Towards an Ontology of Software: a Requirements Engineering Perspective

Xiaowei WANG [a,1] , Nicola GUARINO [b] , Giancarlo GUIZZARDI [c] and John MYLOPOULOS [a]
[a] *Dept of Information Engineering and Computer Science, University of Trento, Italy*
[b] *ISTC-CNR Laboratory for Applied Ontology, Trento, Italy*
[c] *Ontology and Conceptual Modeling Research Group (NEMO), Federal University of Espírito Santo (UFES), Brazil*

Abstract. Although software plays an essential role in modern society, its ontological nature is still unclear. For many, software is just code, but this is not illuminating. Several researchers have attempted to understand the core nature of software and programs in terms of concepts such as *code*, *copy*, *medium* and *execution*. More recently, a proposal was made to consider software as an abstract artifact, distinct from code, just because code may change while the software remains the same. We explore in this paper the implications of such a proposal in the light of software engineering and requirements engineering literature. We make a sharp distinction between different kinds of software artifacts (code, program, software system, and software product), and describe the ways they are inter-connected in the context of a software engineering process.

Keywords. software, artifact, ontology, software engineering, requirements engineering

Introduction

Software changes all the time. Such changes have a huge impact on the software industry, so dealing with software change is absolutely necessary. In the past, several authors have proposed taxonomies intending to describe the different kinds of software change [1], [2], [3], but the very nature of such changes is still unclear: what does it mean for software[2] to change? What is the difference between a new release and a new version? How do we tell that, after a change, it is still the same software?

To address these questions, before attempting to understand how software changes, we need to understand what *software is*. In other words, we need an ontology of software which accounts for the possibility for software to change while maintaining its identity. This means that we need to explicitly account for the *identity criteria* of various software-related artifacts that are implicitly adopted in everyday software engineering practice.

[1] Corresponding Author's E-mail: xwang@disi.unitn.it
[2] The words "software" and "code" are technically mass nouns in English, like the word "sand". However, they are often used as countable nouns in everyday practice as shorthand for software *program*, *system*, or *product*, and *code base* respectively. We adopt this shorthand here.

Unfortunately, the very possibility for software to change while maintaining its identity is in practice ignored by most recent studies, which have mainly focused on the relationships between software code (intended as an abstract information pattern), its physical encoding, and its execution [4], [5], [6], [7]. A different approach is adopted by Nurbay Irmak [8], who suggests that a software program is different from software code, since the code can change (e.g. for fixing a bug) while the program remains the same. For him, this is due to the artifactual nature of software programs.

In this paper we build on Irmak's contribution, starting our analysis from his conclusions: "work still needs to be done on questions such as how software changes, what the identity conditions for software are, and more". So we shall focus on the identity criteria for software originated by its specific artifactual nature, and motivated by the need to properly account for software change. Indeed, our ultimate aim is to establish a well-founded ontological framework to understand and describe the different cases of *software change rationale* across the whole software engineering lifecycle.

To this purpose, we firstly need to understand what software is. As Eden and Turner observe [7], a peculiar aspect of software, with respect to other information artifacts such as laws or recipes, is its *bridging role* between the abstract and the concrete: despite the fact that it has an abstract nature, it is designed to produce specific results in the real world. Therefore, it seems natural to us to adopt a requirements engineering perspective while analyzing the essence of software, looking at the whole software engineering process, including requirements analysis, instead of focusing on computational aspects only. Our analysis is founded on a revisit of Jackson and Zave's seminal work on the foundations of requirements engineering [9], [10], [11], which clearly distinguishes the *external environment* that constitutes the subject matter of requirements, the (computer-based) *machine* where software functions fulfill such requirements, and the interface between the two.

Jackson and Zave define the terms 'requirements' and 'specification' as referring to the desired behavior in the environment and at the machine interface, respectively. Here we refine their terminology using 'high level requirements' to refer to desired behavior in the environment independently of the machine, *excluding* therefore the interface, 'software system specification' to point to the expected behavior *at the interface*, and 'program specification' referring to a specific behavior inside the machine, namely that of the computer that drives the machine. We shall rely on these refined notions to determine the essential properties of three different kinds of software artifacts: *software products*, *software systems*, and *software programs*. In addition, to account for the social nature of software products in the present software market, a further kind of artifact will be introduced, namely *licensed software product,* whose essential property is a mutual pattern of commitments between the software vendor and the software customer.

