Formal Ontology in Information Systems
B. Brodaric and F. Neuhaus (Eds.)

doi:10.3233/FAIA200673

The Computer Program as a Functional Whole

C. Maria KEET [a,1]
[a] *Department of Computer Science, University of Cape Town, South Africa*

Abstract. Sharing, downloading, and reusing software is common-place, some of which is carried out legally with open source software. When it is not legal, it is unclear how many infringements have taken place: does an infringement count for the artefact as a whole or for each source file of a computer program? To answer this question, it must first be established whether a computer program should be considered as an integral whole, a collection, or a mere set of distinct files, and why. We argue that a program is a functional whole, availing of, and combining, arguments from mereology, granularity, modularity, unity, and function to substantiate the claim. The argumentation and answer contributes to the ontology of software artefacts, may assist industry in litigation cases, and demonstrates that the notion of unifying relation is operationalisable.

Keywords. Computer program, mereology, unity, granularity

1. Introduction

End users of Information Technology are familiar with software 'apps', or computer programs, that can be launched through a one- or double- mouse click or finger-tap action. From this user experience perspective, the computer program may appear to be one single entity. While it is not impossible that there is only one single file[2], with the increasingly complex programs nowadays, as well as modularisation practices over at least the past 25 years, there are typically multiple files involved in the running of an application for it to carry out its principal function. The components are stored in some specific directory and are artefacts such as plugins, icons for the interfaces, and configuration files that are typically flat text files or XML files. Disregarding those finer-grained details, one still may consider, e.g., TexShop or Firefox, or their respective source code, a single artefact that is downloaded and installed. Why so, or why might it not be a whole upon closer inspection? Why should one even spend time on answering this question?

There may be many practical and financial consequences following from the answer to such questions. This includes having been fought in court and arbitration tribunals, especially concerning copyright infringements, trade secret violations, and patent infringements. Recent and ongoing cases include, among others, Google vs Oracle on possible

[1]Corresponding Author: C. Maria Keet, Department of Computer Science, University of Cape Town, South Africa; E-mail: mkeet@cs.uct.ac.za

[2]though very rare: even code copied in the command line to print "Hello World" requires the built-in `print` function that is stored in another file than the hand-written code

copyright on APIs [1], Cisco vs Arista on copying parts of a computer network management utility tool where Arista eventually payed Cisco $400 million [2], and copyright claims on an enduser app in the health insurance domain [3]. A software litigation case that the author was called in as expert witness concerned, among others, a question essentially of an ontological nature. This is introduced here in a general way and with broader applicability. One party claims that for the computer program, there are the, say, $n = 1000$, individual source files that allegedly have been copied and wants the defendant to be fined for the 1000 copyright violations, once for each source file; hence, not, say, a €100 fine for the one infringement of the program, but a fine of €100000 altogether. The defendant, for obvious reasons, would rather prefer to pay just the €100 for one infringement, if that were to be deemed to apply according to the court. The argument may likewise be constructed for trade secret violations for stealing the intellectual property of an app, for pirated software that was illegally downloaded, and downloading textbooks in website format (which have multiple files cf. the single pdf option). It might be relevant also regarding the number of patent infringements, but this depends on the patent and the country; e.g., in South Africa, one cannot patent computer programs or algorithms.

On the surface, alleged infringements may sound similar to the well-known case of whether the collapsing bridge is one event or multiple collapsing events; however, here it first needs to be established how all those files of a computer program relate, which is not as straightforward as the structural components of a bridge. The main question to answer, then is:

Q1: Is a computer program or the source code a) a (tight) *whole* with *parts*, b) a whole that is a *collection* of *artefacts*, or c) just a *set* of *artefacts* where each element is a separate self-standing item?

This question generates two more specific ones to answer regarding the parts, in order to be able to answer Q1:

Q2: What is the relation between the files of a computer program (resp. source code) and the computer program (resp. source code)?
Q3: What is the relation among the files of a computer program (resp. source code)?

While it may be obvious intuitively to some that a program is a functional whole that has unity, that it generated a litigation case suggests that that view is not held throughout. There is ample documentation of 'just so' compositionality of programs—i.e., implying parthood relations—and why it is essential to good software design practices (e.g., [4, 5,6]) and to system design more generally [7]. Yet, to the best of our knowledge, there is no extant argumentation regarding the wholeness of a computer program or what it is that makes it a whole consisting of parts, why, and how from an ontological perspective. The more common question that is investigated in philosophy is the nature of a computer program, like whether it is a process or of some other category [8] or when an artefact can be classified as being a computer program, which are considerations *at that level of granularity* of the artefact, not about its compositional nature. Such arguments do, however, strengthen the argument that a computer program is a whole.

To answer the questions, we take insights from part-whole relations and mereology, the notion of unity, and basics of granularity, to combine them with modes of participation and functional parthood. Computer programs have a main function that is carried out by genuine functional parts and those parts have a specific unifying relation among each

other at that finer-grained level of granularity, where one implies the other, therewith making a computer program a functional whole.

In the remainder of the paper, we first introduce some preliminaries in Section 2. We then consider the 'vertical' relation between the whole and its components in Section 3, the 'horizontal' relation between the components in Section 4, and close the argument that software is a proper (complex) whole in Section 5. We conclude in Section 6.

