What is a Subprogram?

José Hernández-Orallo

Universitat Politècnica de València.
Departament de Sistemes Informàtics i Computació,
Camí de Vera 14, Aptat. 22.012 E-46071, València, Spain
E-mail: jorallo@dsic.upv.es

Abstract. A fundamental (but usually neglected) question of computer science is addressed: "what is a subprogram?". To give a formal answer, several characterisations are introduced based on three properties: 1) a subprogram *S* of *P* must be a part of *P*, 2) the output of *S* must be a part of the output of *P*, and 3) *S* must be operationally encapsulated in *P*. Inspired in a proper and formal definition of part [8] based on Kolmogorov Complexity, this paper formalises the previous properties, and studies the combinations of properties 1-2, 1-3, and 1-2-3. The implications and applications from them derived are investigated. The descriptional and formal characterisations of the notion of subprogram that are presented also help understand and formalise many issues which have usually been addressed specifically or an informal way.

1 Introduction

The word 'subprogram' has been falling into disuse in the recent years. Perhaps symptomatically, some popular books on software engineering have even removed the notion of subprogram in later editions (from [19] to [20], for instance). The reason may be found in that the original notion of subprogram of the procedural paradigm as "*a part of the whole program which solves part of the problem*" is no longer valid for other paradigms: declarative, object-oriented, component-based, etc. Other notions have been gradually replacing it, such as module, class, component, subsystem and so on, with the aim of giving a new name to "*a part of the whole program which solves part of the problem*", because the classical notion of subprogram as procedure was not applicable for the new paradigms.

There is, then, a constant notion with changing names and different specific approximations. This paper investigates formal and generic characterisations of this constant notion, and the name "subprogram" will be used for it. Almost every computer scientist has an intuitive notion of subprogram, at least for the programming language (or languages) s/he is used to work with. For functional/procedural programming languages, a function can be recognised as a subprogram, because it is responsible for part of the behaviour of the program.

For object-oriented languages, a class should be better distinguished as a subprogram. In other cases, a cohesive set of functions or classes (i.e. a module) can be identified as a subprogram. For declarative languages, however, a subprogram is defined as a subset of clauses or rules which observe some additional properties about the model of the whole program and the model of the subprogram. Finally, the notion must be much more intuitive and informal for other programming languages or descriptional machines (e.g. Turing machines).

The notion of subprogram is then very varied. This is not dangerous, as far as it is generally used in an informal way, or specifically formalised for languages whose semantics follow some particular properties (e.g. declarative languages). However, from a generic and theoretical point of view, the question of formally stating what a subprogram is has not been addressed since the introduction of the current view of program and computation by Turing [21].

The reason may be found in that the notion of subprogram seems elusive in general. For any Turing machine program, for instance, if a bit is changed or removed, a completely different behaviour can be expected or, as was shown by Turing, the program could not even halt. And the things get more complicated when we consider the existence of incorrect programs, usual in other programming languages or machines. Even more, a proper program codification scheme can be chosen maximising this different behaviour for each bit is changed.

Nonetheless, given a program P and a program P' as a result of removing a bit from P, the information of P and P' is almost equal but their outputs (denoted by $\phi(P)$ and $\phi(P')$ respectively) may be completely different. In other words, P' can be considered a part of P, but, should P' be considered a subprogram of P?

This phenomenon leads commonly to the view that a notion of subprogram (or any generic approximation for it) cannot be defined for any universal machine. It has been even argued that the extremely discontinuous relationship between program and behaviour makes the definition of subprogram nonsensical.

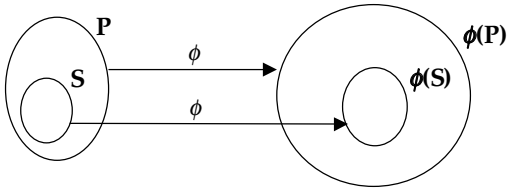
In our opinion, this is due mainly by a misconception of the nature of subprogram. The key issue of the problem of defining what a subprogram is lies on the fact that the notion of subprogram is cognitive. But this cognitive character does not mean that it is completely relative to the cognitive system, which would turn the problem unsolvable once again, but that *there are some informational / descriptional properties that any cognitive system uses when recognising a subprogram*. The same idea has led to the universal notion of Kolmogorov Complexity (or the length of the shortest description) of an object [14], the view of pattern recognition as information compression [22], and a recent definition of universal and cognitive versions of distance between any two given objects [3].

