Defects and Faults in Algorithms, Programs and Instruction Sequences

Bergstra, J.A.

DOI
10.36285/tm.49

Publication date
2022

Document Version
Final published version

Published in
Transmathematica

Transmathematica DOI Prefix 10.36285 ISSN 2632-9212

Defects and Faults in Algorithms, Programs and Instruction Sequences

Jan A. Bergstra
janaldertb@gmail.com
Informatics Institute, Faculty of Science, University of Amsterdam

Submitted: 7 February 2021
Revised: 19 September 2022

Abstract

A new definition of algorithms is given, where algorithms are understood as cognitive 'entities,' the definition of which is done in tandem with so-called algorhymes, which are entities serving as documentation of algorithms. Algorithms are cognitions while algorhrymes are artifacts. Based on this definition the notions of fault and defect are reconsidered in relation to instruction sequences, programs and algorithms. Programs as well as algorithms are considered capable of containing moral defects, the notion of a moral defect is developed in some detail. The notion of a moral fault is considered implausible.

1 Introduction

I will use the notion of an instruction sequence with a required interface as outlined in [4] and [18, 22] as well as the introduction in [23], and the references cited therein. I will start with recapitulating a definition of the notion of program from previous work.

1.1 Definition of program

From [12] I paraphrase the following definition of a program.

Definition 1.1. *A program is an element of a collection of entities L (program notation) for which a computable mapping* $\mathsf{L2IS}$ *is known (called projection function in [12]) which turns $p \in L$ to an instruction sequence $X_p = \mathsf{L2IS}(p)$ in such a manner that the meaning of p consists of the behaviour of X when it is put into effect on a platform known for being able to process instruction sequences coming from many sources.*

Here the projection function may be achieved by computation of another instruction sequence X_C (serving as a compiler), which may result from application of the projection function to yet another another program.

For a program it is not essential that there actually is a machine able to perform the compilation, that is the compiler has been implemented and putting that implementation into effect can actually achieved on real machines. Programs may be practical entities in real life program notations or theoretical entities (pseudocodes) which merely define input for a possibly equally theoretical compiler. For further elaboration on this notion of program and the corresponding projection semantics I refer to [8] which deals with recursion and [15] which deals with the combination of Object Oriented (OO) programming and multi-threading.

Unfortunately, like all definitions of what a program is, which go beyond the world of expressions of a specific notation, Definition 1.1 has significant deficiencies. In this case first of all the nature of the instructions requires attention. I will assume as in most work on instruction sequences from [24] onwards that basic instructions use focus method notation $f.m$ where f is a focus, i.e. a name under which a service is found (say service F) and m names a method to be applied to F or, in other words, which F is asked/called/requested/ordered to process. The consequences of a call $f.m$ are threefold: (i) potentially a state change of the service, (ii) potentially interactions with other system components which the service is connected to, (iii) a single bit reply which is returned to the unit which is putting the calling instruction sequence into effect and which may influence the future behaviour of that unit.

When stating that the projection function L2IS may be computed by means of a (previously) determined instruction sequence (the compiler) it matters which services the compiler may use. If the functionality to be computed is merely a transformation from bit sequences of a fixed length n to bit sequences of fixed length m it suffices to make use of single bit memory services only, though allowing array-like services admitting indirect access (see e.g. [4] for a definition of arrays as services) will allow to write instruction sequences with fewer instructions (a lower LLOC, logical lines of code, metric in the terminology of [4]). For multi-threading on a single processor the same model applies (see [13]). For multi-threading in the context of distributed computing and in the context of non-deterministic interactive and parallel computation the limitation to single bit services and array-services do not suffice. I refer to [14] for possible options to be used in such cases. Determination of which services an instruction sequence may use constitutes a degree of freedom in Definition 1.1 which is a weakness of the definition at the same time. I hold that the concept of services is open ended and that future technologies will reveal new options for constructing services which will impact on what can be computed by means of an instruction sequence.

In Section 3, I will expand on the collection of available services and on the related notions of the required interface for a program and the provided interface for a service family.

1.2 Requirements specification and technical specification

The notion of a program is independent of any notion of specification of behaviour. A program may perfectly well serve as its own specification so to say, perhaps complemented with some comments and explanation. Although in computer science it is conventional to understand programs as artefacts which are supposed to comply with pre-existing specifications or requirements, I prefer to view specifications as no more than tools available to manage the design, construction, and use of programs. A specification need not be formal, it may also consist of informal user expectations. In some cases, in the presence of a specification, it is possible to assess a terminated or ongoing computation and to spot a deviation from specified behaviour. In that case a failure has been detected. So for programs, a notion of failure comes with a notion of specification, and a computation failing to comply with the specification.

I will assume that specifications come in two forms (at least): a requirements specification which provides information on what a program when in use will provide to its context and a technical specification which provides information on what precisely the program is supposed to perform in terms of inputs, outputs, communication, timing, and usage of resources, and which may also provide information about the modular structure of a program and about the various functionalities that these modules are supposed to deliver. A technical specification may be at odds with the requirements specification for the same (imagined) system. If a minor change in the specification settles the problem it is plausible to speak of a fault in the specification. The situation may be quite different, however, to such an extent that fault repair does not apply and redesign of the specification is needed, or worse, no redesign can succeed because the requirements are unimplementable. One may refer to a problem of that kind a RS/TS flaw (req. spec/ tech. spec. flaw). I will use dedicated terminology for the various RS/TS-flaws which may arise.

The requirements specification may further be decomposed into a functional requirements specification and a moral requirements specification. While a functional requirements specification focuses on how a program may contribute to overall system functionality, the contents of a moral requirements specification are much harder to grasp. Ideally a moral requirements specification must be such that (i) all moral faults will be captured as mismatches between system behaviour and requirements specification (as discussed in Section 4 below), and (ii) it will be clear which parts of the software in control of a system are responsible for various moral faults. Moral software fault is a remarkably complex notion, however, which cannot be captured in functional terms.

1.2.1 Technical specification and failure

I will understand a (program) failure, by default, as a mismatch between the behaviour of a program and its technical specification. In the absence of a technical specification a program cannot fail. If the program does not implement its requirements, again a mismatch, though coming to light by way of an assessment

which may well be informal because the requirements at hand are informal, I prefer to refer to that problem as a program/RS-flaw. A program/RS-flaw may find its origin in a problem with the program but just as well in a problem with the requirements specification.

If, however, it can be established that a program/RS-flaw works just as an ALR fault (as defined below), then the default is over-ruled and said mismatch can be termed a failure.

1.2.2 Faults as causes of failures

The terminology of fault as cause of failure was introduced in Laprie [33], while appearing in a more definitive form in Avižienzis, Laprie & Randell [2] and later in Avižienzis, Laprie, Randell, & Landwehr [3].

Definition 1.2 (ALR fault)**.** *An ALR (for Avižienzis, Laprie & Randell) fault is a program fragment which is the cause of an error which in turn is the cause of a failure.*

The concept of an ALR fault in a program `X` is a theoretical notion – it does not matter whether or not its being a fault has actually been detected either empirically or by other means. The idea of an ALR fault allows a ramification of different instantiations depending on how the notion of causality is understood. For a survey of interpretations of causality in the context of ALR I refer to [6] and [7]

1.2.3 Failures: how to define?

If a mathematical function is to be computed then a failure is simply a wrong output. If, however, a program is supposed to control the embedded computer of an airplane, it is much less clear what a failure might be. For an airplane as a system the notion of a failure is clear, but what are failures of components? Clearly if an engine explodes that will count as a failure, but for embedded programs the situation is less obvious. In the case of embedded software some failures may be caused by properties of the software while other failures may not be. Nevertheless, assuming the idea of an ALR fault, for embedded programs: whenever there is a notion of overall system failure and a conception of causality there is a derived notion of (embedded) program fault.

1.2.4 Semantic mismatch failure versus overall system failure

I assume that if a program fails it is in fact a related instruction sequence (the result of projection/compilation) the effectuation of which fails, either in theory or in practice. Now still there is a need to distinguish at least two cases:

(i) (Semantic mismatch failure) The failure is a mismatch between the intended semantics of the instruction sequence and its actual semantics. Here semantics is some mathematical or logical construct which abstracts from the inner working of the process of putting the instruction sequence into effect and

achieves some externally visible form of behaviour from it. A semantic mismatch failure can only be present if an intended semantics (often referred to as functional specification and referred to as a technical specification) in the introduction of Paragraph 1.2) is known or available.

(ii) (Overall system failure) The failure is a system failure of a system of which said instruction sequence is a component. For instance in [9, 10] two crashes of aircraft are discussed. In both cases a system in which a certain software component (in this case the implementation, say X, of the MCAS algorithm) plays a role, fails at a system level. Whether or not this failure has anything to do with a certain software component is not at all easy to assess. But if a local change of X creates a modified system which would not have crashed in a similar way, and which would not be more likely to crash under different conditions, then it is plausible to consider the program fragment which is to be modified as in X as a cause for the crash and to qualify it as a fault. The local change must solve a problem so as to obtain an improvement, that is without creating too many new problems.

1.2.5 System failures which cannot be caused by instruction sequence faults

Consider a car which is equipped with cameras at the front but not at the back. The car is, however, equipped with a program which provides automatic parking. I assume that the car is rather conventional and backward driving cannot be avoided when parking, at least in some cases. In principle that can be done: in forward position the car's eyes scan the scene, the car turns around and knows exactly how to park. Unfortunately and wrongly, however, during the parking process, a bike is placed by a careless child on the location where the car expects to get parked and the back of the car collides with the bike. Now a system level failure has arisen. One may ask if the implementation of the parking algorithm is at fault. Upon closer investigation one may conclude that in the absence of one or more cameras at the back of the car, and in the absence of other remote sensing capability at the back of the car, it is unreasonable to expect the algorithm to prevent this sort of collision.