2. Preliminaries

There are multiple aspects one can investigate about computer programs and which mode is considered in such an analysis, such as the typed up source code files, the interpreted or compiled (machine) code that is the executable, or when it is actually running as a process on the computing device. This includes, among others, their ontological nature (whether it is, e.g., a process, a disposition, or an endurant), the nature and multiplication aspects of the 'information object' aspect of a program, identity criteria, and definitions. For instance, the Copyright Act 98 of 1978 of South Africa[3] defines "computer program" as "a set of instructions fixed or stored in any manner and which, when used directly or indirectly in a computer, directs its operation to bring about a result.". With limited space, it is possible to zoom in on only one aspect of computer programs. Specifically, it first needs to be established that the computer program is a whole, and why, regardless of what category that whole is of, if it is a whole. This is needed to inform debates about identity, determine the boundary, and it may assist determining the ontological nature of either mode of the program. Topics that are of relevance within this specific scope are:

– Part-whole relations, and in particular: i) Mereological parthood, and possibly any of its refinements; ii) Mandatory, immutable, and essential parthood; and iii) Collective entities (collections) and sets.
– Identity and unity.
– Granularity.

We introduce each one briefly informally for the reader unfamiliar with them, so as to keep the paper self-contained. The respective formal characterisations can be found in the literature cited. For the most part, those details are not needed for the argumentation on software wholes, unless stated otherwise.

Parts, wholes, and part-whole relations. Part-whole relations have been used and investigated in several fields of study, notably philosophy, ontologies for information systems, conceptual models, linguistics, and NLP (see [9,10,11] and references therein). This has resulted in a fairly stable list of common part-whole relations, including mereological parthood, refinements thereof, and informal ones in natural language utterances only but not mereologically [9]. The two key principles to organise them are transitivity (or not) of the relation and the category of entities that participate in the relation. Regarding the former, e.g.: if a cell is part of a heart and that heart is part of a human, then that cell is part of that human. An entity may thus play the role of 'part' in one relation and the role of 'whole' in another one. Regarding the latter, one could name the parthood that relates only processes to its part-processes as, e.g., *involvement* (*i*), specialising mereological parthood (*p*) that does not have any constraint on its relata,

[3] inserted through 1 (g) of Act 125 of 1992 (available at `https://www.gov.za/sites/default/files/gcis_document/201409/act125of1992.pdf`; last accessed: 16 July 2020)

i.e., $\forall xy(i(x,y) \rightarrow p(x,y) \wedge process(x) \wedge process(y))$. With an natural language utterance 'each Eating event *involves* a Swallowing event', one then identifies *involves* as a surface realisation of the *involvement* relation, and thus eating has as part swallowing.

Collectives fall within the scope of parts and wholes and part-whole relations. For instance, Project Team may be considered to be a collective consisting of human beings and one then may communicate that, say, "Dr. X is part of the project team". This is better characterised as a *membership* or *grain* relation between the part or grain Human and the collective Project Team rather than parthood, because transitivity cannot be guaranteed [9]. The nature of a collective entity has been investigated in fields such as conceptual modelling [12], biomedicine [13], and social theory [14], which may endow collectives with features or constraints on the whole or on its constituent parts. Widely-agreed upon key characteristics are that there is an identity to the collective as a whole, with some particular meaning, and it may survive its members, i.e., a member may change or be swapped but the whole keeps its identity. For instance, the rock band Queen has had three bass players before John Deacon, yet was already, and remained, Queen. An example of an additional constraint on the collective is that they can and do perform actions [14] and an additional constraint on a collective's parts may be that those parts must all perform the same role and if they do not, then the entity is more complex than a mere collective [12]. Collective nouns in natural language, such as a fleet and flock thus may, or may not, be considered collectives ontologically, depending on such additional constraints.

Collectives stand in contrast to mere sets, where a change in member changes the identity of the set, its members do not need a common 'binding' feature, and therewith a set does not need to have a specific meaning, and sets have no agency. For instance, your left foot and my laptop is a set with two objects as its members, but it is meaningless ontologically and does not instigate anything.

Orthogonal to the notion of types of part-whole relations is how the part and the whole participate in the relation, which might also be referred to as the 'strength' of the participation of the part and the whole. There are four principal options [15]: *i) optional*: the part P may, or may not, be part of the whole W, or the W may or may not have that P, e.g., a camera that may be part of a car; *ii) mandatory*: some P must be part of W, or vv., but not necessarily that one, e.g., for a house to be a house, it has to have a roof as part, but the roof can be renewed, while still being the same house; *iii) immutable*: that particular P must be part of W (or vv.), for as long as W is in a particular role, e.g., a boxer must have as part his own hands and if he loses a hand he ceases to be a boxer but as human they may continue to live; *iv) essential*: that particular P must be part of W and if it loses that P, the W will cease to be W, e.g., how a particular brain is part of a human.

There are further notions of sharability of the part, which is closely linked with the notions of participation: *i) sharable concurrently*: some P may be part of more than one W at the same time, e.g., a talk may be part of a course and of a seminar series; *ii) sharable sequentially*: P is part of one W at a time but can be part of different Ws at different times, e.g., a removable network card may be part of one PC at one time and part of another one another time; *iii) exclusivity*: P can be part of at most one W, e.g., entities such as the brain, spinal cord, or thalamus as part of a human body.

Identity and unity. Ascertaining the identity of an entity is about examining something as being the same or different at one point in time (synchronic identity) or across time (diachronic identity). To establish the identity of an entity, one uses *identity criteria* that the entity has, which is a stronger notion than *identification criteria* (typically a

single artificial attribute) that may be assigned to entities. For instance, one may identify *Tibbles*, the neighbour's cat, by the colour of its fur, its size, and how it meows, whereas artificial attributes of entities, such as one's ID number and the MAC address of a network card, assists with identification. Within the context of programs, it may be of use to also note "striking resemblance" or "comprehensive non-literal similarity" of program code[4], which holds when two pieces of code are identifiably different, but *the semantics/logic of the expression of the idea* is the same. The two typical examples are 1) to take some piece of code and change the names of the variables: they are different documents, but have striking resemblance; 2) to keep all the business logic of the code but translate it line by line from one programming language into another language (say, Java to C++).