In conclusion, we explore in this paper the ontological distinctions underlying the different reasons for software change, by making a sharp distinction between different kinds of software artifacts, and describing the ways they are connected in the context of a software engineering process. While doing so, on one hand we shall take inspiration from the requirements engineering literature to better understand the ontological nature of software, and on the other hand we shall leverage on such ontological analysis to better understand the software engineering process, especially in the light of software change management. To illustrate these ideas, the rest of this paper is organized as follows: firstly, literature works are reviewed in section 1; in section 2, we recognize software as artifact different from code; then, in section 3 and 4, different kinds of

software artifacts are analyzed through the perspective of requirements engineering, and the result is a layered ontology of software artifacts; section 5 discusses an additional software artifact from social and marketing perspective. Section 6 summarizes our contributions and sketches future work.

1. Related Work

In the literature, the term "software" is sometimes understood in a very general sense, independently of computers. For example, Osterweil believes that, in addition to computer software, there are other kinds of software, such as laws or recipes. In a nutshell, he characterizes software as something non-physical and intangible, which could be executed to manage and control tangible entities [12]. Suber assumes software as an even more general concept, defining software as (physically embodied) patterns which are readable, liftable, and executable by a machine [13]. From this definition, we can derive some extremely unintuitive cases, such as "*all circuits deserve the name software, since they are physical embodiments of patterns, readable and executable by a machine, and liftable.* [13]*"*

Although these ideas are certainly intriguing, we focus on a proper ontological account of *computer* software, which is still missing in the literature. Focusing on the computational aspects, several scholars have addressed the complex relationships within i) a software *code*, understood as a set of computer instructions; ii) a software *copy*, which is the embodiment of a set of instructions through a hard medium; iii) a *medium*, the hardware medium itself; iv) a *process*, which is the result of executing the software *copy*.

In their ontology of computer programs [7], Eden and Turner focus on the difference between code and process, which they understand as two main sub-categories of *program*, namely *program-script* and *program-process*. For them, the term 'program' is therefore polysomic: a program-script is a well-formed expression based on a Turing-complete programming language, while a program-process is just the execution of a program-script. A *concretization* relationship links a program-script to the corresponding program-process. Another case of concretization relationship exists for them between a program specification (a kind of *meta-program*) and a program-script. Finally, they tend to consider *software* as synonymous with (and therefore as ambiguous as) *program*, and as posing "unique philosophical questions the observation that programs bridge between the abstract (e.g., Turing automata) and the concrete (desktop computers, microwave ovens)". As we shall see, for us it is exactly such bridging role between the abstract and the concrete that distinguishes software from programs.

Daniel Oberle's Core Software Ontology (CSO [6], but also [14], [15]) attempts to differentiate and formalize the four concepts mentioned above has been done by whose motivations are discussed. Three different senses of 'software' are introduced in CSO, each one specializing a different top-level concept from the DOLCE foundational ontology: *SoftwareAsCode* is an *InformationObject* (a kind of *NonAgentiveSocialObject*); *ComputationalObject* is a kind of *PhysicalEndurant*, and constitutes the physical realization of *SoftwareAsCode* on hardware, but is not the hardware itself; and *ComputationalActivity* (a kind of *Perdurant*) is the result of executing a *ComputationalObject*. This approach builds on the notion of information object proposed by Gangemi and his colleagues in [16].

In [17], Lando and colleagues make similar distinctions, considering a program as *Computer Language Expression*, borrowing therefore the "software as code" idea. Building on the research program by Kroes and Meijers on dual nature of technical artifacts [18], they propose that a program is both a *Computer Language Expression* and an *artifact of Computation.*

A different approach to account for the artifactual nature of software is taken by Irmak [8], also acknowledged by Ray Turner in his recent, comprehensive entry on the philosophy of Computer Science published in the Stanford Encyclopedia of Philosophy [19]. According to Irmak, people have tried to understand software (which he considers as synonymous with program) in terms of algorithm, code, copy and process, but none of these notions can be identified with software, since - due to its artifactual nature - software has different identity criteria. Therefore, a program, which for him is synonymous with software, is different from code. We share very much Irmak's intuitions, as well as the methodology he adopts to motivate his conclusions, based on analysis of the conditions under which a software maintains its identity despite change.