In [40] the notion of an unsynthesizable specification is introduced and methods are proposed for the detection of such specifications in the context of robot programming. In general it may be too hard to determine whether or not a specification admits implementation by any other method than trying to implement it. In robotics, whether or not a specification can be implemented is not simply a matter of logic and mathematics because generic program notations will be used to control different robot embodiments for given tasks (see [1]). Unavoidably it will also depend on the actual robot embodiment being used whether or not a given goal can be reached.

Instead of the introduction of a program fault, or a program defect of a known kind, it may be the case that in some other way the program will not deliver as expected. Now it may be so that during the software process one or more of the steps have not been taken in accordance with the prescribed

software process. If that is the case a software process flaw has arisen (see [10] for that notion): it has not been noticed during the software development phase that the sensor hardware of the car is insufficiently powerful to create a context where a program can do the parking. The software development process should have been stopped, and not delivering a program for parking might have been best.

1.3 Research questions for the paper

The paper aims at providing a definition of a notion of algorithm and then working towards an understanding of notions of fault and defect for algorithms. Secondly an attempt is made to make sense of the notions "moral program defect" and "moral program fault."

Although I am discussing faults throughout the paper, which I understand in connection with single connected fragments being at fault, the discussion can be generalized to so-called multi-hunk faults (see [43]), which consist of a plurality of fragments each of which is supposed to be changed when repairing the fault. The notion of fault as used below is quite restrictive in comparison to its interpretation in works of fault or defect categorization (see e.g. [38]). The intended notion of fault is consistent with the notion of an ALR fault (as given in 1.2 above), and also with the faults which are targeted in spectrum based fault localization ([46, 47]).

2 Algorithms: a cognition based definition

Defining algorithms as a mathematical or logical notion has not been successfully done in the literature thus far. I will define algorithms in an informal manner, thereby implicitly assuming that a formal definition of algorithms cannot be given. I will refer to the definition of algorithm as given below as a cognition based definition. I have no doubt that other definitions of algorithm can be provided as well.

2.1 Tandem definition of algorithm and algorhyme

The above definition of a program is open ended in the sense that increasingly more developed and complex program notations allow writing a compiler (an algorhyme) for increasingly developed and complex program notations. Taking advantage of this definition of programs, it is possible to define algorithms in tandem with so-called algorhymes. Algorithms are cognitions while algorhrymes are artifacts (nowadays probably text in digital form).

Definition 2.1. *(Algorithm) An algorithm is an idea (a cognitive/mental concept) about how to perform a certain computation with an idea in mind of what it is supposed to achieve, both in terms of functionality and in terms of performance and resource utilisation. An algorithm is documented by a family of algorhymes, which collectively convey the underlying intuition/idea/cognition.*

In theory one would hope that an algorithm is an equivalence class of algorhymes, where algorhyme and a corresponding notion of equivalence are unambiguously defined. However there is no indication that a (pseudo) mathemathical definition of that form can be achieved. In practice at any moment of time there is a "flock" (i.e. a collective of entities of the same kind) of algorhymes which together constitute the current view of the algorithm, and algorhymes are regularly rewritten into new program/pseudocode notations in order to maximize appeal and accessibility.

An algorithm may be compared with a theorem with the different proofs being algorhymes. Perhaps more convincingly an algorithm may be compared with an invention and its document with a patent description of it. If the patent office rejects a patent proposal that means that another description of the same invention is contained in an existing and approved or pending patent or in known prior art. The very notion of novelty, with respect to prior art, only exists because the content of a proposed patent file is understood as being more abstract than any individual representation or documentation of it.

Definition 2.2. *(Algorhyme) An algorhyme is a program or pseudocode which is meaningful and comprehensible for a human reader and which is understood as the documentation of an algorithm. Two algorhymes are algorithmically equivalent if they document the same algorithm. Algorithmic equivalence is a matter of informal judgement by human agents rather than of formal definition.*

Algorithm and algorhyme are defined in tandem as these notions cannot be defined independently of one another.

A typical algorithm is C.A.R. Hoare's Quicksort. Quicksort is so to say a "conceptual average" of its documentations via programs and pseudocodes in the literature. The key idea is one-dimensional divide and conquer for a task which can be performed in a bottom up manner.

Unlike algorithms, algorhymes can be quantified in detail regarding various aspects of complexity and performance. Algorithmic equivalence is an informal notion which applies to algorhymes. Decisions about algorithmic equivalence are human decisions, which may be taken in court, rather than matters of logic and mathematics.

Definition 2.3. *(Algorithmic equivalence) Two algorhymes are algorithmically equivalent if both are documentations of the same algorithm.*

Algorithmic equivalence is defined on algorhymes understood as documentation of algorithms. Without a context in which algorhymes play the role of documenting an algorithm it is meaningless to speak of algorithmic equivalence. It is also not implied that if instruction sequences X and Y implement algorithmically equivalent algorhymes A and B respectively, a relation of algorithmic equivalence must hold between X and Y. It is plausible, however that algorithmic equivalence holds between the pairs (X, A) and (Y, B) which each may serve as documentations of the respective algorithms.

Definition 2.4. *(Algorhyme design) An algorhyme design is a design, the realisations of which are algorithmically equivalent algorhymes.*

In the definition of algorithm also algorhyme designs may be included, besides algorhymes, as possible documentations of algorithms.

Algorhymes are not merely specifications because algorhymes come with an awareness of "how to compute", which may but need not be the case for a specification. A specification may be inconsistent (asking to compute a value or pattern that does not exist) or may be impossible to implement (asking to determine information which cannot be found with the available tools), and such a state of affairs is manifestly inconceivable for an algorhyme.

2.2 Some comments on algorithm versus algorhyme

The difference between an algorithm and an algorhyme is as big as the difference between an invention and the textual description of the invention. For human readers two algorhymes may "clearly" describe the same method for solving a problem, the differences being only minor.

Minor differences between algorhymes A_1 and A_2 which supposedly document the same algorithm A, can take the following forms for instance (in an arbitrary order, the listing being far from complete): (i) use of different program notations, technical specification notations or pseudo-code notations, (ii) different ordering of initialisation of registers, different ordering of actions when independent steps are performed in some essentially arbitrary order, different naming of auxiliary registers, different naming of methods, classes, and of formal parameters, (iii) different ordering in the listing of classes, (iv) use of different libraries for essentially the same purpose, different levels of "outsourcing" of tasks to external libraries, (v) marginally different structure of the class hierarchy, (vi) use of iteration versus use of tail recursion, (vii) use of different data structures for the same purpose, where both may be helpful, (viii) use of different synchronisation primitives for concurrency, use of different message passing mechanisms, (ix) different degrees of algorithmic optimisation, (x) differences in exploitation of the presence of garbage collection, (xi) different levels of generality in terms of the ease of interfacing with other programs, (xii) different style of preparation for testing and/or different style of annotation for formal verification.

Both A_1 and A_2 unavoidably come with a significant amount of inessential detail, when understood as documenting A, and it is a matter for a human observer to assess when an accumulation of differences adds up to a real difference in terms of an algorithm being documented. By distinguishing algorithm from algorhyme Intellectual Property Right (IPR) protection of algorithms is completely disentangled from notions of copyrighting which prevail in the world of programs and which do apply to algorhymes.

2.2.1 Can an algorhyme contain a fault?

I will return to this question below in Section 4, but a general remark about this somewhat tricky question can be made here. A reasonable comparison is with mathematical proof. Then I compare an algorithm with the idea of a proof

and an algorhyme with a textual presentation of that idea, that is with what mathematicians usually call a proof. Is a proof still a proof if it contains a fault? For example the proof is not valid, as given, but can be turned into a valid proof by means of a minor textual change. We find that just as a proof can contain a fault an algorhyme can contain a fault, where it is essential that the fault comes with the possibility of local repair. If the proof is wrong in a more significant manner it contains a defect. Then the proof is not a proof anymore, and a new approach is needed (if the envisaged conclusion can be shown at all). So a proof cannot be defective without losing its status as a proof. The same situation applies to an algorhyme: once the conclusion has been drawn that "it is not what we want," the algorhyme ceases to serve as the documentation of an underlying idea. For historical purposes one may have an interest in the algorhyme as a "documentation" of a defective algorithm in order to prove in some later phase that it was not the case that an agent P knew of a certain algorithm at some stage. Such a conclusion may matter for instance in case of conflicts regarding IPR.

2.2.2 Can an algorhyme contain a fault?

Given an algorithm A and its collection of documenting algorhymes A_i. Suppose that upon further analysis it appears that for each of the algorhymes it is clear that the intended requirements specification will not be satisfied with an implementation of the algorhyme then the algorithm is defective.

Claiming that the algorhyme is faulty comes with a concept of repair of an algorhyme. In Section 4, I will discuss this matter in more detail. Making use of so-called reification of an algorithm by means of a flock of algorhymes it is meaningful to speak of a fault in an algorithm. Returning to the comparison between algorhyme and proof: a proof idea is reified by a flock (a finite collection) of proofs that follow the idea. It may turn out to be the case that each of these proofs contains a fault and that in a uniform manner each of the proofs can be locally repaired by changing a certain fragment in a fairly uniform manner. In this, admittedly unlikely, situation it is reasonable to say that the proof idea contains a fault. In a similar manner an algorithm may contain a fault if the algorithm is identified with, that is reified by, a finite collection of its purported algorhymes. If these are faulty and repairable in a uniform manner that state of affairs reveals a fault of the algorithm proper.