Unity [16,17] is a special case contributing to identity, which focuses on the relation that the parts have that make up the whole entity that has some identity. *Unity criteria* are those criteria that have to hold among the parts for it to be a whole. Depending on the nature of the unifying relation, one can identify different types of wholes. Some typical examples are *topological wholes*, such as a tree and a heap of sand, *morphological wholes*, such as a bouquet of flowers and a constellation of stars, *functional wholes*, such as scissors and a bikini, and *social wholes*, which are certain types of collectives, such as a population of a country and a herd of sheep. There is 'something' that binds the parts together to be a meaningful whole with an identity, which stands in contrast to, notably, stuffs (typically indicated with mass nouns, such as water and wood) and arbitrary sets of entities like the set of your left foot and my laptop. Since we only deal with computer programs, we exclude stuffs or amount of matter from the possible relata henceforth.

Guarino and Welty sought to formalise the notion of unity, with a good start made in [16]: with B the generic unifying relation that binds the parts with respect to the whole, and P mereological parthood that is temporally indexed (presumably with time $t \in \mathscr{T}$ as discrete time), their first condition is that

$$\forall xyzt(P(x,y,t) \rightarrow (P(z,y,t) \leftrightarrow B(x,z,t))) \tag{1}$$

Informally: if there is a parthood between x and y as some time (t), then at the same time it holds that for any entity z, that is also part of y at that time, it is in a unifying relation with the part x, and vice versa.

Eq. 1 and the description in [16] assume weak supplementation would apply, under the assumption that $x \neq z$, which they specified as $\forall xy(PP(x,y) \rightarrow \exists z(PP(z,y) \wedge \neg O(z,x)))$ (every proper part PP must be supplemented by another, disjoint [not overlapping $\neg O$], part), because of the implication: there must be another part of y, being z, so that they can relate through B. Guarino and Welty want to exclude any mereological sum that is just a contingent whole. This can be seen in analogy to the difference between mere sets and collectives. Noting a typo in their formalisation in [16] (corrected in their lecture slides on OntoClean), and adding quantification, we formalise it as

$$\neg\forall xzt(B(x,z,t) \leftrightarrow \exists y(P(x,y,t) \wedge P(z,y,t))) \tag{2}$$

That is, it does not hold for all x and z that when they are related through B at time t they are also a part of y, or: x and z may be bound in some way, but they are not therefore also part of a whole y because of that. This raises a few questions about the

[4]The famous cases where this got established are Franklin vs. Apple (1984) and Whelan vs. Jaslow (1987).

exact configuration when Eqs. 1 and 2 are taken together. For instance, can or do they have to be different Bs that relate x and z? Can they be part of different wholes, or: are the ys in Eqs. 1 and 2 supposed to refer to the same whole (which they currently do not)? Weak supplementation aims to rule out the possibility that $x = z$, in that there cannot be a whole with a single *proper* part, but unity is declared with P, not PP, so would unity still apply just in case there is only one part? These finer-grained details about the configuration brings us to the last preliminary: granularity.

Granularity. This refers to the level at which one operates or analyses and that there are different levels [18]. For instance, at level L_i, some amount of water is a type of stuff, but at its finer-grained level L_{i-1}, there is a set or collection of H_2O molecules, or at L_i one has an instance of Team and at L_{i-1} are its grains, Human Beings. The level of analysis may be of importance ontologically. For instance, Organisation is a social entity at level L_i that has one or more instances of Human as members at L_{i-1}, but it is only the former that must have a bearer with a single physical address that holds for that entity at that level of granularity, and that attribute is not inherited by its constituent parts, since, e.g., the company's employees typically live elsewhere. Similarly, it is the program that has, at that level of granularity L_i, the property of directing the computer's operation to bring about a result. That is, the whole entity at L_i has properties of its own, which can be very different from its parts or grains residing at L_{i-1}.

Note that granularity focuses on the more/less details of analysis, which is different from prioritisation at the same level of granularity. For instance, there may be direct parts x, y, and z at level L_{i-1} of whole w at L_i and one is interested only in x and y. This does not push z into a finer-grained level L_{i-2}, since an ontologist's interest is a separate matter.

3. Software components and their relation to the source code and program

We first discuss some important features of computer program as entity, with its parts, construction and use, and how those files relate to the source code or machine code. This will be illustrated with a relevant program that the author contributed to and whose principal components are shown in Fig. 1: it verbalises an ontology in isiZulu [19]. Also, henceforth, a distinction is made between the *source code* as artefact and the *compiled code* as artefact, since multiple source code files may compile into one compiled artefact that is a digital file. For compiled code, one also can include interpreted code, since the arguments also hold for it as it is also executable machine code. x source code files may result in y (with $x \neq y$) files of the software application, and the application has one 'point of entry' to launch it irrespective of the number of files it needs for its proper operation.

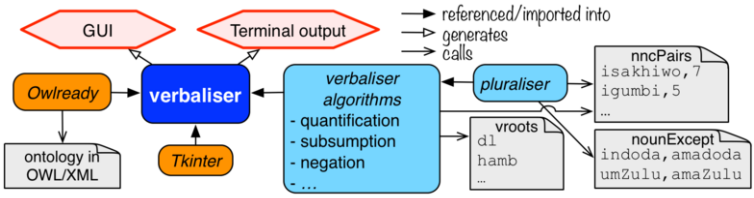

Figure 1. Principal components of the OWL verbaliser for isiZulu (source: based on [19]).