In this paper, we will investigate the use of descriptional tools such as Kolmogorov Complexity and related concepts to essay formal characterisations of subprogram. The characterisation will be descriptional, as well as the behavioural equivalence (much simpler and quite different from algebraic characterisations of observational equivalence such as [1]). The descriptional relationship between program and evidence is completely asymmetric. The program gives all the

information about the output but, in general, the output does not give much information about the program. The program is just a possible solution for it. In order to formalise the notion of subprogram under descriptional complexity, another important issue must also be realised: the relationship between program and subprogram has to be established on two levels. The first level relates the information of the program and the information of a subprogram under the intuition that a subprogram is part of the overall program. The second level must relate the output of the program and the output of the subprogram. More precisely, the following properties will be considered by the definitions of subprogram of this paper:

1) A subprogram *S* of *P* must be a part of *P*.
2) The output of *S* must be a part of the output of *P*.
3) For any program *S'* such that $\phi(S') = \phi(S)$ then *S* can be replaced by any *S'* in *P* giving the same original output of *P*.

The first property accounts for the first level whereas the second and third properties try to account for the second level. The second and third properties are responsible for the definition to make sense but they are also more debatable. For property 2, although it is usually assumed that a subprogram must solve part of the problem, even if the problem can be exclusively represented by the output, this part of the problem can be very different from its appearance, because many abstraction steps could have taken place. It is even possible that the part which *S* solves is not inherent or intrinsic in *P*, but only necessary due to the implementation of *P*. This problem is patently clear if the idea of part is substituted by that of subset, as the following figure illustrates.



By using sets, the results are much more counterintuitive. Consider a program which is constructed as "schema + correction", which is quite common in software practice (normal behaviour + anomalous cases). The schema would not be a subprogram because its output is a superset and not a subset of the output of the whole program.

In other words, a relation of part is necessary for property 1 in order to be applicable to any descriptional language and this relation is also necessary for property 2 in the way that considers "1,2,3,4,5,6,7, …" a part of "1,2,3,a,5,6,7". Moreover, as we will see, property 3 can only be considered in general by the use of descriptional complexity.

Recently, several proper and formal definitions of part have been introduced [8]. This paper re-uses some of them in order to formalise properties 1 and 2, and other descriptional tools are also used to account for characterising subprogram.

The paper is organised as follows. Section 2 introduces some technical preliminaries about programs, descriptions, Kolmogorov Complexity and related notions. Section 3 discusses more deeply the properties that a subprogram should satisfy, and the first two properties are split up into four refined properties. Section 4 recovers the necessary definitions of part and partitions, necessary for defining that of subprogram. Section 5 introduces definitions of subprogram based on properties 1-2, 1-3 and 1-2-3, as well as several variants of program partitions. Section 6 studies some properties of the concepts introduced and the applicability to different kinds of programming languages. Section 7 discusses some existing applications for software science, modelling, machine learning and philosophy of Science. Section 8 closes the paper with the conclusions of this work and the open work.

2 Preliminaries

In order to define the notion of subprogram from a descriptive point of view we must first settle the notions of objects, descriptions and minimal descriptions.

Without loss of generality a binary alphabet $\Sigma = \{ 0, 1 \}$ can be chosen, and any machine can be coded to work with binary strings. An object is any element from Σ^*, with $\cdot$ being the composition operator, usually omitted. The empty string or empty object is denoted by ε. The term $l(x)$ denotes the length or size of x in bits (or elements from Σ).

Under the Church-Turing thesis, any *description* of any object of reality can be converted in a description in a computational machine, so our idea of 'object' is, in this sense, universal. It is important to realise that whatever perception or knowledge that can be acquired about an object or a system is given in a descriptive way.

The complexity of an object can be measured in many ways, one of them being its degree of randomness [14], which turns out to be equal to the shortest description for it [18]. Descriptional Complexity, Algorithmic Complexity $C(\cdot)$ or Kolmogorov Complexity $K(\cdot)$ formalise this idea, and it has been gradually recognised as a key issue in statistics, computer science, artificial intelligence, epistemology and cognitive science (see e.g. [18][6]).

Definition 2.1. Kolmogorov Complexity

The *Kolmogorov Complexity* of an object x given y on a descriptional mechanism (or bias) β is defined as:

$$K_\beta(x|y) = \min \{ l_\beta(p) : \phi_\beta(p|y) = x \}$$

where p denotes any "prefix-free" β-program, and $\phi_\beta(p|y)$ denotes the result of executing p using input y.

The complexity of an object x is denoted by $K_\beta(x) = K_\beta(x|\varepsilon)$. It can be seen elsewhere (e.g. [18]) that Kolmogorov Complexity is an absolute and objective criterion of complexity, and it is independent (up to a constant term) of the descriptional mechanism β. In other words, the invariance theorem states that any

universal machine can emulate another. For this reason many properties are proven just in an asymptotic way. Throughout the paper the relation $<^+$ will be used as the asymptotic extension of $<$, namely $a <^+ b$ iff there exists a positive constant k such that $a < b + k$.