2.3 Algorithms and intellectual property rights

When contemplating intellectual property rights for computer software, such rights may concern at least these classes of entities: (i) computer software (ranging from source code, via intermediate notations to object code), (ii) computer programs, (iii) (computer) program documentation (including: informal, formal, or computed verifications; test suite reports; documented user experiences), (iv) algorhymes (including designs), (v) algorithms, (vi) specifications (both functional, non-functional and combined), (vii) requirements (that is specific col-

lections of requirements), (viii) software engineering methods, (ix) features of program notations, (x) software tools for software engineering.

IPR may be of several types including (a) copyright, (b) invention based patent, (c) design patent, and (d) trademark. The famous "Intel inside" notification indicates how trademarking may be of importance for computers and software. Each of these IPR types can be contemplated in a range of jurisdictions and legal systems, as well as in the light of a body of literature, and as concepts open for academic reflection from first principles.

Acquisition of IPR of a certain type for an entity of a certain class may be obtained under certain circumstances only, the details of which are sensitive to legal frameworks and jurisdictions. Nevertheless instantiations of the following criteria and phenomena play a role: (1) novelty, (2) (non)obviousness, (3) utility, (4) enabledness, (5) disclosure of enabledness, (6) scope, (7) prior art, (8) eligibility and statutory subject matter, (9) filing and granting, (10) licencing and use, (11) infringement and litigation, and (12) moral/ethical/societal justification or disqualification.

Not all combinations of entity class, IPR type and criterion/event are relevant. Seeking a design patent for compiled code is implausible, while using a trademark to highlight a key role in a piece of equipment of that same object code may be plausible. Having, filing or aspiring a design patent on a visual rendering of an algorhyme is not inconceivable. Patenting algorithms "as such" is often seen as lacking of moral/ethical/societal justification. Apparently a combinatorial explosion of combinations can be imagined and, upon being considered of potential relevance, investigated in further detail.

In the context of computer software, copyrighting is most common, with invention based patenting as a second IPR type but only marginal roles for the other types of IPR. In this section I provide some comment relating algorithms, algorhymes and programs to copyrights and patents.

2.4 Human judgement at different levels of abstraction

The definition of algorithm, supported by a definition of algorhyme, can be contemplated in the light of theory about copyrights and patents for software. These topics are complex, dependent on legal systems and traditions, and are to some extent controversial.

Nevertheless, the notion that deciding algorithmic equivalence is a matter of human judgement, as stated above, is unsurprising from the perspective of say patenting. Whether or not a proposed invention replicates acknowledged prior art is a human decision to be taken with the help of procedural protocols. Similarly whether or not a given device, process, or material constitutes an infringement of a given patent is a matter of human judgement.

In [37] it is shown how the classical theory of justice Learned Hand, regarding levels of abstraction, applies to software copyrighting. Similar to our description, control and access to data structures/data types (both encoded in services in our approach) provide levels of abstraction. Our definition of algorithm primarily aims at higher precision for control, not for the datatypes and datastructures.

2.4.1 Contrasting copyright and patent for software

Copyrighting and patenting are complementary, aiming at different levels of abstraction:

(i) Copyright may protect the expression of an algorithm (an algorhyme or a program implementing the algorithm) while not protecting the algorithm proper. Thus, an algorhyme P for algorithm A may infringe a certain copyright while an (algorithmically equivalent) algorhyme Q for A does not infringe the same copyright.

(ii) Copyright protection can only be provided for "expressions," embodyments, for which patent protection cannot be obtained (assuming a liberal patenting regime that tolerates software patents in principle). In [45] a formal criterion to that extent is formulated. In [25] the limitations of copyright protection against so-called cloning, a particular form of copyright infringement, are discussed in detail.

(iii) A software patent, assuming a jurisdiction that allows such patents to be granted at all, may protect an algorithm but cannot be made so specific that it only protects one or a few specific algorhymes for it. In [35] a patenting regime is indicated which may be permissive for software patents while avoiding well-known disadvantages by appropriate limitation of patent scope. Further detailing [35], in [42] it is proposed that scrutinising usefulness along several dimensions can help to overcome worries about the eligibility of patent claims, which otherwise might be considered "too abstract." These ideas can be incorporated in an IPR aware software engineering life-cycle as outlined in [11].

(iv) One notion of algorithm has an abstraction level comparable to invention in the context of patents. Here we use "invention" without any implicit condition of novelty, just as "discovery," in science, or "theorem," in mathematics. Novelty of an invention or algorithm can only be assessed at a certain moment in time. A "new" algorhyme may be novel for a known non-novel algorithm. A new algorhyme may also turn out to be not novel in the sense of copyright.

(v) Algorithms as defined above are ideal entities. In [31] it is argued that ideal entities are problematic subjects for the judgements which need to be made in the context of patents. However, it is suggested that such problems must not stand in the way of working towards the patenting of such abstract (in the sense of ideal, rather than in the sense of general) entities. In [32] these ideas are incorporated into a dedicated logic for reasoning about ideal entities.

(vi) Algorithmic equivalence of algorhymes is more restrictive than being semantic clones according to [41] which requires functional equivalence only. Extending the classification of forms of cloning of [41] one may speak of two algorhymes for the same algorithm as algorithmic clones.

(vii) Patenting algorithms "as such", that is without a clear claim about usability, is not supported by mainstream patent regimes. The proposal for the introduction of research patents as put forward in [44] stems from the background of chemistry where quite often much research may be required to establish the usefulness of a new chemical compound. A similar situation may arise with an algorithm as specified by its functionality combined with characteristics

of performance, compactness and limited resource utilisation. Usefulness may be discovered only later. Still allowing patent protection for the algorithm may be helpful for speeding up the development of software technology.

2.4.2 Algorithm patents: patenting algorithms "as such"

The counterpart of an invention for software is an algorithm. And with copyrighting as the primary, and well-established, means for IPR protection for software, it is plausible to contemplate algorithm patents rather than software patents, thereby removing any risk that the algorithm patent mistakenly protects the invention at hand at an inadequate level of abstraction. I believe that this interpretation of software patents, as being better understood as algorithm patents, complies with the proposals made in [39]. It comes with the concept of an algorithm that a person who knows and understands the algorithm must be able, in principle, to implement it. So an enablement criterion for algorithms must require more than mere implementation, for instance: compliance with a given functional specification, achieving certain performance qualities, admitting certain correctness guarantees, or achieving certain bounds on code compactness. Disclosure of enablement must indicate how an implementation of the suggested quality can be obtained.

For algorithm patents the "as such" aspect is of vital importance. Patenting algorithms "as such" would do without a proof of utility. An algorithm-as-such patent, if meaningful at all, may be understood as an instance of a research patent. Unlike in the case for chemical substances (as discussed in [44]), the potential economic benefits of the introduction of algorithm-as-such patents are not easy to grasp.

Assuming that an algorithm patent comes with a convincing demonstration of usability, then like in the case of chemical compounds the question is whether a patent granted with one application in mind would also cover, that is expose infringement by, another application of the same algorithm.

These matters merit further research. Whereas, in the case of algorithm patents, the notions of novelty and utility for an algorithm admit a convincing informal interpretation, disclosure of enablement and infringement are more problematic notions in need of further scrutiny. The following research question arises, irrespective of whether or not the introduction of algorithm patents is justified, productive or commercially meaningful.

Problem 2.1. *Provide a description of* algorithm patents*, understood as an instance of research patents (following [44]), with too narrow coverage prevented by making use of genus claims that allow some parts of an algorithm to be left undeterminate (see [30]), including one or more proposals on how these might be included in major patenting regimes and jurisdictions.*

As to the relevance of this question the following may be noticed. There is widespread scepticism about the justification for patenting algorithms "as such" (the well-known notion most close to algorithm patent). Now when opposing algorithm patents on principled grounds, one might not notice the difficulty of

establishing in plausible detail such a new kind of patents, including a role for these in an IPR aware software engineering life-cycle, see [11] for that notion, when first leaving the principled objections aside. Raising principled objections against implausible proposals, however, is needless and unconvincing.

2.5 Services: servirithms versus servirhymes

For services one may distinguish three aspects: (i) theoretical, logical, mathematical or formal specifications of service behaviour, (ii) physical realisations including descriptions thereof, (iii) (human) cognitions which capture a service in such a way that an understanding results (by one or more human agents) which supports the design (by said agents) of instruction sequences making use of the service.

In line with the terminology of algorithms and algorhymes, I propose to refer to descriptions under (i) as servirhymes, and to the cognitions under (iii) as servirithms. Under two one finds physical services and descriptions thereof. The additional terminology involving servirithms and servirhymes allows conceptual clarity concerning services. For instance one may speak of service H as "an array of 10 000 bits" as a servirihtm, which on closer inspection calls for a plurality of realising servirhymes. Servirithm H and servirhyme H may both share the same method interface I_H so that instruction sequence design, with the intention to make use of H, can be performed.

2.5.1 Servirhyme faults

Once a service has been specified its realisation may deviate from the specification. Clearly services may feature faults and defects as much as instruction sequences do. However, no fixed format for the description of services exists and the development of any theory of faults and defects for services requires additional choices to be made which I will refrain from doing in this paper.

2.6 Issues of terminology: designer algorithms

There is a societal desire to use algorithm with a very wide scope of meaning. For instance machine learning may create a program able to perform a classification task. One may wish to speak of a learnt algorithm in that case. I would prefer to speak of a learnt program instead. Nevertheless when insisting to speak of a learnt algorithm one will notice that a learnt algorithm does not qualify as an algorithm in the sense of Definition 2.1 above. A learnt algorithm, as just mentioned, is not a cognitive entity. It is not an idea. Now one might proceed in different ways:

(i) speak of a learnt program instead of a learnt algorithm, so that the requirement that the entity produced by the machine learning application is a cognitive entity, an idea, need not apply to it.

