Qualifications of Instruction Sequence Failures, Faults and Defects: Dormant, Effective, Detected, Temporary, and Permanent

Bergstra, J.A.

DOI
10.7561/SACS.2021.1.1

Publication date
2021

Document Version
Final published version

Published in
Scientific Annals of Computer Science

License
CC BY-ND

Link to publication

Citation for published version (APA):
Bergstra, J. A. (2021). Qualifications of Instruction Sequence Failures, Faults and Defects: Dormant, Effective, Detected, Temporary, and Permanent. *Scientific Annals of Computer Science*, *31*(1), 1-50. https://doi.org/10.7561/SACS.2021.1.1

Scientific Annals of Computer Science vol. 31 (1), 2021, pp. 1–50
doi: 10.7561/SACS.2021.1.1

Qualifications of Instruction Sequence Failures, Faults and Defects: Dormant, Effective, Detected, Temporary, and Permanent

Jan A. Bergstra[1]

Abstract

Starting out from the survey of instruction sequence faults from [6] program faults are classified according to the conventional criteria of being dormant, effective, detected, temporary, and permanent. Being retrospectively approved is introduced as an additional qualification. For this theoretical investigation the context is simplified by contemplating instruction sequences as a theoretical model for programs, and by assuming that instruction sequences are supposed to compute total transformations on finite bit sequences of a fixed length only.

The main conclusion which can be drawn from this work concerns the notion of dormancy. First of all it is noticed that the unconventional notion of a dormant failure is both plausible and amenable to a straightforward and convincing definition. The conventional notion of a dormant fault, however, is much harder to grasp and the definition of a dormant fault which is provided in the paper may be disputed.

The notion of a dormant fault seems to admit no convincing intuition. All faults are defects but not the other way around. The idea of a fault exclusively depends on an instruction sequence and a specification of which it is considered to be a candidate implementation. In the presence of a design, however, in addition to faults, the notion of

[1]Informatics Institute, Faculty of Science, University of Amsterdam, Science Park 904, 1098 XH Amsterdam, the Netherlands, email: `J.A.Bergstra@uva.nl`, `janaldertb@gmail.com`.

a deviation from design (DFD) defect arises, which constitutes a class of defects many of which are not faults. For DFD defects the notion of dormancy admits a straightforward and convincing definition.

Keywords: fault modality, Laski fault, MFJ fault, SRT fault, software defect, change justification

1 Introduction

A systematic terminology for software faults, errors, failures, and mistakes has been proposed by several authors. A recent survey of these notions can be found in [1] and in [26]. A survey of formalised notions of software faults can be found in [6]. I refer to that paper also for more references on software faults.

The work in [6] was simplified by considering instruction sequences as program notations only. The outcome of [6] in particular consists of a plurality of so-called fault patterns, also called fault modalities.

There is no guarantee or proof that formal definitions of the most useful kinds of faults were identified in [6] . Some fault patterns have numerical parameters and thereby represent an infinite plurality of fault patterns.

In this paper I will extend [6] with fairly rigorous, though not always formal, definitions of a plurality of predicates which are commonly used in connection with software faults. In particular the following predicates will be considered: dormant, effective, detected, temporary, and permanent. I will refer to such predicates as fault qualifications. The description of such qualifications is extended with qualifications for failures and with qualification for a class of defects which may not be faults at the same time.

1.1 Summary of Results

As new outcomes of this work I mention the following conclusions:

(i) the notion of a dormant failure is very clear and is much more self-evident than the traditionally used notion of a dormant fault,

(ii) introduction of the notion of a primary symptomatic failure for a fault,

(iii) faults are supposed to be of three categories: Laski faults, MFJ (for Milli, Frias and Jaoua) faults and SRT (for successful regression test) faults (this is a limitation, and following [6] a plurality of other categories of fault may be distinguished), each of these having BB (black box), GB (grey box), and WB(white box) versions,

(iv) the notion of a dormant fault most plausibly applies to BB SRT faults,

(v) the notion of a detected fault most plausibly applies to WB SRT faults,

(vi) all faults are defects but not conversely. Defects which are not necessarily faults allow the following ramification: TMO DFD defects, BB DFD defects, and WB DFD defects where the latter two categories ramify into Laski, MFJ and SRT subcategories,