2. Software as an Artifact: from Code to Programs

In the literature, there are a number of entities that are typically conflated with the notion of software. Prominent among them are the notions of *program* and *code*. In the sequel, we argue that these two notions are distinct. In later sections we argue that other notions are required.

Let us start with computer code. We take *computer code* as a well-formed[3] sequence of instructions in a Turing-complete language. Since such instructions are mere sequences of symbols, the identity of code is defined accordingly: two codes are identical if and only if they have exactly the same syntactic structure. So, any syntactic change in a code *c1* results in a different code *c2*. These changes may include variable renaming, order changes in declarative definitions, inclusion and deletion of comments, etc.

A code *implements* an algorithm. Following Irmak [8], we take here an algorithm to mean a pattern of instructions, i.e. an abstract entity, a sort of process universal that is then correlated to a class of possible process executions. So, two different codes *c1* and *c2* can be *semantically equivalent* (e.g., by being able to generate the same class of possible process executions) if they implement the same algorithm, while being different codes. For instance, if *c2* is produced from *c1* by variable renaming, inclusion of comments and modularization, *c2* can possess a number of properties (e.g., in terms of understandability, maintainability, aesthetics) that are lacking in *c1*.

As we have seen, there are proposals [17] that identify the notions of program and computer code, while others [7], [6] distinguish program-script (a program code) from program-process (whose abstraction is an algorithm). However, we agree with Irmak that we cannot identify a program either with a code, a process, or an algorithm. The reason is that this view conflicts with common sense, since the same program usually consists of different codes at different times, as a result of updates[4]. What these different codes have in common is that they are selected as constituents of a program that is intended to implement the same algorithm. To account for this intuition, we need a

[3] So, we do not consider so called 'ill-formed code' as code, but just as text.

[4] Irmak also admits that the same program may have different algorithms at different times.

notion of (technical) artifact. We are aware that many such notions have been discussed in the literature, but the one by Baker [20] works well for us: *"Artifacts are objects intentionally made to serve a given purpose"; "Artifacts have proper functions that they are (intentionally) designed and produced to perform (whether they perform their proper functions or not)"; "What distinguishes artifactual [kinds] from other [kinds] is that an artifactual [kind] entails a proper function, where a proper function is a purpose or use intended by a producer. Thus, an artifact has its proper function essentially"*. These passages are illuminating in several respects. Firstly, Baker makes clear that artifacts are the results of intentional processes, which, in turn, are motivated by intentions (mental states) of (agentive) creators. Moreover, she connects the identity of an artifact to its *proper function*, i.e., what the artifact is intended to perform. Finally, Baker recognizes that the relation between an artifact and its proper function exists even if the artifact does not perform its proper function. In other words, the connection is established by the intentional act through which the artifact is created.

In the light of these observations, a code is not necessarily an artifact. If we accidentally delete a line of code, the result might still be a computer code. It will not, however, be "intentionally made to serve a given purpose". Moreover, we can clearly conceive the possibility of codes generated randomly or by chance (for instance, suppose that, by mistake, two code files are accidentally merged into one). In contrast, a program is *necessarily* an artifact. A computer program is created with the purpose of playing a particular *proper function.* But, what kind of function? Well, of course the *ultimate* function of a program is –typically– that of producing useful effects for the prospective users of a computer system or a computer-driven machine, but there is an *immediate* function which belongs to the very essence of a program: producing a desired behavior, when the program is executed, *inside* a computer endowed with a given *programming environment* (such as an operating system). We insist on the fact that such behavior is first of all *inside* the computer, as it concerns phenomena affecting the *internal* states of its I/O ports and memory structures, not the *external* states of its I/O devices. Examples of such behaviors can be changes inside a file or a data structure, resulting from the application of certain algorithm. In summary, a program has the essential property of being intended to play an internal function inside a computer. Such function can be specified by a *program specification* consisting of a data structure and the desired changes within such data structure[5]. For every program we assume the existence of a unique *specification* of such expected behavior, called *program specification*. In order for a program to exist, this specification must exist, even if only in the programmer's mind.

Since code and program differ in their essential properties (programs are necessarily artifacts and possess essential proper functions; codes are not necessarily artifacts), we have to conclude that a program is not identical to a code. However, if program and code are different entities, what is the relation between the two? In general, the relation between an artifact and its material substrata is taken to be one of *constitution*. As noted in [20], the basic idea of constitution is that whenever a certain aggregate of things of a given kind is in certain circumstances, a new entity of a different kind comes into being.