(ii) rename algorithm as defined in 2.1 above into "designer algorithm" thereby admitting that non-designer algorithms exist, and in fact may be programs.

However, I would expect that working towards patent like IPR for a "learnt algorithm" is not plausible. From an IPR perspective a learnt algorithm behaves like a program. The learning algorithm, an implementation of which converts a set of qualified examples into a learnt program, may well deserve its name, that is be regarded as a designer algorithm, and may for that reason be susceptible for IPR beyond copyrighting.

3 Algorhyme: capturing the concept in more detail

In this section I will first expand on the services on which the definition of an algorithm is based. In particular I will survey a notion of interface which is helpful for conceptualisation of the given definitions of program, algorhyme and algorithm.

I have made the assumption that an algorhyme is comprehensible for some human readers, where some form of "being skilled in the art" just as for patent descriptions may be needed to characterise those readers who might actually be able and willing to acquire said comprehension. Algorhyme may be understood as an expression possibly in an ad hoc and single purpose extension of some "known" program notation. However, if the collection of basic actions in (f.m focus-method notation) is a mere set without any underlying intuitive meaning then it is hard to imagine how any human being could arrive at comprehension of an instruction sequence, program, or algorhyme written in terms of those primitives.

3.1 A service basis

At the basis of interfaces for an instruction sequence stands the library of services. The notion of a service is informal, except for the qualitative description of functionality in terms of method calls, single bit (Boolean) reply to a caller, and state transformation. The informal aspects of services are critical for human understanding.

- Name: an alphanumerical string starting with a letter. Here the name is referred to as `Xsv`
- A list of method names which together constitute the method interface $\mathtt{I}_{\mathtt{method}}(\mathtt{Xsv})$ of `Xsv`. Alternatively a reference to a description of $\mathtt{I}_{\mathtt{method}}(\mathtt{Xsv})$ can be provided.

- intended use:
  - theoretical work,
  - cases studies (with implementation),
  - experimental design,
  - practice
- category of service indicator:
  - RFS (for Reply Function Service),
  - stateful service,
  - stateful service with external interactive behaviour,
  - stateful service with external interactive behaviour given with relevant precision concerning discrete time behaviour,
  - description of initial states when serving as a parameter.
- Features and peculiarities
  - (description of) parameters, bounds etc, e.g. the maximum size of a stack etc.,
  - indication how names will be tagged with identifiers for individual instances of a service,
  - authorship and IPR aspects, (there may be several sources, authors and parties entitled to various forms of formal and informal IPR for a single service); this aspect includes that history of the service,
  - computable or non-computable,
  - description optimised for: readability, match with certain formats, verification and testing,
  - enhancement extension of earlier services,
  - special aspects: probabilistic, use of prospecting, use of lookback.
- familiarity indicator
  - math/logical-imported-standard (amply known definitions, available from a variety of sources)
  - math/logical-imported-pseudo_standard (imported from a source where rigorous definitions are given, with an intention to find stable forms for further use)
  - math/logical-imported-incidental (imported from a source where rigorous definitions are given though not with an objective of use in other contexts, but aware of generic significance)
  - math/logical-imported-ad_hoc (imported from a source where rigorous definitions are given , though aware of the fact that reuse elsewhere is unlikely)

  - math/logical-recent (specified in a recent source, not used elsewhere)
  - informally outlined (to be made more precise during design)
  - informally outlined (not to be detailed any further)
  - open (name only)

- Depending on the category a more or less rigorous specification, or reference to such a specification of the following aspects, when in order:
  - reply function: $\mathtt{I}^{+}_{\mathtt{method(Xsv)}} \rightarrow \{\mathtt{true}, \mathtt{false}\}$,
  - state space,
  - initial state,
  - reply condition function ($\theta(\mathtt{s}, \mathtt{m})$),
  - effect function ($\eta(\mathtt{s}, \mathtt{m}, \mathtt{b})$),
  - process description of external behaviour

A reader of an instruction sequence can only make sense of that text when aware of the structure, meaning and details of various services. It will depend from person to person how much knowledge is needed but together it is quite a lot.

This matter is may be clarified with an indication of the classification of various services which have been used in research work until now.

1. `br` stands for bit register (or Boolean register). A collection of 16 methods with "standard" notation has been proposed in [22]. `br` is math/logical-imported-standard,

2. services for stack queue, Turing tape, both with bounded size and with infinite size" math/logical-imported-pseudo_standard (though naming of methods and precise meaning of these may vary for source to source),

3. Data linkages ([16, 19]) used for the specification of garbage collection and shedding, math/logical-imported-incidental

4. various Maurer machines used in the theory of processor specification: math/logical-imported-ad_hoc

With an ambition to discuss complex examples, comes the need to have more complex services in stock. For modelling real time control systems (as specified in discrete time) comes the need for a library of services able to interact directly via synchronous or asynchronous communication; such definitions are currently unavailable. The library of services is not meant to be stable and rigid, rather it will be updated and upgraded in a steady manner.

At any instant of time a service basis provides authors and readers of instruction sequences with (i) names for services, names for methods, a (iii) method interface $\mathtt{I}_{\mathtt{method}}(\mathtt{H})$ for each service, and (iv) adequate information for obtaining an intuitive understanding of these ingredients.

3.2 Interfaces: required interfaces and provided interfaces

Names of services as given in the service basis may be unwieldy and for that reason it is plausible that a project makes use of simplified and local/temporary/ project_bound service names.

For a project $\mathtt{p}$, a temporary service naming scheme $\mathtt{TSN_p}$ is a partial function from names in a name space to service names. This function may change through time during a project, at time there is $\mathtt{TSN_p^t}$. At time $\mathtt{t}$, $\mathtt{Dom(TSN_p^t)}$ constitutes the collection of service names of relevance for project $\mathtt{p}$, with method interfaces as follows: $\mathtt{I_{method}(V) = I_{method}(TSN_p^t(V))}$. A software engineer in project $\mathtt{p}$ will use names in $\mathtt{Dom(TSN_p^t)}$ all other information regarding the service basis is considered background information.

For an instruction sequence or a thread to make use of a method call $\mathtt{f.m}$ during a computation it is needed that it is known which service (say $\mathtt{H}$) is supposed to be accessible via focus $\mathtt{f}$ and then $\mathtt{m} \in \mathtt{I_{method}(H)}$ is required. Such information is given by a service family, which is an expression of the form $\mathtt{f_1.H_1} \oplus \ldots \oplus \mathtt{f_n.H_n}$.

Various notions of interface are relative to a project $\mathtt{p}$ at a time, say $\mathtt{t}$. Interfaces are defined relative to an focus/service linking. A focus/method interface is a collection of focus method pairs $\{\mathtt{f_1.m_1}, \mathtt{f_2.m_2}, ..\}$. A service family $\mathtt{W}$ comes with a provided interface $\mathtt{I_{provided}(W)}$ which is a focus/method interface.

Instruction sequences and threads work in the context of a service family and require that method calls are processed. Service families on the other hand offer options for processing method calls. An instruction sequence $\mathtt{X}$ comes with its required focus/method interface $\mathtt{I_{required}(X)}$, and a thread $\mathtt{P}$ comes with a required focus/method interface $\mathtt{I_{required}(P)}$ as well. If $\mathtt{P}$ results from $\mathtt{X}$ by thread extraction, i.e. $\mathtt{P} = |\mathtt{X}|$, then $\mathtt{I_{required}(P)} \subseteq \mathtt{I_{required}(X)}$.

The various operators are connected via the equations listed in Tables 1 and 2.

3.3 Algorithm reification with a flock of algorhymes

Viewing an algorhyme as a documentation of an algorithm is not the same as viewing it as a textual or physical form of it. I will speak of reification of an algorithm when the algorithm takes a material or somehow concrete form. An appropriate idea of algorithm reification is as follows:

Definition 3.1. *(Reificiation of algorithm) A reification of an algorithm consists of a reasonably complete collection (flock) of documenting algorhymes for it.*

A reification of an algorithm is reasonably complete if it provides an adequate overview of the various existing documenting algorhymes for it. Reasonable completeness is an informal notion to be assessed by human judgement and for that reason amenable to disagreement.