In the following, unless specified we will assume that β is selected to observe that $\forall x, y \in \Sigma^* K_\beta(x|y) \geq 1$. If there is no possible confusion, the β subscript will be omitted. It is obvious to see that $\forall x \in \Sigma^* K(x \mid x) <^+ 0$ because there is always a program of constant size of the form "*print the input*". It is also easy to see that $\forall x \in \Sigma^* K(x) <^+ l(x)$ because there is always a program of size less than $l(x)$ plus a constant value of the form "*print x*". In the case that $K(x) \geq l(x)$ we say that x is random. We denote x^* as the first (in lexicographical order) β-program for x such that $K(x) = l(x^*)$. In this way we can say that an object is random iff $l(x^*) \geq l(x)$. It is obvious that $K(x \mid x^*) <^+ 0$. However, since $K(\cdot|\cdot)$ is not computable, it is shown elsewhere (e.g. [18]) that the other way is just $K(x^*|x) <^+ \log l(x)$.

Finally, although $K(\cdot|\cdot)$ is an absolute measure of information it is not computable because it does not consider time. In this way, $K(\cdot|\cdot)$ does not reflect a cognitive view of information. There have been many proposals to incorporate time to Kolmogorov Complexity. The most appropriate way to weight space and time of a program, the formula $LT_\beta(p_x) = l(p_x) + \log_2 \text{Cost}(p_x)$, was introduced by Levin in the seventies (see e.g. [16]). In this way, a variant of $K(\cdot|\cdot)$ can be easily defined from it:

Definition 2.2. Levin's Length-Time Complexity

The *Levin Complexity* of an object x given y on a descriptional mechanism β:

$$Kt_\beta(x|y) = \min \{ LT_\beta(p) : \phi_\beta(p|y) = x \}$$

This is a very practical alternative of Kolmogorov Complexity, because as well as avoiding intractable descriptions, it is computable. Later on we will come back to this variant to define cognitive versions of subprogram.

3 Properties of Subprogram to be Considered

As it was outlined in the introduction, the intuitive notion of subprogram is based on different properties at two different levels: comparing the programs and comparing the outputs of the programs. Concretely, the following properties have been advocated:

1) A subprogram S of P must be a part of P.
2) The output of S must be a part of the output of P.
3) For any program S' such that $\phi(S') = \phi(S)$ then S can be replaced by any S' in P giving the same original output of P.

To formalise the first two properties, it is only necessary to define formally what a part is. The third property will be specifically addressed in section 5. In [8] different variants of the notion of part are introduced by using Kolmogorov Complexity. They all are combinations of two primary views of the notion of part:

a) A removable version: a part should be described/removed/recognised significantly easier from the whole.
b) A constructive version: the whole should be described significantly easier from the part.

These two different views, combined with the two properties of subprogram give four properties:

a1) A subprogram *S* of *P* should be easy to recognise/describe from *P*.
a2) The output of *S* must be easy to recognise/describe from the output of *P*.
b1) The program *P* should be easier to construct from *S* than alone.
b2) The output of *P* is easier to construct if the output of *S* is given.

Many different possibilities are acceptable. In principle, the nine possible combinations (a1, a2 or both with b1, b2 or both) should be investigated. Fortunately, the following study is simplified, because in [8], it is shown that removable and constructive variants are asymptotically equal.

However, we will choose a specific version of part for the formalisation, but similar results are expected if other descriptional notions of part are used. As we will discussed at the end of the paper, very different results are obtained if non-descriptional views of part (such as subset, substring or subsequence) are used.

In particular, both a1 and a2 and both b1 and b2 will be assumed. The reason is that, for non-syntactical restricted machines such as Turing machines, given a program *P* for *Q*, *P* would always be a part of *Q* according to the constructive version, and *Q* would be a part of *P* according to the removable version. By using both versions this is not true in either case.

As it was discussed in the introduction, property 2 is the most debatable one One reason may be found in the fact that it precludes the consideration of subprograms with empty output. Concretely, note that according to property b2, an empty string is not useful for constructing anything. In the same way, under property a2, the shorter the output of *S* the more difficult it can be a part of a large output of *P*, since it is more difficult to go through all the output of *P* (telling in which position it is) than quoting $\phi(S)$. The case is extreme for the empty output.

Many useful parts of programs do not do anything apparent by themselves, so property 2 would be not observed by them. Several alternatives have been considered by the author instead of property 2. For instance, a softened property 2 could be contemplated as a mixture of levels:

2') The output of *P* must be easier to describe from *S*.

The problem of this property is that it is only useful if *S* is a compressed subprogram for a part of the output of *P*. If *P* is a 'bad' (non-compressed or highly redundant) program, probably any part of it will be also non-compressed, and *S* will not be found useful to describe the output of *P* shortly.