So, when a code is in the circumstances that somebody, with a kind of *act of baptism,* intends to produce certain effects on a computer, then a new entity emerges, con-

[5] Such specification covers the *functional* aspects of a program. A full specification may also include non-functional aspects, such as time and security constraints.

stituted by the code: a *computer program*. If the code does not actually produce such effects, it is the program that is faulty, not the code.

Consider now a program, constituted by a certain code. Let us observe first that this is not a physical object, as it lacks a location in space. So, in pace with Irmak, we take a program to be a particular kind of *abstract artifact*. On one hand, it behaves like a type (or universal) since it defines patterns that are repeatable in a number of copies. On the other hand, unlike a type, a program is not outside space and time. A program does not exist eternally like other abstract entities such as numbers and set; in particular, a program does not pre-date its creator. As previously mentioned, it is in fact historically dependent on an intentional "act of baptism" and, hence, on its creator. In addition to such historical dependence, we shall assume that a program constantly depends on some substratum in order to exist, in the sense that at least a physical encoding (a *copy*) of its code needs to exist. Finally, we shall also assume that, whenever a program exists, its underlying intention to implement the program specification is recognizable by somebody (possibly thanks to suitable annotations in the code). So, a program *p* is present at *t* whenever: i) a particular code *c* is present at *t* (which means that at least a copy of *c* exists at *t*); ii) a program specification *s* exists at *t*; and iii) at *t,* there is somebody who recognizes *c* as intended to implement *s*, or there is an explicit description of this intention (e.g., via a documentation in the code or in an explicitly described program specification) which is recognizable by someone.

In conclusion, a syntactic structure could be used as an identity criterion of a code, and a program specification along with the intentional creation act could be used as the identity criteria of a program. As we have seen, one of the interesting aspects distinguishing program from code is the possibility to honor the commonsense idea shared among software engineering practitioners that a program can change its code without altering its identity**.**

3. From Programs to Software Systems

As we have seen, the identity criteria of programs are bound to the *internal* behavior of a computer. On the other hand, software is usually intended as an artifact whose ultimate purpose is constraining the behavior of an environment *external* to the computer, which the computer monitors and controls by means of *transducers* bridging between symbolic data and physical properties. In the case of a *stand-alone computer* such transducers just concern the human-computer interface and the standard I/O devices; for *mobile* systems they may also include position and acceleration sensors, while in the case of *embedded systems* they take the form of ad-hoc physical sensors and actuators. So, in the general case, the software's ultimate purpose is achieved by running a code that produces certain effects inside a computer, which drives a physical machine, which in turn produces certain effects on its external environment.

In software engineering, the desired effects the software is intended to have on the environment are called *requirements*. The role of the sub-discipline of software engineering called *requirements engineering* is to elicit, make explicit and analyze these requirements in order to produce a *specification* that describes the behavior of a (computer-based) machine. We assume that a software specification is a functional specification, as defined in standards such as IEEE-STD-830-1993. From an ontological point of view, functions are existentially dependent entities that are intimately related to the ontological notion of disposition (capacity, capability) exhibited by an object. Func-

tions and dispositions are potential (realizable) properties such that when a situation (state of the world) of a particular kind obtains they are manifested through the occurrence of an *event*, determining in this way the object's behavior.

This view has been described in several papers by Jackson and Zave [9], [21], [10], which draw a clear distinction between the environment, which is where the ultimate effects of software are expected, and the machine, the computer-driven system where software operates. Their goal is to show that the intentional relationship between the two can be defined by establishing a logical connection between the intended external behavior of the machine (described by a *specification S*), the relevant assumptions about environmental properties (described by a body of *world knowledge W*), and the desired environmental behavior (described by a set of *requirements R*). Such connection is captured by the following condition that the specification must satisfy in order to fulfill its requirements: if the machine behaves according to the specification, and the world assumptions hold, then the required effects in the environment occur. In a compact form, Jackson et al. describe this condition as follows: $S \wedge W \vDash R$. We say in this case that *S satisfies R* under assumptions *W*.