(vii) an initial discussion of design flaws,

(viii) a characterisation of a bug as TMO DFD defects which admits an obvious (easy to find and apply) resolution (so that bugs may not be faults),

(ix) the claim that testing cannot (by itself) lead to the detection of faults (because testing is insufficient to establish a causal link between a fault and a failure), and also not to the detection of bugs (which are either faults or, if not, are unrelated to failures), and

(x) A straightforward technical observation (in 3.9): if during the debugging stage of a software process the database with production data for a candidate implementation (say instruction sequence `X`) of a given specification contains (perhaps among other detected faults) two (detected) WB SRT faults, say A and B, with both faults pertaining to disjoint fragments of `X`, and if the choice is made to resolve fault A by applying its proposed change of a faulty fragment of `X` thereby obtaining $\mathtt{X}'$, then the residual fault for B is a candidate fault of $\mathtt{X}'$ which may not be a fault proper because the regression test for its proposed change has become unsuccessful. The latter problem is due to the fact that the change proposed in B may fail on the input for the primary symptomatic failure of A on which $\mathtt{X}'$ (as opposed to `X`) works correctly and which, upon having resolved fault A, has been included in the regression test suite which is part of the database with production data for the software process at hand.

As a consequence of this observation one may notice that a detected fault may become "undetected" (I prefer to use discarded) during the software process, without the underlying problem addressed by that particular fault (i.e. its primary symptomatic failure) being resolved. What happens instead is that the fault (the residual fault initially degraded to candidate fault status) ceases to provide a satisfactory solution for its primary symptomatic failure and therefore loses detected fault status.

(xi) A related observation (in 5.5) that if two disjoint defects are present, upon repair of the first one the residue of the second defect may not be a defect of the same modality anymore.

1.2 Simplifying Assumptions

In order to simplify the project at hand I will assume the following extensive combination of simplifications:

1. As programs I will only consider instruction sequences in one of the notations of PGA (program algebra) style instruction sequence theory.

   In particular I will only consider so-called PGLB programs over single bit services. I assume that the definitions given below can be extended to many other instruction sequence notations and to many other imperative program notations without significant problems. The relevance of a focus to a single notation (in this case PGLB) with a theoretical status is that all definitions become mathematically clear and unambiguous. This clarity comes at the price of potentially limited generality, a price which I think is worth paying.

   PGLB was introduced in [8], recent expositions are in [14, 4, 16]. A complete exposition of PGA style notations and concepts can be found in [12]. The case of programs consisting of a plurality of instruction sequences (so-called polyadic instruction sequences) is discussed in [10, 11]. All work done below on faults and defects carries over from single instruction sequences to polyadic instruction sequences.

2. I will focus exclusively on programs which are supposed to compute a total or partial function from $\{0,1\}^{n}$ to $\{0,1\}^{m}$ for certain (decimal) natural numbers n and m. (For a justification of the phrase decimal natural number I refer to [7].) Such functions will be denoted with $P, Q, ..$ below.

   Bits are stored in single bit services named by foci. Focus method notation f.m (as well as the tests +f.m and −f.m) is used for instructions: apply method m to the bit stored in the service named by focus f.

   Clearly a (partial) function P is a finite object, simply a list of pairs of tuples of bits.

3. Although a computation of an instruction sequence X over n bits may take time exponential in n it is assumed that all computations can be effectuated. This assumption restricts the collection of instruction

sequences which are considered candidate implementations of a given specification.

4. A failure is a mere mismatch between what happens and what is supposed to happen, there is no context in which adverse consequences of failures are analysed or even suggested.

5. Assuming that P constitutes a (partial) function which is to be computed. Then the oracle problem for P has been solved. This means that checking that $P(\mathtt{b_1}, \ldots, \mathtt{b_n}) = (\mathtt{c_1}, \ldots, \mathtt{c_m})$ is considered easy, or at least feasible.

6. For providing formal specifications for P, I will use first order expressions over the structure of bits where the names of foci for bits serve as variables. ϕ is then supposed to specify the graph of P, or at least to constrain it. The use of quantifiers helps giving shorter formulae, while removal is possible at the price of a combinatorial explosion: $\exists \mathtt{b} \in \{\mathtt{0}, \mathtt{1}\}.\phi \equiv [\mathtt{0}/\mathtt{b}]\phi \vee [\mathtt{1}/\mathtt{b}]\phi$.