3.1. Modularisation and compositionality

Good design principles entail *modularisation* and *compositionality*—be it for software engineering or other engineering (e.g., [4,5,6,7]). Informally: this means breaking up a large problem into smaller ones, to devise solutions for the smaller problems and to put those together—declaring relations or rules-as-relations to specify how the components interact—to solve the larger problem. Benefits of this approach include, among others, manageability, reusability, quality control, and maintainability. For instance, with the isiZulu verbaliser (Fig. 1), the Python module Owlready is reused, and the vocabulary is easy to maintain by just updating a text file. A mereology-based approach joined with category theory for multiple types of part-whole relations has been applied to component-based software architectures [20], but not at the level of code. The principle can straightforwardly be extended by assuming that each component of the architecture is realised by one file. Practically, each component may be divided up into smaller components to realise the purpose of the component, which brings us to the next point.

There is a difference between what the artefact is designed to do, i.e., its function, and *how* that is realised. This follows directly from the modularisation, since there may be multiple ways of breaking up the design into smaller components. For some simple artefact, such as a hammer, these options are limited, but this is not so for software artefacts. Even a fairly straight-forward webpage, where all content could be declared in one single HTML file, can be split up into multiple files: the text to be displayed written in an `.html` file, the layout relegated to a style sheet in `.css` called from the `.html` file, and some javascript in a separate `.js` file also called from the `.html` file. For programs, e.g., Java generates separate `.class` files for each class declared in the code—fewer classes thus result in fewer files, and while a single C source file will result in a single program file, in the intermediate steps there are other files and processes involved: when compiling the source file into machine code, `.lib` files are called by the linker to generate the program file. It is possible to devise source code that consists of one file, does not make sense to divide up, and does not import anything, but this is a corner case exception rather than the norm (e.g., it may appear in an introduction to programming course, in the first lecture), especially for production-level computer programs.

Related to function and modularisation is the *user experience* of the computer program, where a function may be realised by more or less well integrated components that may be automatically invoked or manually started. Manual actions may give the user the impression of it being distinct but related components, whereas automated processes will provide the impression of it being one entity. This may be by design. For instance, one could by default block the running of scripts in spreadsheets, so when there is one, the user has to approve of its execution; when running scripts is enabled by default, on the other hand, end-users will not be aware that it is executing the script as a sub-process of the spreadsheet process or even that there is a script running.

3.2. Mode of participation in the relation of the files in the software

We have to consider the mode of participation in the relation between the files of a program, and, more precisely, assess how the optional, mandatory, immutable, and essential participation apply to source code and the compiled program. We will address this under the assumption that the program is free of bugs.

Optional participation of a file: In the interest of optimal design, there should be no optional files either in the source code or in the running program—if it can continue to operate without any bugs without that file, then that file is redundant and should be removed. Nonetheless, there very well may be sub-optimal source code and programs, especially when some modules or libraries are imported: operationally redundant files may not have been removed to save oneself testing whether it could be removed. For the running example: the programmers did not check whether all the files of the imported Owlready are used. Probably they are not, since the tool only fetches data from the OWL file but does not modify it, but there are functions stored in separate files to modify the ontology, such as `instance_editor.py` that is part of Owlready.

Mandatory participation of a file: the file has to be present, but it can be swapped for another without altering the function of the program. For the running example, this applies as well: the code looks up nouns in the `nncPairs.txt` file for any axiom it has to verbalise. If the file is not there, it will throw an error, but it can be overwritten with a different file also called `nncPairs.txt` that has the same outward function and minimal (but possibly more) content, e.g., one that has more nouns with noun classes listed in the file. The function remains the same regardless.

Immutable participation of a file: applying the definition of immutable, then when the program performs a particular role, it is essential to it. It is debatable whether this mode of participation would be applicable for either source code or compiled code. If at some point it has to perform some function x, then it is part of the specs at all times. The verbaliser has two 'modes' of operation—terminal-based and an end user interaction with a GUI. Tkinter is only essential for the GUI operation mode, which still performs the essential function of verbalising ontologies, so `Tkinter` might be an immutable part of the verbaliser. Yet, if the requirement is 'to be able to operate in two modes, terminal and with end user formatting', then omitting `Tkinter` breaks the overall functionality, and would therewith be at least mandatory if not essential, rather than immutable.

Essential participation: remove that particular file and the program is broken. In praxis, there are different levels of 'broken', such as minor bugs but the main function still can be achieved, major issues so that it only works partially, and the program not being able to run at all. From the binary viewpoint of 'bug-free or not', these differences do not matter: if not all specified functions work when that particular file is removed, then it is an essential file. For the OWL verbaliser, this may hold for Owlready: if any other version than `Owlready-0.3` is not backward or forward compatible, then that version is essential to the function of verbalising ontologies by the isiZulu verbaliser.

In sum, for bug-free operation, straight-forward mandatory participation in the (presumably part-whole) relation is the default case, with optional participation amounting to time-constrained coding and essential participation would be brittle coding.

Regarding sharability of a program's files: it depends on whether a file has to have a lock on it when open because of concurrency. If so, then that component file of the program is only sequentially sharable. Due to the unclear ontological status of programs, exclusivity is debatable depending on the choices—among others, whether copies count as distinct and whether it is the file as information artefact, etc. It does not matter for the notion of whole, and therefore we leave this aspect for future work.

3.3. Functional parthood

Since there is the compositionality thanks to modular design, and the types of files with their contents are distinct rather than subsets, the applicable relation between those component files and the whole—Source code or Executable [compiled/interpreted] computer program—is that of *parthood*. One may argue that it would also be a *proper* parthood in most cases, since it is unusual for production-level software to consist of one file only, where the part-files may or may not be shared either sequentially or concurrently, depending on the exact configuration of the code.

