study appropriate to illustrate how the model proposed in this paper can be applied in the current legal landscape on software ownership is the dispute between Google and Oracle over the implementation of the Java Virtual Machine (JVM) Application Programming Interface (API) in the Dalvik VM for the Android Operating System (OS).

Let us recall the steps of the dispute so far. The case focused on 37 packages of source code developed by Oracle America (and its predecessor Sun Microsystems) in the development of the JVM. During the development of the Android OS, Google and Oracle negotiations for the licensing of the Java libraries on the mobile OS failed on various grounds, and Google decided to develop a new version of the VM called Dalvik. The 37 API calls in question and over 11k lines of code involved in the dispute were extracted from the open-source development of Java by the Apache Software Foundation. Subsequently Oracle (who had meanwhile acquired Sun) sued Google for both copyright and patent infringement, and damages. The copyright claim referred to the use of a specific function rangeCheck, several test files, and structure, sequence and organization (SSO) of the APIs.

In 2012 a first verdict on the copyright phase established that using a different implementation method left anyone free to write code to carry out the same function or specification, and that identity of declaration or method header lines does not matter. In this first verdict the APIs were declared not copyrightable. The Appeals Court reversed this decision, on the basis of the claim that Oracle's APIs were the result of creative and original work, and stressing in particular that the APIs SSO were copyrightable. The amount of literal copying was not considered minimal. A second trial in 2016 followed this appeal and the jury decided that the re-implementation of the APIs was protected by fair-use, in that re-implementation of the APIs was required to assure the interoperability of the Android OS. Oracle appealed and in 2017 the Appeals Court established that Google had performed not minimal, untrasformative reuse with respect not only to the literal *verbatim* copying, but in particular of the SSO of the APIs, and it was harmful of Sun/Oracle interests.

The difficulty of finding a straight solution to this case is testimony to the different assessments of the level of abstraction at which the ruling should be performed. The first ruling, by declaring APIs not copyrightable, focuses on algorithms as specifications or informal descriptions of procedures which, as such, are considered not protectable by copyright. This analysis obviously dismisses the creativity and originality of the procedures developed by Sun/Oracle, which are highlighted by the Appeals Court instead. The principle applies not so much at the level of the high-level language instructions, where the *verbatim* copying was easy to assess, but rather at the level of the SSO of the product. This can be roughly identified with the algorithm as abstract machine LoA, i.e. the linguistic implementation of the algorithm as a procedure in a formal structure which is independent of the specifics of the programming language, but able to preserve the logical structure of the processes.

Protecting the algorithms as abstract machines of the APIs, rather than any high-level program implementing them, prevents both *verbatim* copying and copying using different high-level programming language instructions implementing the same SSO. It prevents from literal copying in that the copied source code inherits the SSO of the copied high-level language instructions. Most importantly, it prevents from non-literal copying, that is, from developing a different code for the same SSO. Google was found involved in both activities. In the case of the development of the Dalvik VM, non-literal copying was achieved through a *reverse engineering* of the JVM APIs. Reverse engineering in software development is the practice of identifying functions and data structure of a software system by only analysing the machine code operations and the execution LoA of that system, in order to develop a new system implementing those functions and structure. In other words, given a developed system S, reverse engineering of S aims at developing a system S' implementing the same functionalities of S, usually using a different high-level programming language, but without having access to the source code of S. Reverse engineering is often performed when one needs to develop software which be interoperable with respect to a system whose source code is protected under copyright law. Reverse engineering allows one to copy a software system without incurring in any literal copyright infringement. According to the analysis provided in this paper, reverse engineering should count as a property right infringement if the algorithm as abstract machine is the legally protected LoA. Indeed, one is developing a different high-level language instruction set which implements copied algorithms as abstract machines.

From the methodological viewpoint it should be noted here that, had Google implemented the SSO only through reverse engineering and using a different high-level programming language, it would have been less obvious to identify the result as a copy, and only an expression of the SSO at the level of the abstract machine would have allowed the identification of the same processes. In terms of the exact/inexact/approximate distinction, the Dalvik VM is to be considered as an inexact copy of the JVM in that it copied 37 packages and over 11k lines of code, that is, it did not copied all functionalities of the JVM. An analysis of the SSO at the algorithm as abstract machine level may reveal the exact amount of computational paths that were copied, both through literal and non-literal copying. This, in turn, may allow to shed light on whether the reusability of functions claimed by Google counts as a fair use or not, as declared by the 2016 trial and subsequently disavowed by the Appeals court.