Conceptual proposition 3.1. *(Algorithms are not programs) It is not the case (as is often claimed) that an algorithm is a method in the form of a sequence*

$$\mathtt{x} \cup \emptyset_{\mathtt{interface}} = \mathtt{x} \tag{1}$$
$$\mathtt{x} \cup \mathtt{y} = y \cup \mathtt{x} \tag{2}$$
$$\mathtt{x} \cup (\mathtt{y} \cup \mathtt{z}) = (\mathtt{x} \cup \mathtt{y}) \cup \mathtt{z} \tag{3}$$
$$\mathtt{x} \cup \mathtt{x} = \mathtt{x} \tag{4}$$
$$\mathtt{f}.\{\mathtt{m}\} = \{\mathtt{f}.\mathtt{m}\} \tag{5}$$
$$\mathtt{f}.\emptyset_{\mathtt{method-interface}} = \emptyset_{\mathtt{interface}} \tag{6}$$
$$\mathtt{f}.(\mathtt{h} \cup \mathtt{u}) = \mathtt{f}.\mathtt{h} \cup \mathtt{f}.\mathtt{u} \tag{7}$$
$$\mathtt{H} \oplus \mathtt{0}_{\mathtt{service-family}} = \mathtt{H} \tag{8}$$
$$\mathtt{H} \oplus \mathtt{L} = \mathtt{L} \oplus \mathtt{H} \tag{9}$$
$$\mathtt{H} \oplus (\mathtt{K} \oplus \mathtt{L}) = (\mathtt{H} \oplus \mathtt{K}) \oplus \mathtt{L} \tag{10}$$
$$\partial_{\emptyset_{\mathtt{focus-collection}}}(\mathtt{H}) = \mathtt{H} \tag{11}$$
$$\partial_{\mathtt{U}}(\mathtt{0}_{\mathtt{service-family}}) = \mathtt{0}_{\mathtt{service-family}} \tag{12}$$
$$\partial_{\{\mathtt{f}\}}(\mathtt{f}.\mathtt{R}) = \mathtt{0}_{\mathtt{service-family}} \tag{13}$$
$$\mathtt{f} \neq \mathtt{g} \rightarrow \partial_{\{\mathtt{f}\}}(\mathtt{g}.\mathtt{R}) = \mathtt{g}.\mathtt{R} \tag{14}$$
$$\partial_{\{\mathtt{f}\}}(\mathtt{H} \oplus \mathtt{K}) = \partial_{\{\mathtt{f}\}}(\mathtt{H}) \oplus \partial_{\{\mathtt{f}\}}(\mathtt{K}) \tag{15}$$
$$\partial_{\mathtt{U} \cup \mathtt{V}}(\mathtt{H}) = \partial_{\mathtt{U}} \circ \partial_{\mathtt{V}}(\mathtt{H}) \tag{16}$$
$$\mathtt{f}.\mathtt{H} \oplus \mathtt{f}.\mathtt{K} = \mathtt{f}.\mathtt{0}_{\mathtt{service}} \tag{17}$$
$$\mathtt{I}_{\mathtt{method}}(\mathtt{0}_{\mathtt{service}}) = \emptyset_{\mathtt{method-interface}} \tag{18}$$
$$\mathtt{I}_{\mathtt{method}}(\mathtt{H}) = \{\mathtt{m} \mid \mathtt{m}\ \mathtt{is\ a\ method\ of}\ \mathtt{H}\} \tag{19}$$
$$\mathtt{I}_{\mathtt{provided}}(\mathtt{0}_{\mathtt{service-family}}) = \emptyset_{\mathtt{interface}} \tag{20}$$
$$\mathtt{I}_{\mathtt{provided}}(\mathtt{f}.\mathtt{H}) = \mathtt{f}.\mathtt{I}_{\mathtt{method}}(\mathtt{H}) \tag{21}$$
$$\mathtt{I}_{\mathtt{provided}}(\mathtt{f}.\mathtt{H} \oplus \partial_{\{\mathtt{f}\}}(\mathtt{K})) = \mathtt{I}_{\mathtt{provided}}(\mathtt{f}.\mathtt{H}) \cup \mathtt{I}_{\mathtt{provided}}(\partial_{\{\mathtt{f}\}}(\mathtt{K})) \tag{22}$$
$$\mathtt{I}_{\mathtt{focus-collection}}(\mathtt{0}_{\mathtt{service-family}}) = \emptyset_{\mathtt{focus-collection}} \tag{23}$$
$$\mathtt{I}_{\mathtt{focus-collection}}(\mathtt{f}.\mathtt{H}) = \{\mathtt{f}\} \tag{24}$$
$$\mathtt{I}_{\mathtt{focus-collection}}(\mathtt{V} \oplus \mathtt{W}) = \mathtt{I}_{\mathtt{focus-collection}}(\mathtt{V}) \cup \mathtt{I}_{\mathtt{focus-collection}}(\mathtt{W}) \tag{25}$$

$$\mathtt{x} \subseteq \mathtt{y} \iff \mathtt{x} \cup \mathtt{y} = \mathtt{y} \tag{26}$$

Table 1: Equations for provided interfaces

$$\mathtt{I}_{\mathtt{required}}(\mathtt{S}) = \emptyset_{\mathtt{interface}} \tag{27}$$
$$\mathtt{I}_{\mathtt{required}}(\mathtt{D}) = \emptyset_{\mathtt{interface}} \tag{28}$$
$$\mathtt{I}_{\mathtt{required}}(\mathtt{P} \trianglelefteq \mathtt{f.m} \trianglerighteq \mathtt{Q}) = \mathtt{I}_{\mathtt{required}}(\mathtt{P}) \cup \{\mathtt{f.m}\} \cup \mathtt{I}_{\mathtt{required}}(\mathtt{Q}) \tag{29}$$

$$\mathtt{I}_{\mathtt{required}}(!) = \emptyset_{\mathtt{interface}} \tag{30}$$
$$\mathtt{I}_{\mathtt{required}}(\#\mathtt{n}) = \mathtt{I}_{\mathtt{required}}(\backslash\#\mathtt{n}) = \emptyset_{\mathtt{interface}} \tag{31}$$
$$\mathtt{I}_{\mathtt{required}}(\mathtt{f.m}) = \{\mathtt{f.m}\} \tag{32}$$
$$\mathtt{I}_{\mathtt{required}}(+\mathtt{f.m}) = \mathtt{I}_{\mathtt{required}}(-\mathtt{f.m}) = \{\mathtt{f.m}\} \tag{33}$$
$$\mathtt{I}_{\mathtt{required}}(\mathtt{X};\mathtt{Y}) = \mathtt{I}_{\mathtt{required}}(\mathtt{X}) \cup \mathtt{I}_{\mathtt{required}}(\mathtt{Y}) \tag{34}$$
$$\mathtt{I}_{\mathtt{required}}(\mathtt{X}^{\omega}) = \mathtt{I}_{\mathtt{required}}(\mathtt{X}) \tag{35}$$

Table 2: Equations for the required interfaces of threads and instruction sequences

of steps, or a mechanism for generating sequences of steps. Like the invention of glasfiber connection cannot transmit light, only realisations of that invention can do the real work, it is up to the implementations of algorhymes to be useful in practice.

Upon being proposed, say in a research paper, an algorithm is documented by a single algorhyme as given in that paper, i.e. a singleton algorhyme flock. It is tempting to identify the algorithm with that single algorhyme but doing so is just as implausible as identifying a theorem with the first proof which has been found for it. Nevertheless it is valid for the author to claim that an algorithm has been discovered.

Conceptual proposition 3.2. *(Expanding universe of algorithms) The stock of algorithms, and of corresponding algorhymes is steadily developing and extending. Stronger program notations allow more complex algorithms to be documented (by more expressive algorhymes). The development of algorithms cannot be understood independently from the development of program notations, and in practice not independently from the development of underlying technology. Unavoidably algorhymes will make use of implementations of pre-existing algorithms as building blocks. Algorithms breed algorithms so to say.*

I propose not to maintain a notion of equivalence between algorithms. Algorithms are algorhymes modulo algorithmic equivalence, although this is the case merely in an informal sense. Maintaining second order algorithmic equivalence seems to serve no purpose in this approach.

3.4 Implementation of algorithms

With implementation I will refer to to the result of implementation rather than to the act of implementing.

Definition 3.2. *(Implementing an algorithm) An implementation of an algorithm is an implementation of any of its (up to date) algorhymes. When implementing an algorhyme the work involved may vary from using a compiler, via compilation by hand, to a more informal design process taking the algorhyme as a starting point. An implementation is usually a program or an instruction sequence but it may be a piece of hardware just as well.*

An implemented algorithm can be used, by being put into effect. An algorhyme can be used by being copied in whole or in part or after some revision into an algorhyme for a different algorithm.

Definition 3.3. *(Use of an algorithm) There are three ways for an algorithm to be used:*

(i) Operational use of an algorithm: if a system contains an implementation of an algorhyme that documents an algorithm then the system is said to make operational use of said algorithm.

(ii) Developmental use of an algorithm: if a programmer implements an algorithm they are said to make use of it by way of program development.

(iii) Conceptual use of an algorithm: if one or more documenting algorhymes for an algorithm are used for the design of a new algorithm, or for an ad hoc designed application program, see below, then conceptual use is made of it.

3.5 Intra-computer science versus extra-computer science

Many if not most algorithms are intrinsic computer science. This means that a definition in terms of logic, mathematics and theoretical computer science can be given of the expected behaviour, both in terms of functionality and in terms of resource utilisation. Such algorithms may be called intra-Computer Science algorithms (intra-CS algorithms). Other algorithms may import significant knowledge or expertise from outside computer science, and are called extra-Computer Science algorithms (extra-CS algorithms). A chess program which uses a large database of known chess games by human players implements an extra-CS algorithm, while a self-learning chess program which creates all relevant knowledge by itself is an intra-CS program.

The distinction between intra-CS algorithms and extra-CS algorithms is far from unproblematic. I have the impression that the number of extra-CS algorithms is very much on the rise nowadaways, while at the same time Artificial Intelligence (AI) techniques steadily deliver new ways in which extra-CS algorithms can be replaced by programs that are found as outputs of implementations of intra-CS algorithms. AI expands the scope of CS, and human designers expand the scope of algorithms, and which of the two processes will outrun the other, if such a thing happens at all, is unknown at the moment.

Definition 3.4. *(Moral neutrality of intra-computer science algorithms.) Intra-CS algorithms are morally neutral, that is neither good nor bad.*

Definition 3.5. *(Moral quality of extra-computer science algorithms.) Extra-CS algorithms may have a moral quality – good or bad.*

For instance, a program which maintains a database with data which ought to be thrown away in some cases may be considered morally problematic.