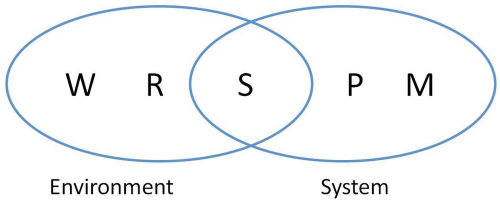

Figure 1. A model of software requirements engineering (from [11]).

In their earlier papers, Jackson et al. were somewhat vague – perhaps deliberately – on the exact meaning of what they were calling 'machine', and therefore on the exact boundary between the machine and the environment. In particular, in [9]and [21] the machine is a *computer* (and the transducers are part of the environment), while in [10] the machine is a *computer-based machine*, which seems to incorporate the transducers. The picture gets clearer in a more recent paper [11], where the machine *M* is constituted by a generic *programmable*[6] *platform* which includes the transducers that connect it to the environment as well as a programming environment (i.e., an operating system), and whose behavior can be controlled by means of a *program P*.

This view is depicted in Figure 1. In our understanding, the two ovals represent the possible phenomena involving the environment (to the left) and a programmable platform connected to the environment, called the machine (to the right). The intersection represents phenomena at the interface between the machine and the environment. The letters mark specific subsets of relevant phenomena: those which are required (*R)*, those which are not required but nevertheless play a relevant causal effect in the environment (*W*), those which concern the desired behavior of the machine at its interface with the environment (*S*), those generated by the program inside the computer that drives the machine (*P*), and those occurring internally to the machine (*M*). This view, considered nowadays as a fundamental model for *requirements* engineering, emphasizes the role of the specification of machine's behavior at its interface with the environ-

[6] In the original paper, the term used is 'programming platform', but we believe that 'programmable platform' is more perspicuous.

ment. From the perspective of *software* engineering, however, we are interested not in the machine as such, but in the program which drives it, and ultimately in the relationship between the program and the high level requirements. As [11] observe, such a relationship can be obtained as a composition of two relationships:

If *(i) S properly takes W into account in saying what is needed to obtain R, and*
(ii) P is an implementation of S for M, then
(iii) P implements R as desired.

In conclusion, while in the previous section we focused on the *immediate* function of programs as technical artefacts –producing some effects inside a computer– here the Jackson and Zave's model allows us to understand how programs play their *ultimate* function, which is producing certain effects in the external environment (i.e., satisfying the high level requirements).

Such function is realized in two steps: first, the internal computer behavior resulting from running the program generates some physical effects in the environment at the interface with the machine (i.e., the programmable platform, including the I/O transducers). For instance, a message appears on the screen. Second, under suitable assumptions concerning the environment (for instance, there are people able to perceive and understand the message), the ultimate effects of the program are produced (e.g., the person who reads the message performs a certain action).

The presence of these two steps in realizing the ultimate function of a program suggests us to introduce two further artifacts, a *software system*, whose essential property is being intended to determine a desired external behavior at the machine interface and a *software product*, whose essential property is being intended to determine some desired effects in the environment *by means* of the machine behavior at the interface, given certain domain assumptions.

So, the Jackson and Zave's model can be replicated at three different levels, each corresponding, in our proposal, to a different artifact, based on the different reasons for why a certain piece of code is written. In summary, given (*a*) a set *R* of high level requirements concerning a certain environment, and independent of the presence of any machine, and (*b*) a programmable platform (called 'the machine') constituted by a computer *C* running a specific operating system and endowed with suitable transducers in order to monitor and control the environment:

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

4. From Software Systems to Software Products

Let us now focus on the notion of *software product* previously introduced. While the essential function of a software system is to control the external behavior of a certain machine (i.e., according to the Jackson et al.'s approach, that part of the behavior that is "visible" to both the environment and machine), the essential function of a software product is to control the environment's behavior which is not visible to the machine, but can be influenced by it, under given environment (domain) assumptions, as a result of the interaction with the environment.

It is important to note that a software product is intended to achieve some effects in the external environment *by means of a given machine*, and *under given environment assumptions*. So, assuming they have exactly the same high-level requirements, MS Word for Mac and MS Word for PC are different software products (belonging to the same *product family*), since they are intended for different kinds of machines. Similarly, country-oriented customizations of Word for Mac may be understood as different products, since they presuppose different language skills, unless the requirements already explicitly include the possibility to interact with the system in multiple different languages.