Since multiple part-whole relations have been proposed, this relation may be refined at some stage. Currently, however, because the ontological status of a file and of a computer program is not fully settled [8], a refinement along the line of [9] is not possible yet. The program does have a specific assigned function, however, such as Text processing for *MS Word* and Song management for *iTunes*, which each have sub-functions, such as Text formatting, Printing, and Playlist creation, i.e., there are functions and part-functions, and they in turn may have part-functions, such as Change text to bold face, Add song to playlist etc. Many types or categories of function have been identified and the ontology of function [21] identifies 89 of them. Also, several definitions of function have been proposed to assist with specifying functions. The Oxford Dictionary[5] states that function is an "activity that is natural to or the purpose of a person or thing". Yet, a function of an object might never be realised and so then cannot be an activity that is happening; e.g., a newly bought hammer may never be used. Also, an object can have one intended function but may be used for something else; e.g., an old cooking pan may obtain the function of flower pot. These sort of complications have been investigated in detail and the aim is to avail of that here and use it as-is, rather than digress in details. Mizoguchi et al. [22] did so most recently, who define function as "a role played by a behavior specified in a context", after observing that a "function is necessarily supported by the structure and/or properties of the things" and that "one of the most significant properties which function must have is implementation-independence." [21]. While this definition is less straightforward than the Oxford one, it is more accurate and at least it is broad enough to include the hammer and cooking pot cases or, say, repurposing a Web browser in full screen mode as a presentation application.

Taking function as such and proceeding with the 'vertical' relation between the part and the whole, then other than to realise the overall function, there must be component functions that contribute to that and the structures that realise that would thus be "functional parts" of the whole in some way [23,24]. The definition of *functional part* by Mizoguchi and Borgo [23] is: "Given an entity A and a behavior B of it, a functional part for that behavior is a mereological part of A that, when installed in A, has a behavior that contributes to the behavior B of A." This is somewhat underspecified when taken in isolation, since it has to be seen in the context of their definition of function. We refine the definition as follows:

Definition 1 (Functional part) *Given an entity x and a behaviour of type B of it that plays the role of function F in that context c, a functional part for that behaviour of type B is a mereological part of x that, when installed in x, has a behaviour of type D that contributes to B of x, where D is plays the role of part function F′.*

[5] `https://www.lexico.com/definition/function`; last accessed on 14 July 2020)

Mizoguchi and Borgo informally describe four types of functional parts; it is clear from the context in the paper, that it is to be understood as *proper* part. A part is a *genuine* functional part if it is installed correctly in x, from a structural viewpoint; a *replaceable* functional part is as specified for 'mandatory', above; a *persistent* functional part is either an essential or an immutable part; and a *constituent* functional part is a generic part regardless its assigned position as it may be temporarily taken out physically [23, p6]. A possible further interpretation with respect to a role theory, as alluded to in [23], may be of interest, but these four listed already help clarify the general modes of participation for files as expounded upon in the previous section. Under the assumption that source and compiled code each have more than one file and there are no optional files, then

- Computer program source code: all files are genuine functional parts and replaceable. The files have to be in specific locations to be called.
- Computer program compiled/machine code: all files are genuine functional parts and possibly also persistent (if not, then they are replaceable). The files have to be in the right folders and only there.

This settles the 'vertical' relation between the parts and the whole. We will turn to the relation among the parts in the next section.

4. The relation that binds the components

Taking stock, the files participate mostly mandatorily in the genuine functional parthood relation with the source code or compiled and executable program, normally there is more than one file involved, and each file either contributes to some function or performs one or more functions to contribute to the specified function(s) of the program. In terms of applying Eq. 1, a first step would be the following:

$$\forall xyzt(genuineFP(x,y,t) \rightarrow (genuineFP(z,y,t) \leftrightarrow B(x,z,t))) \tag{3}$$

Observe that it is time indexed, since it was so in the original (Eq. 1) and there is no harm in keeping the door open for code versioning; hence, it is more precise and stable. This brings us to question Q2 posed in Section 1: resolving the "unifying relation" B for the source code and the compiled program, and related loose ends from the formalisation of unity in Section 2. Characterising B from Eq. 3, i.e., the relation among the parts, ensures we arrive at either it being a (functional) whole since it has unity, or, if it cannot be specified, that then there is no whole after all. The exposé on unity in [16] does not have even one example of such a unifying relation, however, nor do the OntoClean examples [17]. Others, such as [14,25], do not fare better, like with a mere "x is unified in the right way" [25, p180], or go the other extreme with non-reusable situation-specific instance-level relations like "carrying out research in the same sub-area of the area of distributed systems in the University of Twente" [12, p159]. For instance, while Bikini is a functional whole consisting of two parts, the particular type of unifying relation that holds between the two parts, for any bikini, is left undetermined. Assessing programs, this unifying relation needs to be ascertained, however, since it would help explain *why* it is a whole.

Let us first consider two cases with common sense examples. For a morphological whole, such as the Constellation example (e.g., the stars making up the *Sagittarius* sign), B might be that all the stars involved share that they participate in the connecting the drawing lines relation to make up the figure. For social wholes, such as Electorate

of a country, its parts or members of the collection—the voters—share that they stand in a relation fellow adult citizen of some country. Applying this to Eq. 1, then the latter example would be formalised as follows: $\forall xyzt(VoterIn(x,y,t) \rightarrow (VoterIn(z,y,t) \leftrightarrow fellowAdultCitizen(x,z,t)))$. So, if The Daily Show presenter *Trevor Noah* is a voter in the *Electorate of South Africa*, then if there is another voter, say, the national rugby team's captain *Siya Kolisi*, in the *Electorate of South Africa*, then *Trevor Noah* and *Siya Kolisi* are fellow adult citizens and vv.; hence, it can be applied.