Definition 3.6. *(Ad hoc designs) An ad hoc design is a design which is so particular for a certain application context that it is rather unlikely that a plurality of alternative documenting algorhymes will ever be made for it, thereby facilitating the gradual of abstraction (that comes with an increasing number of available documenting algorhymes) which is needed to bridge the gap between a design or a program and an algorithm. Ad hoc designs are programs which hardly transcend beyond their textual appearance.*

Ad hoc designs may contain algorhymes as components. The simplest notion of program component in the setting of instruction sequence theory is given by so-called polyadic instruction sequences from [17]. Notions of failure, fault and defect for ad hoc designs are simply inherited from the corresponding notions of programs.

4 Faults and defects of instruction sequences, programs, and algorhymes

For an algorhryme various notions of fault can be introduced just as was done for instruction sequences in [6]. Here I assume that with the notion of an algorhyme comes, by necessity, an idea of specification or at least of requirements: what behaviour is supposed to result from running the algorhyme, or rather from running an implementation of it, an implementation which has been obtained in a deterministic manner by way of compilation, which is a transformation which in principle may be carried out by hand.

Definition 4.1. *(WB SRT algorhyme fault; abbreviations used for White Box and for Successful Regression Test.) Let V be a collection of tests which have been successfully passed by algorhyme X, then X contains a WB SRT fault if a known change Y_c can replace a fragment Y of X and a change Y_c for Y in such a way that:*

(i) there is a known input s_{psf} (for Primary Symptomatic Failure) on which X fails,

(ii) obtaining X_c by replacing Y in X by Y_c it is the case that X_c works correctly on s_{psf} (the problem of the primary symptomatic failure has been resolved), and

(iii) X_c passes all tests in V, i.e. X_c successfully completes the regression test.

In the above notation if X is improved and the next phase of the development of X becomes X_c then the regression test set is then extended with s_{psf}, i.e. $V \Rightarrow V \cup \{s_{psf}\}$.

A BB (for Black Box) SRT fault is defined similarly but now only the fragment Y is known while of s_{psf} and of Y_c only the existence is known, and in fact many choices for both s_{psf} and of Y_c may exist. GB (Grey Box) versions of SRT faults may provide either input for a primary symptomatic failure, or a proposed change.

Definition 4.2. *(MFJ algorhyme fault) A WB MFJ fault (MFJ for Mili, Frias and Jaoua) is defined in the same way as a WB SRT fault with the difference that condition (iii) is changed as follows: X_c produces correct results on all inputs where X produces correct results.*

Definition 4.3. *(Laski algorhyme fault) A WB Laski fault is defined in the same way as a WB MFJ fault with the difference that condition (iii) is changed as follows: X_c produces correct results on all inputs.*

According to these definitions each Laski fault is also an MFJ fault and each MFJ fault is also an SRT fault. BB and GB versions of MFJ faults and of Laski faults are defined accordingly. In order to become technically non-trivial these definitions need to be augmented with size bounds on the size of Y and of Y_c. I will not discuss such matters here, and I refer to [6, 7] for more precision on sizes of fragments in these definitions.

When programs are taken for instruction sequences, the number of instructions, LLOC, (for Logical Lines of Code following the notation of [4]) can be used for the quantification of the size of fragments, candidate faults, and of candidate changes thereof. Many variations of the notion of a fault in an algorhyme can be imagined, but I hope that these definitions suffice for the work in this paper.

4.1 Defective algorhymes

An algorithm is defective if it cannot serve as documentation for an algorithm for which it is intended to play that role while at the same time it cannot be considered faulty because limited local changes do not suffice to obtained a proper documentation. An instance of a defective algorhyme is an algorhyme which claims to document a well-known algorithm, for example in lecture notes, while it is plainly wrong. I find it hard to imagine an algorhryme which is defective, though not faulty, and which is still credible, that is not plainly wrong.

4.2 Faulty algorithms?

There is a gap between the notion of a faulty algorithm and a faulty algorhyme. A faulty algorhyme may, for that reason, not constitute an adequate documentation for an algorhyme. This may be apparent if various other algorhymes

documenting the same algorithm are known which don't feature the same defect. It may also be the case that an algorhryme features a defect which sheds doubt on the algorithm itself: the idea of the algorithm may be flawed in that there is no indication that a method for solving a certain problem, with certain quality attributes, has been discovered. This theme is continued in Section 5.

4.3 Under/overperformance defects of programs and algorhymes

I will include correctness in the notion of performance so a failure is an instance of bad performance. I will count too slow computation and too high use of resources also as defective performance. The latter defects only exist in a context where specifications are given which indicate what must be achieved.

4.3.1 Hash functions, an example

In [20] the hash function SHA-256 is specified by way of an instruction sequence for its computation. This text may be considered one of many algorhymes for SHA-256. In connection with faults and defects the following remarks can be made:

(i) It is certainly possible that there is something wrong with the instruction sequence given in [20], in the sense that on some inputs a failure occurs with respect to the official definition of SHA-256. That would be a DOA (Deviation Of Algorithm) defect of this algorhyme (see also 4.4 below). If it is a local defect allowing resolution by way of a local change it is a fault of some kind.

(ii) It is also possible that there is something wrong with the SHA-256 algorithm itself. The absence of collisions is a mere act of faith, though supported by a massive amount of experience. But if it should become possible to easily generate collision pairs one would start to consider the SHA-algorithm as being defective by underperforming. Underperfomance understood as the inability to defeat attempts to create collision pairs.

(iii) It is somewhat unlikely that if an underperformance defect were found this problem can be resolved by a local replacement of a fragment of the text in [20]. But in theory the future detection of such a fault cannot be excluded.

4.3.2 Moral underperformance defects in the context of humanoid autonomous robots

One can imagine a robot, e.g. a humanoid robot $R_{AB(X)}$ equipped with dangerous equipment (see [5] for some reflections on that), with $AB(X)$ representing its artificial brain, essentially made up of a platform for running an instruction sequence X, which in turn implements an algorhyme A. Now one may assume that due to a lack of problem solving competence, that is a lack of quality of its artificial intelligence, $AB(X)$ solves a class C of practical problems in a more destructive manner than would be achievable with a more appropriate instruc-

tion sequence X' instead of X, that is $R_{AB(X')}$ performs adequately on tasks in C. In this case X has a repairable underperformance defect.

If, however, $R_{AB(X)}$ performs badly in comparison to competent human performance concerning tasks of class C and it is unknown how to improve X so that adequate performance is obtained, then X features a non-repairable underperformance defect. With the improvement of software technology a non-repairable underperformance defect may become a repairable underperformance defect. If a moral underperformance defect has result from technological progress then repairing the defect, upon that having become doable may be considered a moral duty for maintenance of X. Failing to resolve the defect may be considered a moral maintenance flaw with respect to the maintenance of X.

4.3.3 Overperformance defects of algorhymes

An overperformance defect arises if an algorhyme shows a capability which is problematic as such. From a computer science perspective overperformance is not a well-known kind of problem, and is not considered to be a big deal for that reason. However, in principle it may arise. For instance one may imagine bit sequences which encode a collection of digital images of the same person. Now an algorhyme or rather an implementation of it, may be able to figure out from these data aspects of a person's state of mind, or ethnic background, or age or whatever matter which is best kept private in principle. In this case the algorhyme shows objectionable overperformance. In other words the algorhyme is morally defective by being capable of successfully performing one or more tasks which should not even be tried to perform – not bay an algorithm and not by any human agent.

The kind of defect just mentioned is an absolute overperformance defect. In no context would the very existence of an algorhyme with said performance be unproblematic.

4.3.4 AI competence overperformance defects: artificial stupidity as the solution

Moral overperformance defects occur in several forms:

(i) Absolute overperformance as discussed above. The negative moral assessment is hardly bound to AI, and supposedly holds for all agents.

(ii) Competence overperformance defects. Imagine a nuclear missile with artificial brains which allow it to redirect its targets in flight depending on its analysis of the political situation. It seems reasonable to assume that this situation is undesirable by any means. Given the power the embedded computer has regarding physical control, a system which starts figuring out by itself where to deliver its payload is unacceptable beyond a certain level of destructive power. A high level of intelligence may be used to reach the given target with high precision and reliability, whereas that same intelligence may not be used to intentionally deviate from the target.

If too much artificial intelligence is the problem, artificial stupidity is the solution. The embedded system must demonstrably restrict the reach of its influence, and of the optimisation it is working out.

(iii) Pseudo competence overperformance defects. A competence overperformance defect may have been intentionally made invisible with a minor change in an algorithm which prevents the relatively high competence from taking effect on the course of events. However, in such a case a minor change of the algorhyme will make it document an algorithm that upon having been implemented will feature competence overperformance. An occurrence of this case can be qualified as a pseudo competence overperformance defect.

(iv) Technical overperfomance defects. A technical overperformance defect arises if a system is able to optimise its behaviour in clever ways, though misses out on the adequate assessment of the consequences of its actions. The control intelligence features an overperformance defect. Rather than introducing artificial stupidity, however, improved assessments of the expected consequences of various actions may be introduced which allows the system to use its intelligence and at the same time to find out when certain optimisations go too far.

4.3.5 Moral defects of an algorhyme: who is to blame?

Suppose that one agrees that a specific algorhyme A features a moral defect. Now if human agent P knows about the moral defect and uses an implementation P_A of A in critical circumstances, that is conditions under which the overperformance under consideration is likely or merely able of causing a wrong outcome, then A acts immorally by making use of a robot that has P_A embedded in its software. It is less clear to what extent the designer of A is to blame. I hold that it is conceivable that the designer of X has acted in a morally objectionable manner simply by designing X. How and when that is the case is non-obv ious and merits further research.