Consider now one such software product, say MS Word for Mac. Starting with version V1, which denotes the specific software system constituting the software product at the time of its first release, this product will suffer a number of possible changes in its constituents in order to fix bugs, to include new functionalities, to improve performance, security, precision, etc. Each of these changes leads to distinct code, but some of them (those that are not just bug fixings) will also lead to a new program, while others (those that concern changes in the external interface) will also lead to a distinct software system, namely to a different version (V2) of *the same product*.

To reflect these changes, *the code* will be marked in order to distinguish itself from the former codes, and in order to identify the program, the software system, and the software product. According to the usual conventions, the software system could be identified by the *version number,* the program by the *release number,* and the code by the *sub-release number*. In this way, we see how an ontology of software artifacts based on the Jackson et al.'s architecture can produce a version numbering system which reflects the *product change rationale*.

In summary, the core ontological distinctions induced by the different requirements engineering perspectives we have discussed are illustrated in Figure 2. To the left, we see different software artifacts all ultimately constituted by some code (which is a specific syntactic expression). They have different essential properties, resulting from the fact that each of them is constantly dependent on a different intentional entity. Each of these entities refers to an expected behavior involving different parts of a complex socio-technical system, which in turn emerges from the interaction of a computer-driven machine and a social environment.

A brief account of the main relations appearing in the picture is reported below. As usual, the subsumption relation is represented by an open-headed arrow. The closed-headed arrows represent some of the basic relations discussed in the paper. For some of them (constitution and specific constant dependence), the intended semantics is rather standard, while for others we just sketch their intended meaning, postponing a formal characterization to a future paper.

constitutedBy: We mean here the relation described extensively by Baker [20]. We just assume it being a kind of generic dependence relation that is both asymmetric and non-reflexive, and does not imply parthood. We can borrow a minimal axiomatization from the DOLCE ontology.

specificallyConstantlyDependsOn: If *x* is specifically constantly depending on *y*, then, necessarily, at each time *x* is present also *y* must be present. Again, we can borrow the DOLCE axiomatization. When this relation holds, being dependent on *y* is for *x* an essential property.

intendedToImplement: This relation links an artifact to its specification, as a result of an intentional act. Note that the intention to implement does not imply that the implementation will be the correct one (e.g., bugs may exist).

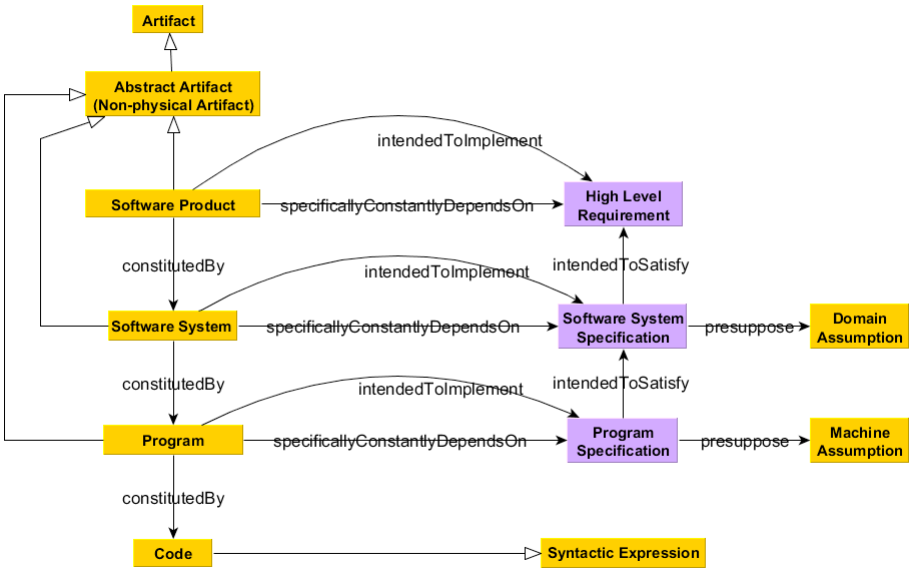

Figure 2. Different abstract software artifacts induced by different requirements engineering notions.

intendedToSatisfy and **presuppose.** These two relations are proposed to capture the structure of the formula proposed by Jackson et al. to describe the nature of requirements engineering, $S \wedge W \vDash R$. *S*, which presupposes *W*, is intended to satisfy *R*. Presupposition is a kind of historical dependence on certain knowledge states.