Under the ethical framework provided in this paper, reverse engineering should not be allowed in that the functionalities of a software system S can well be replicated by a system P defined by a different design, i.e. characterised by different algorithms as abstract machines and data structure. However, in order for any software system Q to interact with both S and P, it may be required that P share some of S’s design. Indeed, the legitimacy of reverse engineering is often claimed under the fair use clause in order to assure for the interoperability of the developed systems. However, while one needs to preserve the principle of the Copyright Act which allows reuse of a function, one also needs to preserve the creativity in its implementation, which the abstract machine implementation allows. Properties of reuse can be quantitatively and qualitatively assessed through an analysis at this LoA, namely by means of the number of copied paths in simulation partitions. As specified in the previous section, fair use can be better delimited by specifying a threshold of paths of a TS for an abstract machine that one is allowed to copy, thereby providing a clear demarcation of fair use from property rights infringement. This would in turn permit to determine whether Google's development of the Dalvik VM of the Android OS involved or not a fair usage of the 37 API packages from the JVM.

7. Conclusions

This paper aimed at contributing to the computer ethics debate on software ownership protection by showing how the analysis of property rights infringement poses ontological, methodological, and ethical problems that come prior to the legal discussion characterizing the ongoing debate. It has been stressed how the copyright/patent debate presupposes the ontological problem of identifying the computational LoA that better fits for legal protection. It has been argued that both the high-level language instructions and the execution LoA do not successfully allow one to define property rights infringements. While it is at the algorithm level that such assessment should be done, we have also provided an ontological analysis of algorithms to show how their hierarchical ontology requires to distinguish between algorithms as informal specifications, algorithms as (linguistically construed) procedures, and algorithms as (implementable) abstract machines. Whereas algorithms as specifications and as procedures can be identified with ideas and mathematical statements which, as such, are not eligible for legal protection, the algorithm as abstract machine level has been recognized as the software LoA to be chosen for property rights claims on a methodological and ethical basis.

From a methodological point of view, only algorithms as abstract machines allow one to establish when a software system is a copy of a former system. Abstract machines permit to define exact, inexact, and approximate copies relations in terms of process algebra relations between TSs representing them; additionally, those relations can be checked algorithmically.

From an ethical viewpoint, the algorithm as abstract machine LoA is the only one able to comply with the constraints put forward by both deontologists and consequentialists for property right allowance. On the one hand, protecting software systems at this LoA does not prevent others from developing software systems of the same kind, that is implementing the same algorithms as specifications, but using different algorithms as abstract machines. On the other hand, admitting approximate copies of abstract machines allows for the free circulation of innovative algorithmic solution to critical computational problems. Software innovation and security is not jeopardised in this way.

Two further steps are required for the present work to be translated into actionable principles. First, the opportunity of fostering software innovation and security, by favouring simulations of relevant computational paths, requires some logical work on the formalization of computer security properties. Here the task is defining what types of properties are preserved through approximate copies and under which logical constraints.

Second, the ontological, methodological, and ethical conclusions reached by this paper should guide the debate on the legal schemes for software ownership protection. As highlighted in the introduction, the present analysis is orthogonal with respect to the patent/copyright debate. Indeed, the algorithm as abstract machine LoA may well be protected using both schemes. Abstract machines are patentable *qua* processes, namely as being descriptions of the executions of any high-level program implementing them. *And* abstract machines are also copyrightable in that they can be given as a *textual description* of the prescribed behaviours of any implementing high-level program, as done for the MergeSort abstract machine in section 3. We leave to jurists and policy makers the task of choosing any or both of the two schemes.

References

Abbot, J. (2003). Reverse Engineering of Software: Copyright and Interoperability. *JL & Inf. Sci.*, *14*, 7.

Ammann, P., & Offutt, J. (2016). *Introduction to software testing*. Cambridge University Press.

Angius, N. (2013). Model-based abductive reasoning in automated software testing. *Logic Journal of IGPL*, *21*(6), 931-942.

Angius, N. (2014). The problem of justification of empirical hypotheses in software testing. *Philosophy & Technology*, *27*(3), 423-439.

Angius, N., & Primiero, G. (2018). The logic of identity and copy for computational artefacts. *Journal of Logic and Computation,* 28(6), 1293-322.

Baier, C., & Katoen, J. P. (2008). *Principles of model checking*. MIT press.

Breakey, H. (2010). Natural intellectual property rights and the public domain. *The Modern Law Review*, *73*(2), 208-239.

Callahan, J., Schneider, F., & Easterbrook, F. (1996). Automated software testing using model checking. In *Proceeding Spin Workshop*, J. C. Gregoire, G. J. Holzmann and D. Peled, eds, pp. 118–127. Rutgers, 1996.

Carrara, M., & Soavi, M. (2010). Copies, replicas, and counterfeits of artworks and artefacts. *The Monist*, *93*(3), 414-432.