Another alternative is to withdraw property 2 (or 2'). In this case we come up directly to a notion of subprogram equal to the notion of part, and we refer to [8] for their properties and more refined versions than those presented in the next section. However, this would generate other problems, such as the existence of parts of programs which are not logical or structural programs and they have

nothing to do with the behaviour of the initial program (e.g. take a Turing machine program and select the 'subprogram' which consists of the odd bits of the initial program. This subprogram, if halts, would almost surely not do anything related about the initial program).

As a result, we are forced, for the moment, to assume property 2). Nonetheless, the definition of subprogram which is derived from 1a), 1b), 2a) and 2b) will be called non-auxiliary subprogram, because later on other characterisation of subprogram will be introduced, which will allow the existence of subprograms that help the whole program but do not yield a part of the overall output.

Let us introduce the definitions of part and partitions that will be used for the different variants of subprogram:

4 Part and partitions

This section reviews some of the notions introduced in [8] and some other new results which will be necessary for the rest of the paper.

The first version of part is based on the idea of "a part can be easily *described* or *removed* from the whole", formalised in the following way:

Definition 4.1. Removable Universal Part

An object X is a removable universal part of an object Y in β, denoted by $X \overset{\downarrow}{\subseteq}_\beta Y$, iff $K_\beta(X|Y) \leq \log K_\beta(X)$

The meaning of the definition is that the part is significantly easier to describe if we have the whole that if we must describe the part from scratch. The use of the logarithm is a very convenient way for measuring this "significantly easier", although other function could be used. It is interesting to compare and differentiate the definition of part with the notion of subset or substring.

Specifically, definition 4.1 has some technical inconveniences (quite different from classical properties of sets and substrings):

Theorem 4.1. ε is not always a part of any non-empty object. [8]

Theorem 4.2. Some objects are not parts of themselves. [8]

These two problems are solved in [8] by an appropriate use of constants and embedding operations, which can be applied to any descriptional machine. However, without any modification at all, it can be shown that:

Theorem 4.3. For every universal descriptional machine β, there is a constant k which only depends on β, such that for every object x with $K_\beta(x) > k$, then $x \overset{\downarrow}{\subseteq}_\beta x$.

Proof. Consider any universal descriptional machine β. Since it is universal there exists a program π of the form "print the input" with length $l(\pi)$. For every object x, we have that $K_\beta(x|x) \leq l(\pi)$, because the program π can be used to print the input x. Consequently, there exists a constant $k = 2^{l(\pi)}$, such that if $K_\beta(x) > k$ then $K_\beta(x|x) \leq l(\pi) = \log k < \log K_\beta(x)$, and, by definition 4.1, x is a removable part of itself, i.e, $x \overset{\downarrow}{\subseteq}_\beta x$. □

Although the expression is exponential, this constant is usually negligible, since the program print the input is usually extremely short.

On the other side, the other view of part as "the whole is *described* or *constructed* more easily if some of its parts are given" generates the following definition:

Definition 4.2. Constructive Universal Part

An object X is a constructive universal part of an object Y in β, denoted by $X \overset{\uparrow}{\subseteq}_\beta Y$, iff $K_\beta(Y|X) \leq K_\beta(Y) - K_\beta(X) + \log K_\beta(X) + 1$

The intention of the term $-K_\beta(X) + \log K_\beta(X) + 1$ is to force that almost all the information of X is used for constructing Y. Moreover, this is done without much adaptation cost, a cost that is represented by $\log K_\beta(X) + 1$.

It can be contrasted with many examples that definition 4.2 behaves in a similar way as definition 4.1. This is justified by the following theorem:

Theorem 4.4. Definitions 4.1 and 4.2 are equivalent up to an additive term $O(\min\{\log K(X), \log K(Y)\})$. [8]

Although theorem 4.4 says that definitions 4.1 and 4.2 are equivalent up to a small additive, they can be quite different for small objects. For this and other reasons, it would be interesting to use both definitions in a unified way:

Definition 4.3. Joint Universal Part

The object X is a joint universal part of an object Y in β, denoted by $X \subseteq_\beta Y$, iff $X \overset{\uparrow}{\subseteq}_\beta Y$ and $X \overset{\downarrow}{\subseteq}_\beta Y$

One of the nice properties of this joint definition is that it avoids that the program x^* would always be a part of x. In the case of $\overset{\uparrow}{\subseteq}$, it would be very likely, because $K(x|x^*) = 0 + k$, with k being very small. However, according to $\overset{\downarrow}{\subseteq}$, the disequality $K(x^*|x) < \log l(x) + k'$ is very close to the limit and the constant is large; this would rarely be less than $\log K(x^*)$ and it would not be a part.