5. From Software Products to Licensed Software Products

As we have seen, the different kinds of software artifacts we have discussed are based on a requirements engineering perspective. We can't ignore however another perspective that deeply affects the current practice of software engineering, namely the *marketing perspective*. In the present software market, software products don't come alone, since what companies sell are not just software products: in the vast majority of cases,

a purchase contract for a software product includes a number of rights and duties on both parties, including the right to download updates for a certain period of time, the prohibition to give copies away, the right to hold the clients' personal data and to automatically charge them for specific financial transactions, and so on. Indeed, the very same software product can be sold at different prices by different companies, under different *licensing policies*. The result is that software products come to market in the form of *service offerings*, which concern *product-service bundles.* According to [22], a *service offering* is in turn based on the notion of *service*, which is a bundle of social commitments that under certain circumstances can trigger the execution of certain actions by a service provider. Service offerings are therefore meta-commitments, i.e., they are commitments to engage in specific commitments (namely, the delivery of certain services) once a contract is signed. Notice that such services may not only concern the proper functioning of software (involving the right to updates), but also the availability of certain resources in the environment where the software is supposed to operate, such as remote servers (used, e.g., for Web searching, VOIP communication, cloud syncing). So, when Skype Inc. releases Skype, it publicly commits to engage in such kind of commitments. By the way, this means that, when buying Skype from Skype Inc., Microsoft is not only buying the software product, but it is also buying all the rights Skype Inc. has regarding its clients. On the other hand, Microsoft is also obliged to honor the commitments that the *legal owner of Skype* has to the clients of that software product.

Now, on the basis of these observations, we believe there is another kind of software artifact emerging, which has a strong social nature: a *Licensed Software Product.* Such an artifact, which is constituted by a software product, comes into existence when a software product is bought, and it historically depends on a pre-existing service offering concerning a product-service bundle with a certain licensing scheme. So, while, at a given time, there is only one software product named MS Word for PC, there are many licensed products (one for each customer), each one in turn being encoded in several licensed copies. The essential property of a licensed software product is the mutual commitment relationship existing between the vendor and the customer.

6. Conclusions and Future Work

In this paper we presented a first attempt to analyze the ontological nature of software artifacts in the light of the Jackson and Zave's model, considered nowadays as a foundation for requirements engineering. Such a model has helped us to provide an answer to the question concerning the identity criteria of software artifacts raised by Irmak: there three different kinds of software artifacts, exhibiting different essential properties depending on people's intentions to produce effects in different parts of complex computer-driven sociotechnical systems. In addition, there is a fourth kind of artifact reflecting the social nature of software products, whose essential properties are based on the mutual commitments between vendors and customers. Such different essential properties are summarized in Table 1.

Besides contributing to clarify concepts and terminologies in the software engineering community, our work could also be used as a foundation for software evolution. For instance, on the basis of our analysis, a refined terminology for different kinds of software change may be proposed: *Refactoring* refers to the creation of new codes, keeping the identity of the program; *re-engineering* refers to the creation of new pro-

grams, keeping the identity of the software system; *software evolution* refers to the creation of new software systems, keeping the identity of the software product.

Table 1. Essential properties of software artifacts

Object	Essential Properties
Licensed Software Product	Mutual vendor-customer commitments
Software Product	High Level Requirements
Software System	Specification of external machine behavior
Program	Specification of computer behavior

By identifying the rationale for these changes in the different abstraction layers, our work contributes to establish a rigorous foundation for software versioning. Traditional version codes are usually decided on the basis of the significance of changes between releases, but the decisions of the significances are entirely arbitrary and up to the author. On the basis of our approach, versioning numbers can be established in a rigorous standard way (e.g. v 1.2.3: 1 - software system specification number; 2 - program specification number, 3 - code number).

In future works, we plan to provide a formalized version of the current work with detailed axioms. Following that, an ontology of software evolution will be developed, focusing on different kinds of software changes. We hope that our work could be used as a foundation for the relating domains, including software maintenance, software project management, software measurements and metrics, and others.

Acknowledgements. Support for this work was provided by the ERC advanced grant 267856 for the project entitled "Lucretius: Foundations for Software Evolution" (http://www.lucretius.eu), as well as the "Science Without Borders" project on "Ontological Foundations of Service Systems" funded by the Brazilian government.