7. Formally verifying that a candidate implementation X produces a partial function P which satisfies specification ϕ is always possible in principle although it may be prohibitively complex in practice.

8. I will assume that only a single agent A is using an instruction sequence, testing it, applying changes to it, and collecting information about it. This single agent may be a team of software engineers, but I won't consider users who don't acquire information from tests and testers who don't know how the instruction sequence looks like, and programmers who are unaware of the statistics of use etc. Investigating the epistemic logic of instruction sequences for a heterogeneous group of agents with different roles is intriguing but doing so is too complex at this stage.

9. I will only consider single faults, that is faults consisting of a single fragment of an instruction sequence. This leaves so-called multi hunk faults and multiple faults untouched. In Bergstra [6] various other notions of fault are distinguished: orthogonal multi-MFJ faults, essential MFJ-faults, MFJ* faults (which allow a chain of MFJ faults leading to a correct implementation), proper MFJ* faults.

10. Assessment of compliance of a given fragment in an instruction sequence X with the corresponding fragment of the design D on the basis

of which `X` has been constructed is supposed to be a matter of human judgement.

Constrained by these formidable simplifications the paper contains theoretical work only, but I hope that the insights obtained extend beyond these narrow confinements. As it will turn out even under these severe restrictions providing formal definitions of fault qualifications is challenging, while giving formal definitions of failure qualifications is straightforward.

I will assume that the number `n` of input bits is so large that testing all $2^{\mathtt{n}}$ inputs is undoable in practice in spite of the fact that it is a computable task. This is expressed with the following assertion, to be read against the background of the listed simplifying assumptions.

Conceptual proposition 1.1 *Verifying that* `X` *implements P by means of exhaustive testing is unfeasible.*

I will not commit to the idea that testing can only reveal failures: a single test may for instance establish that a proposed factorisation program `X` finds a factorisation for a natural number which could not be found by earlier methods and thereby the test contributes to confidence in the quality of `X`. For a factorisation program partial correctness may be trivial to prove by formal means, but because the program that diverges always is partially correct, proving mere partial correctness is insufficient to create confidence in the usefulness of `X` for the purpose of factorization. Testing, however, may well create such confidence.

1.3 Classical Informal Definitions of: Failure, Error, Fault and Mistake

The four informal definitions listed in Definition 1.1 have become widespread in software engineering. These definitions make sense under the simplifying assumptions mentioned above just as well, be it that, admittedly, a hypothetical practice of writing instruction sequences for bit sequence transformations might probably not by itself give rise to such an introduction of this kind of definitions.

Definition 1.1

Failure. *If a program is executed and a result (an action, a state, or an output) is produced which is contrary to one of its specifications, that event is called a failure.*

Error. *A failure is the externally visible part of an error, which is a state in which a program and its related data happen to enter during a computation, which, according to the specification, or according to additional requirements is "forbidden", i.e. it should not happen.*

Fault. *A fault in a program is a fragment of it (which may itself consist of different parts) which can be considered the cause of the existence of a computation which leads to an error and which externally shows up as a failure.*

Mistake. *A (programming) mistake is an action of a programmer which causes the presence of a fault (in a program).*

This terminology was introduced in Laprie [20], appearing in a more definitive form in Avižienzis, Laprie & Randell [2] and later in Avižienzis, Laprie, Randell, & Landwehr [3]. Persistency of faults apprears in Glass [19]. For a discussion of this terminology I refer to [6].

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

The concept of an ALR fault in a program X is a theoretical notion and it does not matter whether or not its being a fault has actually been detected either empirically or by other means.

Regarding the use of causality I understand the definition as follows: for a fault there must be at least one particular failure of which the fault is considered a cause. Taking a particular failure in focus matters as follows: imagine an instruction sequence which supposedly solves a combinatorial problem, and upon closer inspection it does so faster than a known lower bound allows. For an example, one may consult the lower bound on instruction sequence size for parity checking as determined in [14]. In that case an instruction sequence is provably too short for solving the problem of which it is a candidate implementation for, there is also proof of the presence of a failure, while there is no indication of the presence of a fault, nor is there a clue on how to spot a particular failure.

As a theoretical notion ALR fault is an informal notion, because the underlying notion of causality is in need of further explanation. As it turns out such explanations may be provided in many different ways, none of which is manifestly most plausible.