On the other hand, it avoids that x would be always a part of the program x^*. According to $\overset{\downarrow}{\subseteq}$, it would be very likely, because $K(x|x^*) = 0 + k$, with k being very small. However, according to $\overset{\uparrow}{\subseteq}$, $K(x^*|x) < \log l(x) + k'$. In order to be less than $K(x^*) - K(x) + \log K_\beta(x) + 1$, since the first two terms are similar, x should be almost random to make $K(x) > l(x)$. In this case, x^* is "PRINT x" and $K(x^*|x)$ is close to 0. This is highly intuitive since a program "PRINT x" has as part "x".

Whatever definition of part is used, generically represented by $\subseteq_\beta$, we can define a proper subpart as:

Definition 4.4. Proper Subpart

The object y is a proper subpart of an object x in β, denoted by $y \subset_\beta x$,

iff $y \subseteq_\beta x$ but $x \not\subseteq_\beta y$.

Finally, we also define some definitions of partition:

Definition 4.5. Partition

A set of objects $Y = \{y_1, y_2, ..., y_m\}$ is a partition of an object x in β iff

$\forall y_i : 1 \leq i \leq m$: $y_i \subseteq_\beta x$ and there exists an ordering o_j of Y such that

$x \subseteq_\beta y_{o1} \cdot y_{o2} \cdot ... \cdot y_{om}$ where $\cdot$ represents the composition of objects.

Definition 4.6. Reduced Partition

The set of objects Y is a reduced partition of an object x in β iff it is a partition of x in β and $\neg\exists Y' \subset Y$ such that Y' is a partition of x.

Definition 4.7. Proper Partition

A set of objects $Y = \{y_1, y_2, ..., y_m\}$ is a partition of an object x in β iff

$\forall\, y_i : 1 \leq i \leq m$: $y_i \subset_\beta x$ and there exists an ordering o_j of Y such that

$x \subseteq_\beta y_{o1} \cdot y_{o2} \cdot ... \cdot y_{om}$ where $\cdot$ represents the composition of objects.

Other variants of partition are defined in [8].

5 Subprogram and Program Partitions

Once the idea of part has been formalised, a first definition of non-auxiliary subprogram can be directly derived from any of the previous definitions of part (4.1, 4.2 and 4.3) to formalise properties 1 and 2. By choosing definition 4.3, we have the following definition:

Definition 5.1. Non-Auxiliary Subprogram (or Subtheory)

The object S is a non-auxiliary subprogram of an object P in β iff

$$S \subseteq_\beta P \text{ and } \phi(S) \subseteq_\beta \phi(P)$$

Although we have seen that some objects are not parts of themselves, for most cases the property $x \subseteq_\beta x$ holds. To avoid these cases, a stricter version can be defined.

Definition 5.2. Proper Non-Auxiliary Subprogram (or Subtheory)

The object S is a proper non-auxiliary subprogram of an object P in β iff

$$S \subset_\beta P \text{ and } \phi(S) \subseteq_\beta \phi(P)$$

The main interest of definitions 5.1 and 5.2, and the use of the notion of part developed in [8], is that it gives a solution for the problem of finding the part of a program in any descriptional language (property 1), which is the most difficult one, and, secondly, it clarifies the problem of what is inherent, implicit or intrinsic [9] in the problem (to express property 2).

The term 'non-auxiliary' refers to the fact that, under definitions 5.1 and 5.2, only parts of a program which solve part of the whole problem are considered. Seen the other side, if a part of a program helps to the solution of the problem but the partial problem it solves is not inherent in the overall problem, then it is not considered a subprogram.

It would be then interesting to find an extension or variation of the previous definitions which would consider subprograms whose output is part of the solution and not part of the problem. The key issue lies in the notion of 'auxiliary'. Property 3 is taken up again to account for this:

3) For any program S' such that $\phi(S') = \phi(S)$ then S can be replaced by any S' in P giving the same original output of P.

In other words, this third property represents the encapsulation or independence property. If the specification of a subprogram is preserved, its implementation can change in any way without affecting the specification or behaviour of the overall

program. Note that we are not dealing with efficiency or parallel processes, where the execution time may affect the result of the overall program.

The formalisation of property 3 seems more complicated, though. The difficulty is due to the term "replaced". Let us try to formalise it:

Definition 5.3. Replacement

Given three objects S, S' and P, such that $S \subseteq_\beta P$, a replacement R of S by S' in P, denoted by $R = P[S \leftarrow S']$, is any object R such that:

$$\exists P' : \; K(P|<P',S>) \leq K(S|P) \text{ and}$$
$$K(P'|<P,S>) \leq K(S|P) \text{ and}$$
$$K(R|<P',S'>) \leq K(S|P)$$

The meaning of the previous definition is understood as a removal of S from P, which leaves P'. This P' adjoined with S' gives the replacement R. The use of descriptional tools instead of classical set operators to define replacement is that it allows the intuitive notion of substitution for any descriptional mechanism and not only those languages which are composed of set of clauses or rules. The term $K(S|P)$ represents the cost of extracting S from P, and the cost of constructing P from P' and S, extracting P' given P and S, and constructing R from P' and S' cannot be greater than this term, in order to avoid the introduction of unrelated information.