Since the notion of unity seems to be usable, let's now consider computer programs: what is it that binds the files, other than the vague 'being another component'? Since source code is compiled, with as consequence that the files comprising the source code are different from the file(s) comprising the compiled program, the relation among the file is, perhaps, different. In both cases, we use the 'realistic case' argument, i.e., setting aside the corner case of one main file without calling anything else ever, since we are interested in practical applications. We shall address each in turn.

Recall that the whole is the source code for the computer program, such as *Firefox source code*, which operates at L_i level of granularity, regardless how that is organised. The components and the organisation thereof—i.e., the files that make up the source code like a file with the `main()` and imports—operate at L_{i-1} level of granularity. This is likewise for the computer program, such as the *Firefox executable app,* with the compiled code. There are various ways to link files to the main file; e.g., through an `import` statement in, among others, Python and Java, an `#include` in C and C++, and other coupling mechanisms [5, ch9]. An example of cascading imports is depicted in Fig. 2, which form a tree that may have files with more than one parent. Mathematically, it is a directed graph with a top-node and the other nodes may have more than one incoming edge and more than one outgoing edge, and ideally it is acyclic. A circular import may not be forbidden by the syntax of a programming language, but it is considered an anti-pattern and, consequently, there is an *Acyclic Dependencies Principle* with strategies for breaking cycles to foster good design [5, ch9]. Any two component files of the source code, being genuine functional parts, thus relate through that directed graph of dependencies, either further down in the hierarchy or upward and into another branch. One thus always can construct a path between any two nodes (ignoring the direction of the edge). Let us call that the *SCgraphPath* property, then Eq. 3 can be updated into Eq. 4:

$$\forall xyzt(genuineFP(x,y,t) \rightarrow (genuineFP(z,y,t) \leftrightarrow SCgraphPath(x,z,t))) \quad (4)$$

We still need to verify it is not just a contingent whole; either the following holds:

$$\neg\forall xzt(SCgraphPath(x,z,t) \leftrightarrow \exists y(genuineFP(x,y,t) \wedge genuineFP(z,y,t))) \quad (5)$$

or, at least, that it holds with Eq. 2's generic P. Is it possible that x and z are (genuine functional) parts of some whole y but not do stand in at *SCgraphPath* relation to each other, or v.v.? This is indeed very well possible: they may be parts of, e.g., a library y of header files or x and z are in a repository of modules y. In that case when in a library or repository, x and z are neither in the *SCgraphPath* relation nor, arguably, *genuineFP* but another part-whole relation, since there they are members of a collection.

Lastly, Eq. 1 implicitly assumed weak supplementation, although not formalised as such, and thus that there ought to be more than one source code file. There exist source code that has no `import` or `#include` statement, although this is rare for production-

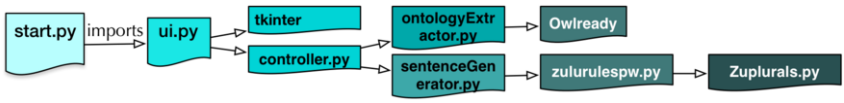

Figure 2. Series of actual `import` statements of the isiZulu OWL verbaliser, for the Python files only (it calls other files as well, not shown).

level software. Still, with the current formalisation, this is not a problem, since 1) weak supplementation is asserted for *PP*, not *P*, but *P* is used in the unity axiom and 2) for "$\neg\forall xz$" in the axiom to hold, even just one example satisfies it, which we just have with the module library; hence, this is not a problem even for the corner case.

We thus can define source code to be a functional whole as follows:

Definition 2 (Program source code as a functional whole) *For any computer program (y) in source code form that has one or more genuine functional parts $x_1,\ldots,x_n$ with $n \geq 1$ and $1 \leq i,j \leq n$, and time $t \in \mathscr{T}$ where $\mathscr{T}$ is the set of discrete time point, it holds that $\forall x_i x_j y t(genuineFP(x_i,y,t) \rightarrow (genuineFP(x_j,y,t) \leftrightarrow SCgraphPath(x_i,x_j,t)))$. For at least some whole (y'), $SCgraphPath(x_i,x_j,t)$ does not hold, i.e.,:* $\neg\forall x_i x_j t(SCgraphPath(x_i,x_j,t) \leftrightarrow \exists y'(genuineFP(x_i,y',t) \wedge genuineFP(x_j,y',t)))$.

Next, how can Eq. 3 be finalised for the executable program? At that stage, the explicitly stated to be imported files have been included, so *SCgraphPath* is not applicable. Yet, there still may be multiple files that are part of the computer program that "brings about a result"; examples include an `.ini` file for initialisation setting, an `.xml` file with the locale setting for menu options in a different language, and a dictionary to load for spellchecking. Some of these files could have been put 'inside' the executable, but from an engineering and usability viewpoint, they allow for more flexible customisation of the program and *shareability* of files when accessible separately. In any case, regardless the variation in configuration, if there is more than one file for the compiled code, they are *linked*—in the sense of the compilation processes in compilers, not the generic term 'linked'—so that for compiled code, we obtain a similar definition as for the source code functional whole, but then $linked(x_i,x_j,t)$ rather than with $SCgraphPath(x_i,x_j,t)$. There are several differences between the two relations: the latter is mutable and the former is not; the latter can have those files swapped, the former not; they hold between different types of files; and the latter has the links manually specified by the programmer's design, whereas this happens automatically in the compilation by the optimisation algorithms.