4.3.6 Moral overperformance defects cannot be faults at the same time

Suppose that a moral overperformance defect is observed for algorhyme X. Then it is plausible that a minor modification "breaks the code" and leads to an algorhyme without any significant performance let alone overperformance. Even if the residual performance is such that the change involved might be seen as a solution of the moral overperformance defect it is not the case that the resulting algorhyme is without any moral overperformance defect. Indeed it is merely one minor change, in a backward direction, away from an algorhyme that features a moral overperformance defect and for that reason the algorhyme shows this defect itself by definition as a case of pseudo overperfpormance defect.

4.4 DOA defects for algorhymes

Suppose that X is an algorhyme in a flock F of $n+1$ algorhymes for an algorithm A with $n >> 1$. Now it may be the case that on closer inspection except X all other algorhymes in F do qualify as equivalent documentations for the same algorithm, while X is an outlier which, although it has the same functionality, that is semantics, as the other elements of F, it can not reasonably count as a documentation for "the same method for computing a certain functionality" as the other algorhymes do. Moreover assume that a fragment Y is found in X, together with a change Y_c so that upon replacing Y by Y_c in X a credible element of F results which can be positively assessed as documenting the same computational method as the other elements of F do. Now a defect in X has been found. This defect is a deviation of algorithm (DOA) defect.

5 Failures, faults, and defects for algorithms

The initial idea for a defect of an algorithm is as follows;

Definition 5.1. *(Algorithm defect.) If each of the algorhymes documenting algorithm A contains the same or a very similar defect then that defect is also considered a defect of the algorithm.*

In practice the above definition of an algorithm is unworkable because quantification over all documenting algorhymes is undoable. By working with algorithm reifications, that is with finite algorhyme flocks more workable definition of fault and defect can be obtained.

5.1 Faults of algorithm reifications

Algorithms as defined in 2.1 above are not sufficiently concrete to contain faults. There is no plausible notion of component or fragment of an algorithm and by consequence there is no plausible notion of replacement of fragments of an algorithm. However, when considering reifications of algorithms by flocks of algorhymes the matter changes.

Let F_A be a flock of algorhymes for algorithm A, and assume that none of the algorhymes in F_A features a DOA defect. Now it is quite possible that none of the algorhymes in the flock complies with the requirements R which are supposed to be met by these. Non-compliance with requirements may take the form of semantic mismatch failures or it may take the form of overall system failures. This distinction was mentioned above in 1.2.4.

Now it is conceivable that in a systematic way each algorhyme X in F_A can be upgraded by replacing a fragment in it so as to be improved in such a way that the failure withe respect to requirements F is resolved. If these replacements are somehow similar, that is if a replacement for one of the algorhymes is known, replacements for the others can be derived from that, one may speak of a fault in each of the algorhymes in the flock which at the same time counts as a fault in the algorithm which is reified by the flock.

The mere option for a uniform, for all algorhymes in the flock, and pairwise mutually comparable, inter-derivable, improvement which resolves a failure does witness a fault. It is plausible to apply notions like SRT fault, MFJ fault, and Laski fault to algorithm reifications.

In the same way a notion of fault may sometimes be lifted from algorhymes to algorithm reifications, various notions of defect may be lifted from algorhymes to algorithms. Thus algorithm reifications may feature various moral defects including moral overperformance defects and moral underperformance defects.

5.2 (Im)plausibility of faults of algorithm reifications

The very presence of a substantial flock of algorhymes for an algorithm indicates that the algorithm is successful in practice. At the same time it indicates that it somehow works and for that reason is not faulty.

In the area of cryptology algorithms are used which have become widespread because of success, and for which by consequence a significant family of documenting algorhymes has evolved which together constitutes a reification.

At any time however, it may turn out that the algorithm is broken, for instance in such a manner that the keys it computes can be deciphered, and is not considered safe any more so that from that moment onwards the algorithm is considered defective with respect to its original objectives.

Now my observation would be that in such cases it is unlikely that the defect amounts to a fault, that is a defect with an easy solution via a small textual change. Claiming the necessity of a major redesign of the algorithm and its plurality of documenting algorhymes constitutes more plausible diagnosis of the situation.

5.3 Algorithms for embedded systems

I imagine a platform design P equipped with an Operating System (OS) able to control all its physical functions. The OS is supposed to be generic, that is it is independent of the intended use and is capable of being installed on many different platform designs. Instances of P, that is physical platforms S_i^P, called P-systems, of which a plurality may be around at some instance in time, are to be loaded with control software consisting of a number of applications $X_1, \ldots, X_n$ running as a multithread by way of strategic interleaving (see for example [13, 14, 15]), in addition to a number of utilities $X_{n+1}, \ldots, X_{n+m}$. I will write $S_i^P(\vec{X})$ for the system with these particular software components loaded, and active in the manner just specified.

The main source of variation between P-systems lies in the software which is in control of their actions. I will formulate a number of assumptions about the software.

Assumption 5.1. *(InSeq design in the lead.) $X_1, \ldots, X_n$ are instruction sequences (inSeq's), written in a notation L_{is} which have been constructed on the basis of designs D_j denoted in design notation L_{design}. In some cases programs*

p_j in a program notation L_{prog} constitute a stage between D_j and X_j and X_j has been constructed by as $X_j = C_{\mathsf{prog2is}}(p_j)$, with C_{prog2is} a generic compiler written in notation L_{comp} which has been used for other projects already, which can be put into effect on remote host H from where outcomes can be distributed over various platforms. The design for utilities $D_{n+1}, \ldots, D_{n+m}$ has also been generic, and so was the construction of the corresponding instruction sequences X_j, and the latter have been used before, and are being used, for other projects.

Assumption 5.2. *(An algorithm as a point of departure.) For $j \in \{1, \ldots, n\}$ the design D_j has been developed on the basis of an algorithm A_j, which itself is given as a flock of k_A, supposedly pairwise algorithmically equivalent, algorhymes $X^A_{j,1}, \ldots X^A_{j,k_A}$. These algorhymes have different backgrounds including open scientific literature.*

Assumption 5.3. *(Context aware design of peripheral drivers.) For all $j \in \{1, \ldots, n\}$ the design D_j has been developed in context, that is aware of the intended use of the platforms for which the corresponding X_j are in part or even entirely in control.*

I will assume that various human agents, all included in a large group H_{agents}, work with and for the various systems and platforms. Matters of knowledge and judgement are to be viewed from the perspective of a human agent or of a group of agents represented by one of its members.

Assumption 5.4. *(Transparent design histories.) For each artefact R a design and construction history is known or partially known to an agent B For instance the construction of X_i from D_i has been carried out by team $T_i^{c[t_1,t_2]}$ in the time span $[t_1, t_2]$, while maintenance is performed by team T_i^m. Members of teams for design, construction and maintenance are called engineers. For each team T the composition $comp(T, t)$ at time t is known as a list of persons with identifying data plus a list of zero or more, perhaps an unknown number, of pointers to unknown/anonymous members of the team.*

Assumption 5.5. *(Active maintenance assumption.) A team of human operators exercises some form of control over the population of P platforms in existence at a given moment in time. Depending on permissions, the setting of which is regulated by an internal hierarchy among the operators, various operators are able to add, delete, replace, and upgrade application programs as well as utility programs both to the entire class of P-platforms in operation and to individual platforms among these.*

By combining the listed assumption a workflow model is obtained which, in principle, allows to attribute the occurrence of various moral and other faults in inSeq's, algorhymes and algorithms to activities made by individual engineers or by teams of engineers. For faults which must not occur by any means, facilitating such attribution is essential.

For moral faults we are left with the question: In connection with the notion of algorithm faultm as well as design fault, algorhyme fault, program fault, and

instruction sequence fault, in the context of embedded systems I wish to mention the following considerations:

1. A typical software engineering task involves the development of a new component say X_{n+1} which will add to the control of the system, where it is assumed that without the new component the system already has a reasonably well-understood working.

2. A new component must satisfy COSC requirements (for Contribution Of Software Component) in the terminology of [6].

3. COSC requirements are likely to be extra-computer science requirements (see 3.5 above). Moreover COSC requirements are likely to be overall system requirements rather than semantic requirements (see 1.2.4 above).

4. Under the above conditions notions of failure, fault, and defect are meaningful for a new component X_{n+1}.

5. Following the analysis of [9, 10] the MCAS algorithm may be modelled as an additional thread, comprising an implementation of the MCAS algorithm, to a mature multi-threading system. Now the suggestion of [10] is that the implementation of MCAS in the Boeing 737 Max system has not been faulty, and shows no defect for the very reason that no resolution of the failure, which appeared with both deadly crashes, can be achieved by simply replacing a fragment of the instruction sequences for an implementation of X_{n+1}, in particular not by simply making the program look at two angle of attack sensors instead of inspecting just one.

6. Instead, so it is argued in [10] it is conceivable that during the design of the MCAS software a so-called software process flaw has occurred so that it has not been noticed that given the available sensors and computing infrastructure, in a Boeing 737 Max at that moment, no adequate implementation of the COSC requirements could be found, which according to [10] is a consistent and even plausible assumption.

6 Concluding remarks

This paper proposes a novel definition of algorithms supported by the novel notion of an algorhyme. The definition is an elaboration into a more rigorous form of the definition of an algorithm given by Jeff Edmonds in his textbook "How to think about algorithms" [27] which reads as follows:

> An *algorithm* is a step-by-step procedure which, starting with an input instance, produces a suitable output. It is described at the level of detail and abstraction best suited to the human audience that must understand it. In contrast code is an implementation of an algorithm that can be executed by a computer. Pseudocode lies between these two.

In [21] and attempt is made to provide a definition of algorithmic equivalence under quite restrictive conditions. As mentioned in that paper and in various references cited in there, it is questionable that such an attempt can succeed. Algorithmic equivalence as defined in [21] is unable to abstract from details which human readers upon further reflection consider to be of marginal importance only. "True algorithmic equivalence", that is in an intuitive and informal sense allows to identify, that is consider equivalent, more instruction sequences than is done with the definition in [21].