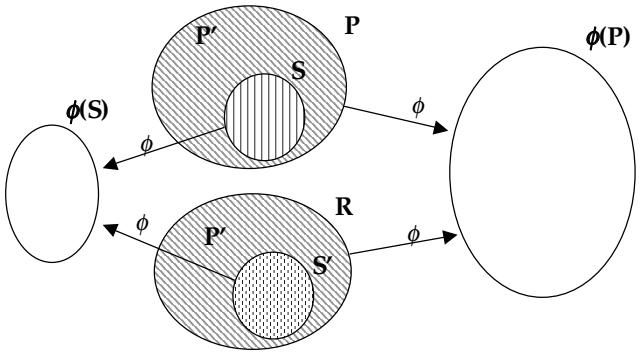
Finally, from here, a more general definition of subprogram can be introduced:

Definition 5.4. General Subprogram

The object S is a subprogram of an object P in β iff $S \subseteq_\beta P$ and

$$\forall S' : \phi(S') = \phi(S) \Rightarrow \exists R : R = P[S \leftarrow S'] \text{ and } \phi(R) = \phi(P)$$

which formalises properties 1 and 3. The following figure uses a analogy with sets to clarify these two latter definitions:



Note that this definition avoids the selection of a subprogram as a part of a program P in the way "S = the odd bits from P", because it is quite improbable that for every program S' such that $\phi(S') = \phi(S)$, the reconstruction of P with S' would give the initial $\phi(P)$. Nonetheless, there are some parts of programs considered subprograms by definition 5.4 which are somehow counterintuitive. Consider the case of a part of P which is not used; then anything that is put in its place will not affect the whole behaviour of the program. There is a way of

correcting this problem by considering that given a part S of a program P, if there is another S', such that $\phi(S') \neq \phi(S)$ and $\phi(P) = \phi(R)$ still holds, then S is not a subprogram. However, this condition would be much too strict, because sometimes the whole behaviour of the subprogram is not used (only part of it), and the rest can differ without affecting the whole program.

As a result, definition 5.1 could be considered at first sight a sufficient set of conditions for a subprogram, whereas definition 5.4 could be considered a necessary set of conditions. However, both are not at the same level, and it cannot be shown that, in general, a subprogram according to definition 5.1 is always a subprogram according to definition 5.4, because the fact that a subprogram can solve part of the problem (as definition 5.1 demands) does not entail that it is necessary used by the whole program (as definition 5.4 requires).

A joint definition between 5.1 and 5.4 is also possible, and it would be, in our opinion, a sufficient (although not necessary) characterisation to cover the intuitive notion of subprogram:

Definition 5.5. Basic Subprogram

The object S is a basic subprogram of an object P in β iff

$$S \subseteq_\beta P \text{ and } \phi(S) \subseteq_\beta \phi(P) \text{ and}$$

$$\forall S' : \phi(S') = \phi(S) \Rightarrow \exists R : R = P[S \leftarrow S'] \text{ and } \phi(R) = \phi(P)$$

In my opinion, it is difficult to find a definition which would be more approximate than definition 5.5 and, at the same time, still simple to be used.

For the moment, let us use the different versions which have been introduced to give several variants of program partitions.

Definition 5.6. Proper Non-Auxiliary Program Partition

A set of objects $Y = \{y_1, y_2, ..., y_m\}$ is a program partition of an object x in β iff Y is a proper partition of x and every y_i is a non-auxiliary subprogram of x.

Definition 5.7. Proper Covering Program Partition

A set of objects $Y = \{y_1, y_2, ..., y_m\}$ is a program partition of an object x in β iff Y is a proper partition of x and $Y' = \{\phi(y_1), \phi(y_2), ..., \phi(y_m)\}$ is a proper partition of $\phi(x)$.

It is easy to show that there is not always a proper program partition.

Definition 5.8. Proper General Program Partition

A set of objects $Y = \{y_1, y_2, ..., y_m\}$ is a program partition of an object x in β iff Y is a proper partition of x and every y_i is a general subprogram of x.

The use of the notions of program partition serve to analyse whether a program is separable while still covering the whole initial evidence, and this concept can be applied to many different areas, as will be discussed in section 7.

The reduced partition (definition 4.6) can also be used to give more variants of program partition, as well as other versions of partition introduced in [8], such as easy, natural, characteristic and thinner partitions.

6 Properties

The previous definitions of part, subprogram and the corresponding partitions are universal, in the sense they have been developed by using the absolute version of Kolmogorov complexity (K). However, as we said in the introduction, the notion of part and subprogram are cognitive, and the use of absolute Kolmogorov complexity can consider parts which are not so intuitive because it ignores time, whereas human notion of part is cognitive, and very intricate parts are not usually considered from an intuitive point of view. Moreover, K is not computable, and so restricts the applicability of the previous definitions.