References

[1] E. B. Swanson, "The dimensions of maintenance," in *Proceedings of the 2nd international conference on Software engineering*, 1976, pp. 492–497.

[2] N. Chapin, J. E. Hale, K. M. Kham, J. F. Ramil, and W.-G. Tan, "Types of software evolution and software maintenance," *J. Softw. Maint.*, vol. 13, no. 1, pp. 3–30, 2001.

[3] J. Buckley, T. Mens, M. Zenger, A. Rashid, and G. Kniesel, "Towards a taxonomy of software change: Research Articles," *J. Softw. Maint. Evol.*, vol. 17, no. 5, pp. 309–332, 2005.

[4] J. H. Moor, "Three myths of computer science," *Br. J. Philos. Sci.*, vol. 29, pp. 213–222, 1978.

[5] T. R. Colburn, *Philosophy and Computer Science*. M.E. Sharpe, 2000.

[6] D. Oberle, *Semantic Management of Middleware*, vol. 1. New York: Springer, 2006, p. 268.

[7] A. H. Eden and R. Turner, "Problems in the ontology of computer programs," *Appl. Ontol.*, vol. 2, no. 1, pp. 13–36, 2007.

[8] N. Irmak, "Software is an Abstract Artifact," *Grazer Philos. Stud.*, vol. 86, no. 1, pp. 55–72, 2013.

[9] M. Jackson and P. Zave, "Deriving specifications from requirements: an example," in *Proceedings of the 17th international conference on Software engineering*, 1995, pp. 15–24.

[10] P. Zave and M. Jackson, "Four dark corners of requirements engineering," *ACM Trans. Softw. Eng. Methodol.*, vol. 6, no. 1, pp. 1–30, 1997.

[11] C. A. Gunter, M. Jackson, and P. Zave, "A reference model for requirements and specifications," *Software, IEEE*, vol. 17, pp. 37–43, 2000.

[12] L. J. Osterweil, "What is software?," *Autom. Softw. Eng.*, vol. 15, no. 3–4, pp. 261–273, 2008.

[13] P. Suber, “What is software?,” *J. Specul. Philos.*, vol. 2, no. 2, pp. 89–119, 1988.

[14] D. Oberle, S. Lamparter, S. Grimm, D. Vrandečić, S. Staab, and A. Gangemi, “Towards ontologies for formalizing modularization and communication in large software systems,” *Appl. Ontol.*, vol. 1, no. 2, pp. 163–202, Jan. 2006.

[15] D. Oberle, S. Grimm, and S. Staab, “An Ontology for Software,” S. Staab and D. Rudi Studer, Eds. Springer Berlin Heidelberg, 2009, pp. 383–402.

[16] A. Gangemi, S. Borgo, C. Catenacci, and J. Lehmann, “Task Taxonomies for Knowledge Content D07.” Metokis Project, 2004.

[17] P. Lando, A. Lapujade, G. Kassel, and F. Fürst, “An Ontological Investigation in the Field of Computer Programs,” in *Software and Data Technologies SE - 28*, vol. 22, J. Filipe, B. Shishkov, M. Helfert, and L. Maciaszek, Eds. Springer Berlin Heidelberg, 2009, pp. 371–383.

[18] P. Kroes and A. Meijers, “The Dual Nature of Technical Artifacts-presentation of a new research programme,” *Techné Res. Philos. Technol.*, vol. 6, no. 2, pp. 4–8, 2002.

[19] R. Turner, “The Philosophy of Computer Science,” in *The Stanford Encyclopedia of Philosophy*, Fall 2013., E. N. Zalta, Ed. 2013.

[20] L. R. Baker, “The ontology of artifacts,” *Philos. Explor.*, vol. 7, no. 2, pp. 99–111, Jun. 2004.

[21] M. Jackson, “Specialising in Software Engineering,” in *Software Engineering Conference, 2007. APSEC 2007. 14th Asia-Pacific*, 2007, pp. 3–10.

[22] J. C. Nardi, R. De Almeida Falbo, J. P. A. Almeida, G. Guizzardi, L. Ferreira Pires, M. J. van Sinderen, and N. Guarino, “Towards a Commitment-Based Reference Ontology for Services,” in *Enterprise Distributed Object Computing Conference (EDOC), 2013 17th IEEE International*, 2013, pp. 175–184.