Can x_i,x_j be linked but not be part of a computer program y, or be genuine functional part but not linked, i.e., that the bi-implication does not hold? For code, there are *object code libraries* that are used in the linking stage of the compilation process, which x_i,x_j are part of and while being part of that, they are not linked. With sub-optimal program code and that optional participation, surely also at least one example can be found where either x_i or x_j is not a *genuineFP* of y. We thus obtain the following definition:

Definition 3 (Compiled program as a functional whole) *For any computer program (y) in compiled form that has one or more genuine functional parts $x_1,\ldots,x_n$ with $n \geq 1$ and $1 \leq i,j \leq n$, and time $t \in \mathscr{T}$ where $\mathscr{T}$ is the set of discrete time points, it holds that $\forall x_i x_j y t(genuineFP(x_i,y,t) \rightarrow (genuineFP(x_j,y,t) \leftrightarrow linked(x_i,x_j,t)))$. For at least some whole y', $linked(x_i,x_j,t)$ does not hold, i.e.,:* $\neg\forall x_i x_j t(linked(x_i,x_j,t) \leftrightarrow \exists y'(genuineFP(x_i,y',t) \wedge genuineFP(x_j,y',t)))$.

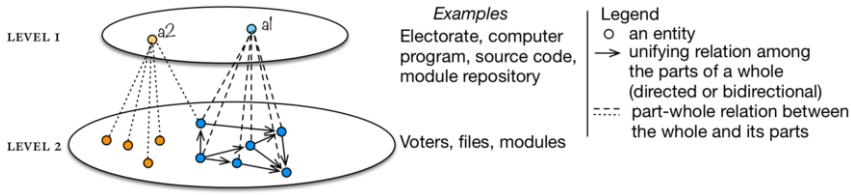

Figure 3. Levels of granularity, with entities residing in each level, which may be related 'horizontally' at the same level and 'vertically' between the parts and the whole residing in different levels. The orange object `a2` visualises a collective with its members and the blue `a1` visualises an integral whole with its parts.

The situation is visualised in Fig. 3, which shows two of the possibly many levels of granularity, each having entities residing at that level. The 'vertical' relations between the whole at level L_1, and its parts at the finer-grained level L_2 are typically a part-whole relation, but may be different [18]. The relation among the parts at a particular level is the unifying relation, such as being in a source code graph and begin a fellow adult citizen of a country. Informally, this constructs a triangle of relations between the entities: if you have one vertical side, there must be the other two.

5. Computer programs as wholes, collections, or set?

Having answered Q2 and Q3—*genuineFP*, *SCgraphPath*, and *linked*—we now can turn to Q1: is it a whole, a mere collection, or just a set? It has been argued colloquially that OH Dean puts forth in his Handbook of South African Copyright Law that the component files of a computer program should be considered to be a set of independent files for the purpose of copyright infringement assessments. However, clearly, a program is not just a set of discrete independent files, for the files have a specific relation to each other, and they need to have that tight coupling to achieve the proper functioning of the program. Sets, on the other hand, do not pose any such conditions on their members.

That it is not a collection either may be less obvious, and the arguments may depend on the definition of collection. With the requirement on the parts or grains to have to perform the same role [12,13], the program is *not* a collective, because its components do not have all the same function or role nor are they necessarily the same type of file. Moreover, a crucial aspect of modularisation in software development is to separate the different functions into different modules and files. Thus, when following established software design principles, it would never be a collective with this requirement.

Considering then Copp's criteria for collective [14]: the program can and does perform actions, therewith meeting one of the three criteria. He focusses on *social* collectives only, and therefore also proposes the requirement that a collective is composed of persons and has a "plausible theory of the legal system", which clearly do not apply. He generalises collective so that possibly they are "mereological sums of collective stages linked by a unity relation" where the "unity" refers to a property of the whole to track the collective through time to determine diachronic identity and the mereological sum applies to aggregating the time-slices, not about the parts being summed into a whole. Diachronicity is not applicable here as a criterion, however, supporting yet further the notion that software is not a collective. Thus, also by Copp's criteria and definition, a computer program is not a collective.

It does raise again the topic of mereological sum: could a program be just a mereological sum? Bare mereological sums permit a sort of 'contingent unity', however, like a sum of *your right index finger and right thumb*. This is not the case with program code, since each unity is *deliberate* by design and has properties at the level of the whole. Guarino and Welty tried to avoid simple mereological sums by assuming that "An object x is an *intrinsic whole under* ω if, at any time where x exists, it is a contingent whole under ω." [16] where their ω is B here. That is, B is essential to the whole, whereas just a mereological sum need not to have B holding among its parts all the time or even at some time. For instance, a Python module repository may contain the modules Owlready v0.3 and NLTK v2, which is a mereological sum and also a collective, yet the modules are independent from each other and can be removed; the files in one's 'Downloads' folder have even less unity. The files of a particular version of the source code or compiled program, however, *always* will have a binding relation to each other for as long as they exist; hence, then indeed is an intrinsic whole under *SCgraphPath* or *linked*, respectively.

Having used both the process of elimination and having provided supporting arguments as to what makes a whole a whole (recall Definitions 2 and 3), a computer program—be it in source code form or as compiled code—qualifies as being a *whole*. It is a *functional* whole, since the function aspect comes from the whole having a function, which has 'sub' or part-functions (recall Section 3.3), as per design of the artefact. Thus, then the source code and the computer program are a *functional whole*. The definitions implicitly also indicate the *boundary* of a program: those files in the source code graph/linked are surely 'in', with their function calls and inclusion by reference. That means other files not in that graph are 'out'. Notably, there are also loose coupling mechanisms, such as system calls for an application to request a service from the OS directly or through an API and pipes between programs. Excluding such loose coupling mechanisms is also in line with GNU's licensing interpretations on open source software for distinguishing between whether there are two programs or one with two parts[6]. If such loose coupling were to have to been included and somehow the OS would become part of the app due to a system call or API to that extent, then no non-open source program would legally be allowed to run on an open source OS, which is clearly not the case. More consequences may ensue from the definitions and arguments provided, which is left for future work.