Fortunately, the preceding definitions can be redefined in the obvious way by using Kt instead of K, and for the rest of the paper these re-definitions will be referred as cognitive versions. Moreover, many slight problems disappear if one assumes that $Kt(\varepsilon|X)=0$ and $Kt(X|X)=0$, and $Kt_\beta(x|y)$ is used in any other case, and also assuming that $\forall x, y \in \Sigma^* : x \neq \varepsilon$ and $x \neq y$, $Kt_\beta(x|y) \geq 1$.

For instance, these assumptions allow to show the following theorem without any limitation about the program size:

Theorem 6.1.

For any program P, P is a subprogram of itself (according to definitions 5.1, 5.4 and 5.5, redefined by using Kt instead of K_β).

Proof. For the three definitions we have that since $Kt(P|P) = 0$ then $P \subseteq_\beta P$, i.e. property 1. For property 2, it is also trivial that $\phi(P) \subseteq_\beta \phi(P)$. Finally, for property 3, we have that since $S = P$ then $\forall S' : \phi(S') = \phi(S) = \phi(P) \Rightarrow \exists R : R = P[S' \leftarrow S]$, and $\phi(R) = \phi(P)$, because this $R=S'$, since $\exists P'=\varepsilon : Kt(P|<P', S>) = Kt(P|<\varepsilon,P>) \leq Kt(S|P) = Kt(P|P)$, $Kt(P'|<P,S>) = Kt(\varepsilon|<P,P>) \leq Kt(S|P) = Kt(P|P)$ and $Kt(R|<P',S'>) = Kt(S'|<\varepsilon,S'>) \leq Kt(P|P)$. □

Other theoretical questions could be posed, such as the situation when P is a program for Q and Q is a program for R. In this case any positive relation is avoided under the joint notion of part. However, is it still the case when P is a program for Q and Q is a program for P?

For the moment, let us study other more practical properties of the notions of subprogram, especially their interpretation with classical programming languages and paradigms.

For instance, declarative languages have usually been defined as set of clauses or set of rules, in order to avoid the problem of part. In the case of functional programs, properties 1 and 2 can be approximated by using subset. Indeed, both properties are intuitive for the case of functional programs, since a subprogram is usually considered a subset of rules (property 1) that define a function, and the extension of this function is a part of the whole behaviour (semantics) of the overall program (property 2). On the contrary, property 3 is not appropriate for functional programs, at least if function symbols are mixed between subprograms (encapsulation is not preserved).

A similar result is obtained for logical programs, at least for definite programs (without negation) and other kinds of declarative languages with monotonic semantics. This inspires a definition of monotonic machine from a descriptional point of view:

Definition 6.1. Monotonic Machine

A machine β is monotonic if for every program P for β that halts and for every S such that $S \subseteq_\beta P$ then S halts and either $\phi_\beta(S) = \varepsilon$ or $\phi_\beta(S) \subseteq_\beta \phi_\beta(P)$.

If part is substituted by subset and ε by $\varnothing$, we have the classical definition of monotonic languages for declarative languages.

In a similar way, contextual-free languages are those languages for which if part is substituted by substring, property 3 always holds (e.g. in a context-free grammar a program is a term to be reduced).

However, in procedural and other modern programming languages, not only the notion of implicit subprogram is increasingly less clear but also that of explicit subprogram, and, usually, the term is no longer used. Modern object-oriented languages and new programming environments are based on multi-level collections of things (class hierarchies, methods dependencies, module organisation, separate implementation, components, themes…). To discern what a subprogram is or even a part of the whole application is difficult. For this kind of software systems is for which the notions developed in this work are more appropriate. Theoretical studies on these languages are many times impossible because some basic concepts such as subprogram are not available. The next section discusses some of the possible applications in this line.

7 Applications

It is not expectable that even with the use of *Kt* instead of *K* would make the characterisations of subprogram automatisable, due to its intractability. Consequently, their expected applications are mainly theoretical, although not only to software practice but to other different areas.

The theoretical study of software quality interdependence relies on how well a program can be separated into parts and the behaviour of each part. For instance, reusability, modificability and maintainability could be studied for any descriptional language, by using these and other tools from Kolmogorov complexity and even in a statistic fashion.

In this way, some results of declarative languages can be extended to other languages by the use of the notion of part instead of the notion of subset, at least theoretically. For instance, the generalisation taken in [10] for any constructive language based on rules could be further generalised to any descriptional mechanism.