Secondly it is argued that in practice an algorithm may be thought of, reified, as a flock, that is a finite collection, of algorhymes. Looking at an algorithm that way allows to define so-called DOA, deviation of algorithm defects in an algorhyme, and more importantly allows to determine a notion of fault for an algorithm.

Thirdly I propose how to conceptualize the notion of a moral defect and a moral fault for programs, algorhymes and algorithms. I expect that the availability of such notions can to the ongoing debate on moral aspects of so-called "killer robots," as well as being helpful for advancing the analysis of moral aspects of robots in care and in various forms of entertainment.

References

[1] Aein, M.J., Aksoy, E.E., Wörgötter, F.. : Library of actions: implementing a generic robot execution framework by using manipulation action semantics. The International Journal of Robotics research., **38** (8), pp. 910–934 (2019).

[2] Avižienzis, A. Laprie, J.C., Randell, B.: Fundamental concepts of dependability. In Workshop on Robot Dependability: Technological Challenge of Dependable Robots in Human Environments, Seoul (2001).

[3] Avižienzis, A. Laprie, J.C., Randell, B., Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. IEEE Transactions on dependable and secure computing, **1** (1), pp.1–23 (2004).

[4] Bergstra, J.A.: Quantitative expressiveness of instruction sequence classes for computation on bit registers. Computer Science Journal of Moldova, **27** (2), pp. 131-161 (2019). http://www.math.md/publications/csjm/issues/v27-n2/12969/

[5] Bergstra, J.A.: Promises in the context of humanoid robot morality. International Journal of Robotic Enginering **5** (2), 20 p, (2020).

[6] Bergstra, J.A.: Instruction sequence faults with formal change justification. Scientific Annals of Computer Science **30** (2), pp. 105–166 (2020).

[7] Bergstra, J.A.: Qualifications of instruction sequence failures, faults and defects: dormant, effective, detected, temporary, and permanent. Scientific Annals of Computer Science, **21** (1), pp. 1–50, (2021) http://doi.org/10.7561/SACS.2021.1.1

[8] Bergstra, J.A., Bethke, I.: Predictable and reliable program code: virtual machine-based projection semantics. In Handbook of Network and System Administration, Elsevier 2001, pp. 653–686 (2007).

[9] Bergstra, J.A., Burgess, M.: A promise theoretic account of the Boeing 737 Max MCAS algorithm affair https://arxiv.org/abs/2001.01543v1 [cs.OH] (2019).

[10] Bergstra, J.A., Burgess, M.: Candidate software process flaws for the Boeing 737 Max MCAS algorithm and a risk for a proposed upgrade. https://arxiv.org/abs/2001.05690v1 [cs.CY] (2020).

[11] Bergstra, J.A., Klint, P.: About "trivial" software patents: the IsNot case. Science of Computer Programming, **64**, pp. 264–285 (2007).

[12] Bergstra, J.A., Loots, M.E.: Program algebra for sequential code. Journal of Logic and Algebraic Programming **51** (2), pp. 125–156 (2002).

[13] Bergstra, J.A., Middelburg, C.A.: Thread algebra for strategic interleaving. Formal Aspects of Computing, **19** (4), pp. 445–474, (2007).

[14] Bergstra, J.A., Middelburg, C.A.: Distributed strategic interleaving with load balancing. Future Generation Computer Systems, **24** (6), pp. 530–548, (2008).

[15] Bergstra, J.A., Middelburg, C.A.: A thread calculus with molecular dynamics. Information and Computation, **208**, pp. 817–844, (2010).

[16] Bergstra, J.A., Middelburg, C.A.: Data linkage dynamics with shedding. Fundamenta Informaticae **103** (1-4), pp. 31-52 (2010).

[17] Bergstra, J.A., Middelburg, C.A.: Thread algebra for poly-threading. Formal Aspects of Computing **23** (4), pp. 567–583 (2011).

[18] Bergstra, J.A., Middelburg, C.A.: Instruction sequence processing operators. Acta Informatica **49** (3), pp. 139–172 (2012).

[19] Bergstra, J.A., Middelburg, C.A.: Data linkage algebra, data linkage dynamics, and priority rewriting. Fundamenta Informaticae, **128** (4), pp. 367–412, (2013).

[20] Bergstra, J.A., Middelburg, C.A.: Instruction sequence expressions for the secure hash algorithm SHA-256. arxiv [cs.PL] 1308.0219 (2013).

[21] Bergstra, J.A., Middelburg, C.A.: On algorithmic equivalence of instruction sequences for computing bit string functions. Fundamenta Informaticae **138** (4), pp. 411–434 (2015).

[22] Bergstra, J.A., Middelburg, C.A.: On instruction sets for Boolean registers in program algebra. Scientific Annals of Computer Science **26** (1), pp. 1–26 (2016).

[23] Bergstra, J.A., Middelburg, C.A.: A short introduction to program algebra with instructions for Boolean registers. Computer Science Journal of Moldova, **26** (3), pp. 199–232 (2019). http://www.math.md/files/csjm/v26-n3/v26-n3-(pp199-232).pdf

[24] Bergstra, J.A., Ponse, A.: Combining programs and state machines. Journal of Logic and Algebraic Programming **51** (2), pp. 175–192 (2002).

[25] Ciancarini, P., Russo, D., Sillitti, A., Succi, G.: A guided tour of the legal implications of software cloning. ICSE '16, Austin, TX, USA, http://dx.doi.org/10.1145/2889160.2889220 (2016).

[26] Diallo, N., Ghardallou, W., Mili, A.: Relative correctness: a bridge between testing and proving. CEUR Workshop Proceedings, (2016).

[27] Edmonds, J.: How to think about algorithms. Cambridge University Press, ISBN 978-521-61410-8, (2008).

[28] Glass, R.L.: Persistent software errors. IEEE Transactions on Software Engineering **7** (2) pp.162–168 (1981).

[29] Grottke, M., Tivedi, K. S.: A classification of software faults. 16 IEEE Symp. on Software Reliability Engineering, pp 19–20 (2005).

[30] Karshtedt, D., Lemley, M.A., Seymore S.B.: The death of the genus claim. To appear in Harvard Journal of Law and Technology. https://scholarship.law.gwu.edu/faculty_publications/1528/ (2021).

[31] Kroeger, O.: Why are software patents so elusive? Masaryk University Journal of Law and Technology, **5** (1), pp. 57–70 (2011).

[32] Kroeger, O.: Plato on reification and intellectual property. Mag. Phil. Thesis Universität Wien, https://othes.univie.ac.at/19656/1/2012-03-06_0001289.pdf (2012).

[33] Laprie, J.C.: Dependable computing and fault tolerance: concepts and terminology. FTCS-15 pp. 2–11 (1985).

[34] Laski, J.: Programming faults and errors: towards a theory of software incorrectness. Annals of Software Engineering **4** pp. 79–114 (1997).

[35] Lemley, M.A., Risch, M. Sichelman, T., Polk Wagner, R.: Life after *Bilsky*. Stanford Law Review, **63**, pp. 1315–1347 https://scholarship.law.upenn.edu/cgi/viewcontent.cgi?article=1737&context=faculty_scholarship (2010).

[36] Mili, A., Frias, M.F., Jaoua, A.: On faults and faulty programs. P. Höfner et al. (Eds.): RAMiCS 2014, LNCS 8428, pp. 191–207, (2014).

[37] Ogilvie, J.W.L.: Defining computer program parts under Learned Hand's abstractions test in software copyright infringement cases. Michigan Law Review **91**, pp. 526–570, https://repository.law.umich.edu/mlr/vol91/iss3/5 (1992).

[38] Ploski, J., Rohr, M., Schwenkenberg, P., Hasselbring, W.: Research issues in software fault categorization. ACM Software Engineering Notes **32** (6), pp. 1–8 (2007).

[39] Plotkin, R.: Computer programming and the automation of invention: a case for software patent reform. International Review of Law, Computers and Technology, **17** (3), pp. 337–346 https://papers.ssrn.com/sol3/papers.cfm?abstract_id=503822 (2003).

[40] Raman, V., Kres-Gazit: Explaining impossible high-level robot behaviours. IEEE Transactions on Robotics **29** (1), pp. 94–1004 (2013).

[41] Rattan, D., Bathia, R. Singh, M.: Software clone detection: A systematic review. Information and Software Technology, **55**, 1165–1199 (2013).

[42] Risch, M.: A surprisingly useful requirement. George Mason Law Review **19** (1), pp. 57–111 (2011).

[43] Saha, S., Saha, R. K., Prasad, M.K.: Harnessing evolution for multi-hunk program repair. IEEE ICSE-2019, https://arxiv.org/abs/1906.08903 (2019).

[44] Seymore, S. B., The Research Patent. Vanderbilt Law Review, Vol. 74, pp. 143-186, Vanderbilt Law Research Paper No. 20-25, Available at SSRN: https://scholarship.law.vanderbilt.edu/cgi/viewcontent.cgi?article=2220&context=faculty-publications, (2021).

[45] Sumner, J.P., Lundberg, S.W.:Patentable computer program features as uncopyrightable subject matter. AIPLA Q.J. **17**, pp. 237–155 (1989).

[46] Wong, W.E. et. al.: A survey of software fault localization. IEEE Transactions on Software Engineering, **42** 8 pp. 707–740 (2016).

[47] Xiaojian, L., Ting J., Xiaofeng, D.: Formal definition of program faults and hierarchy of program fault-tolerant abilities. IEEE ICISCE2017, (2017).