6. Conclusions

An argumentation was presented that a computer program—be it as source code or as compiled (machine) code—is a functional whole, and why. The unifying relation among the parts (files) is the graph for source code and being linked for compiled code. The relation between the component files and the program is one of functional parthood, since the files perform (a) subfunction(s) of the function of the program. These additional insights into computer programs and the notion of the internal structure of wholes may assist practically with, among others, litigation cases in software development, illegal downloads, and copyright infringements, as well as more generally by having demonstrated that the notion of a unifying relation is indeed operationalisable.

[6] `https://www.gnu.org/licenses/gpl-faq.en.html#MereAggregation`; last accessed: 16-7-2020.

References

[1] Samuelson P. API copyrights revisited. Communications of the ACM. 2019;62(7):20–22.

[2] Brachmann S. Arista pays cisco $400m to end patent litigation at district court and itc; 2018. Last accessed: 8-6-2020. *IPwatchdog*. Available from: https://www.ipwatchdog.com/2018/08/08/arista-pays-cisco-400m-end-patent-litigation-district-court-itc/id=100102/.

[3] Cairns P. Does afrocentric have a R1bn problem?; 2018. Last accessed: 8-6-2020. *The Citizen*. Available from: https://citizen.co.za/business/1979904/does-afrocentric-have-a-r1bn-problem/.

[4] Bjørner D, Eir A. In: Compositionality: Ontology and Mereology of Domains – Some Clarifying Observations in the Context of Software Engineering. Springer Berlin Heidelberg; 2010. p. 22–59.

[5] Noback M. Principles of Package Design: Creating Reusable Software Components. 1st ed. Apress; 2018.

[6] Prehofer C, van Gurp J, Bosch J. Compositionality in software product lines. In: Emerging Methods, Technologies, and Process Management in Software Engineering. John Wiley & Sons; 2008. p. 21–42.

[7] Tripakis S. Compositionality in the Science of System Design. Proceedings of the IEEE. 2016;104(5):960–972.

[8] Turner R, Angius N, Primiero G. The Philosophy of Computer Science. In: Zalta EN, editor. The Stanford Encyclopedia of Philosophy. spring 2019 ed. Stanford University; 2019. .

[9] Keet CM, Artale A. Representing and Reasoning over a Taxonomy of Part-Whole Relations. Applied Ontology. 2008;3(1-2):91–110.

[10] Varzi AC. Mereology. In: Zalta EN, editor. Stanford Encyclopedia of Philosophy. fall 2004 ed. Stanford; 2004. http://plato.stanford.edu/archives/fall2004/entries/mereology/.

[11] Winston ME, Chaffin R, Herrmann D. A taxonomy of partwhole relations. Cognitive Science. 1987;11(4):417–444.

[12] Guizzardi G. Ontological Foundations for Structural Conceptual Models [PhD Thesis]. University of Twente, The Netherlands. Telematica Instituut Fundamental Research Series No. 15; 2005.

[13] Rector A, Rogers J, Bittner T. Granularity, scale and collectivity: When size does and does not matter. Journal of Biomedical Informatics. 2006;39:333349.

[14] Copp D. What Collectives Are: Agency, Individualism and Legal Theory. Dialogue. 1984;23:249269.

[15] Artale A, Guarino N, Keet CM. Formalising temporal constraints on part-whole relations. In: Proc. of KR'08. AAAI Press; 2008. p. 673–683. Sydney, Australia, Sept. 16-19, 2008.

[16] Guarino N, Welty C. Identity, Unity, and individuality: towards a formal toolkit for ontological analysis. In: Horn W, editor. Proc. of ECAI'00. IOS Press, Amsterdam; 2000. p. 219–223.

[17] Guarino N, Welty C. An Overview of OntoClean. In: Handbook on Ontologies. Springer Verlag; 2009. p. 201–220.

[18] Keet CM. A Formal Theory of Granularity [PhD Thesis]. KRDB Research Centre, Faculty of Computer Science, Free University of Bozen-Bolzano, Italy; 2008.

[19] Keet CM, Xakaza M, Khumalo L. Verbalising OWL ontologies in isiZulu with Python. In: ESWC'17 Satellite Events. vol. 10577 of LNCS. Springer; 2017. p. 59–64.

[20] Lê DTM, Janicki R. In: Peters JF, Skowron A, editors. A Categorical Approach to Mereology and Its Application to Modelling Software Components. Berlin: Springer Berlin Heidelberg; 2008. p. 146–174.

[21] Mizoguchi R. Functional Ontology of Artifacts. In: Interdisciplinary Ontology Conference 2008. Keio University Press Inc.; 2008. p. 1–10.

[22] Mizoguchi R, Kitamura Y, Borgo S. A unifying definition for artifact and biological functions. Applied Ontology. 2016;11:129–154.

[23] Mizoguchi R, Borgo S. A Preliminary Study of Functional Parts as Roles. In: Proc. of FOUST-II – JOWO'17. vol. 2050 of CEUR-WS; 2017. p. 9p. 21-23 September 2017, Bolzano, Italy.

[24] Vieu L, Aurnague M. Part-of Relations, Functionality and Dependence. In: Categorization of Spatial Entities in Language and Cognition. John Benjamins, Amsterdam; 2005. .

[25] Koslicki K. Substance, Independence, and Unity. In: Feser E, editor. Aristotle on Method and Metaphysics. Palgrave McMillan; 2013. p. 169–195.