Outside software science, the notion of subprogram has also been shown useful, especially for philosophy of Science, modelling and machine learning. A program can be considered a theory and viceversa. Notions like intensionality (avoidance of extensional parts of a theory) and consilience/coherence (unified

covering of the evidence) can be formalised by the use of subprogram and by establishing properties that these subprograms must observe. This has been done for intensionality in [12] and for consilience in [11]. In fact, an exception is defined as a subprogram that covers part of the evidence separated from the main rule. In this way, it is possible to formally define explanatory counterparts to classical descriptional approaches to induction, like the MDL principle [13].

In fact, the present stage of the notion of subprogram presented in this paper was initially much more informal and has co-evolved with the previous applications.

8 Conclusions

Some descriptional-based characterisation of subprogram have been introduced. Although they are difficult to apply for concrete cases and they must be understood in an asymptotic way, as for any application of Kolmogorov complexity, they make it possible to formalise and study theoretically some notions that depend on that of subprogram.

Many other notions and applications are envisaged, and more insight has to be made to fully understand the properties and implications of definitions 5.1 to 5.5, as well as more research in better and simpler approximations, if they exist.

Acknowledgements

The author would like to thank S. Lucas for discussing the notions of subprogram, especially for functional programs. C. Monserrat and M.C. Juan have also contributed with a fruitful discussion held about properties 2 and 3.

References

1. Arnold, A.; Castellani, I. An algebraic characterization of observational equivalence. *Theoretical Computer Science*, 156: 289-299, 1996.
2. Balcázar, J.; Díaz, J.; Gabarró, J., *Structural Complexity*, Springer Verlag, vol.1., 1988.
3. Bennett, C.H.; Gács, P.; Li, M.; Vitányi, P.M.B.; Zurek, W.H. Information Distance, *IEEE Trans. Information Theory*, IT-44:4, 1407-1423, 1998.
4. Blum, M., A machine-independent theory of the complexity of recursive functions, *J. ACM* 14, 4:322-6, 1967.
5. Chaitin, G. J., *Algorithmic Information Theory*, 4th printing, Cambridge Univ. Press 1992.
6. Gammerman, A.; Vovk, V. "Special issue on Kolmogorov Complexity" *Computer Journal*, Vol. 42, No. 4, 1999.
7. Herken, R. *The Universal Turing Machine*, Oxford University Press, 1994.[P&N7]
8. Hernández-Orallo, J. "Universal and Cognitive Notions of Part" in Ferrer, L. et al. (eds.) "Proc. of the 4th Systems Science European Congress", pp. 711-722, 1999.
9. Hernández-Orallo, J., *Computational Measures of Information Gain and Reinforcement in Inference Processes*, Doctoral Dissertation, University of Valencia, 1999.
10. Hernández-Orallo, J.: Constructive Reinforcement Learning, *Intl. J. of Intelligent Systems, to appear*. URL: http://www.dsic.upv.es/~jorallo/escritsa/IJISHern.ps.gz

11. Hernández-Orallo, J. "A Computational Definition of 'Consilience'", *Philosophica*, to appear.
12. Hernández-Orallo, J.; García-Varea, I. "Distinguishing Abduction and Induction under Intensional Complexity" in P. Flach, A. Kakas (eds.) *Proc. of the European Conf. of Artificial Intelligence (ECAI'98) Ws. on Abduction and Induction in AI*, pp. 41-48, Brighton 1998.
13. Hernández-Orallo, J.; García-Varea, I. "Explanatory and Creative Alternatives to the MDL Principle", *Foundations of Science*, Kluwer, to appear,
14. Kolmogorov, A.N., Three Approaches to the Quantitative Definition of Information, *Problems Inform. Transmission*, 1(1):1-7, 1965.
15. Kolmogorov, A.N., Logical basis for information theory and probability theory, IEEE *Trans. Inform. Theory*, vol. IT-14, pp. 662-664, sept. 1968.
16. Levin, L.A., "Universal search problems", *Problems Inf. Transmission*, 9:265-266, 1973.
17. Lewis, H.R.; Papadimitriou, C.H. *Elements of the Theory of Computation*, 2nd Edition, Prentice-Hall International, Inc., 1998.
18. Li, M.; Vitányi, P., *An Introduction to Kolmogorov Complexity and its Applications*, 2nd Ed. Springer, 1997.
19. Pressman, R. S. *Software engineering : A practitioner's approach*, 3th ed., McGraw-Hill, 1992.
20. Pressman, R. S. *Software engineering : A practitioner's approach*, 4th ed., McGraw-Hill, 1997.
21. Turing, A.M. "On computable numbers with an application to the Entscheidungsproblem" *Proc. London Math. Soc.*, series 2, 42:230-265, 1936. Correction, Ibid, 43:544-546, 1937.
22. Watanabe, S., Pattern recognition as information compression, in: Watanabe (ed.) *Frontiers of Pattern Recognition*, New York: Academic Press, 1972.