Chisum, D. S. (1985). The patentability of algorithms. *U. Pitt. L. Rev.*, *47*, 959.

Clarke, E. M., Grumberg, O., & Peled, D. (1999). *Model checking*. MIT press.

Dijkstra, E.W. (1970). Notes on structured programming. *T. H.—Report* 70-WSK-03.

Floridi, L. (2008). The method of levels of abstraction. *Minds and machines*, *18*(3), 303-329.

Floridi, L., Fresco, N. & Primiero, G. (2015). On Malfunctioning Software, *Synthese* (2015) 192: 1199-1220. https://doi.org/10.1007/s11229-014-0610-3

Fokkink, W. (2013). *Introduction to process algebra*. Springer Science & Business Media.

Gordon, W. J. (1993). A property right in self-expression: Equality and individualism in the natural law of intellectual property. *The Yale Law Journal*, *102*(7), 1533-1609.

Gurevich, Y. (2012). What Is an Algorithm? *SOFSEM 2012*: 31-42.

Hill, R. K. (2016). What an algorithm is. *Philosophy & Technology*, *29*(1), 35-59.

Hughes, Justin, 1988, “The Philosophy of Intellectual Property”, *Georgetown Law Journal*, 77: 287.

Johnson, D. G. (1998). *Computer ethics, 4th edition*. Pearson.

Koepsell, D. R. (2003). *The ontology of cyberspace: philosophy, law, and the future of intellectual property*. Open Court Publishing.

Locke, John, 1690, *The Second Treatise of Government*. [Locke 1690 available online]

Mooers, C. N. (1975). Computer software and copyright. *ACM Computing Surveys (CSUR)*, *7*(1), 45-72.

Moore, A. D. (2001). *Intellectual Property and Information Control: Philosophic Foundations and Contemporary Issues*, New Brunswick, NJ: Transaction Publishers.

Moore, A. D. (2008). Personality-based, rule-utilitarian, and lockean justifications of intellectual property. In K. Himma and H. Tavani (eds), *The handbook of information and computer ethics*, pp. 105-129, John Wiley & Sons.

Moore, A. D. & Himma, K. (2018). Intellectual Property. *The Stanford Encyclopedia of Philosophy* (Winter 2018 Edition), Edward N. Zalta (ed.), URL = <https://plato.stanford.edu/archives/win2018/entries/intellectual-property/>.

Moschovakis, Y. (2001). What is an algorithm?, In *Mathematics unlimited -- 2001 and beyond,* edited by B. Engquist and W. Schmid, Springer, 2001, pages 919-936.

Newell, A. (1985). Response: The models are broken, the models are broken. *U. Pitt. L. Rev.*, *47*, 1023.

Nissenbaum, H. (1995). Should I copy my neighbor's software. *Computer Ethics and Social Values. Prentice Hall, Upper Saddle River, NJ*, 200-213.

Primiero, G. (2016). Information in the philosophy of computer science. In *The Routledge Handbook in the Philosophy of Information*, L. Floridi ed., pp. 90–106. Routledge.

Rappaport, J. (2018). Is Proprietary Software Unjust? Examining the Ethical Foundations of Free Software. *Philosophy & Technology*, *31*(3), 437-453.

Rapaport, W. J. *Philosophy of Computer Science*. Draft, 2018, available at https://cse.buffalo.edu/~rapaport/Papers/phics.pdf.

Rice, H. G. (1953). Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, *74*(2), 358-366.

Samuelson, P. (1989). Why the look and feel of software user interfaces should not be protected by copyright law. *Communications of the ACM*, *32*(5), 563-573.

Samuelson, P. (1990). Legally speaking: should program algorithms be patented? *Communications of the ACM*, *33*(8), 23-27.

Samuelson. P. (2013) Is software patentable?. *Communications of the ACM,* 56(11), , 23-25.

Samuelson, P. (2016). Functionality and Expression in Computer Programs: Refining the Tests for Software Copyright Infringement. *Berkeley Tech. LJ*, *31*, 1215.

Samuelson, P., Davis, R., Kapor, M. D., & Reichman, J. H. (1994). Manifesto concerning the Legal Protection of Computer Programs, A. *ColUM. l. reV.*, *94*, 2308.

Shavell, S., & Van Ypersele, T. (2001). Rewards versus intellectual property rights. *The Journal of Law and Economics*, *44*(2), 525-547.

Stallman, R. (1985). The GNU manifesto.

Symons, J., Horner, J. (2014). Software Intensive Science, *Philosophy and Technology*, Volume 27, Issue 3, pp 461-477.

Turner, R. (2018). *Computational artifacts: Towards a philosophy of computer science*. Springer.

Vardi, M.Y. (2012). *What is an algorithm*, *Communications of the ACM*, 55(3), Page 5.