1.4 The Status of Errors

In the discussion below errors play a marginal role only. I will not introduce any significant notion of error. For the rest of this paper an error is a state of a computation, made up from an input and a program counter (the number of the next instruction to be put into effect from that state of the computation) with the property that either (i) the next step in the computation will unavoidably give rise to an immediate failure (that is: wrong output, or output where divergence was expected), or (ii) the computation has entered in a loop from which it cannot escape while termination of the computation is demanded by the specification.

Dealing with error this way turns error into a derived notion, which may not comply with the intentions of the authors of [2]. By using this definition of error (as a derived notion given the notion of a failure and the notion of a diverging computation) one can eliminate "error" from the collection of primitives required for giving an account of instruction sequences faults based on the principles of [2]. It is an open problem how to incorporate a convincing notion of error in the discussion of instruction sequence faults.

2 Instruction Sequence Failures

Following ALR it is common to speak of dormant faults rather than of dormant failures. Nevertheless is much easier to provide a convincing definition of a dormant fault than of a dormant fault. I will begin with contemplating qualifications of failures.

I will assume that a function $P : \{0,1\}^{\mathtt{n}} \to \{0,1\}^{\mathtt{m}}$ is given as well as an instruction sequence $\mathtt{X}$ which is considered to be a possibly faulty implementation of P. I will use service family notation from [12]. Inputs are encoded in a service family $\mathtt{IN}(\mathtt{b}_1,\ldots,\mathtt{b}_\mathtt{m}) = \mathtt{in{:}1.sbs}(\mathtt{b}_1) \oplus \ldots \oplus \mathtt{in{:}m.sbs}(\mathtt{b}_\mathtt{m})$ with $\mathtt{b}_\mathtt{i}$ bits in $\{0,1\}$ and $\mathtt{sbs}(-)$ a single bit service. Outputs $\mathtt{o}_1,\ldots,\mathtt{o}_\mathtt{n}$, if any are computed, i.e. if the computation of

$$\mathtt{X} \bullet (\mathtt{IN}(\mathtt{b}_1,\ldots,\mathtt{b}_\mathtt{m}) \oplus \mathtt{AUXO}_{1,\mathtt{k}}(0,\ldots,0))$$

terminates, (with $\mathtt{AUXO}_{1,\mathtt{k}}(\mathtt{c}_1,\ldots,\mathtt{c}_\mathtt{k}) = \mathtt{aux0{:}1.sbs}(\mathtt{c}_1) \oplus \cdots \oplus \mathtt{aux0{:}k.sbs}(\mathtt{c}_\mathtt{k})$ a family of auxiliary registers) are encoded in a service family $\mathtt{OUTO}(\mathtt{o}_1,\ldots,\mathtt{o}_\mathtt{n})$ $= \mathtt{out0{:}1.sbs}(\mathtt{o}_1) \oplus \ldots \oplus \mathtt{out0{:}m.sbs}(\mathtt{o}_\mathtt{n})$, with bits $\mathtt{o}_\mathtt{i} \in \{0,1\}$.

A failure arises with inputs $(\mathtt{b}_1,\ldots,\mathtt{b}_\mathtt{m})$ if, with $(\mathtt{o}_1,\ldots,\mathtt{o}_\mathtt{n}) = \mathtt{P}(\mathtt{b}_1,\ldots,\mathtt{b}_\mathtt{m})$ it is not the case that

$$\partial_{\{\mathtt{in{:}1},\ldots,\mathtt{in{:}m}\}\cup\{\mathtt{aux0{:}1},\ldots,\mathtt{aux0{:}k}\}}(\mathtt{X} \bullet \mathtt{IN}(\mathtt{b}_1,\ldots,\mathtt{b}_\mathtt{m}) \oplus \mathtt{AUXO}_{1,\mathtt{k}}(0,\ldots,0))$$

$$= OUT0(o_1, \ldots, o_n)$$

A failure is characterised by its inputs, which uniquely determine outputs (if any).

2.1 Test/use Histories

A test/use history for X and P consists of a sequence $(b_{1,1}, \ldots, b_{m,1})^{v_1}, \ldots, (b_{1,k}, \ldots, b_{m,k})^{v_k}$ of successive inputs, coupled with verdicts $\mathtt{v_i} \in \{\mathtt{pass}, \mathtt{fail}, \mathtt{fail}{:}\mathtt{temp}, \mathtt{fail}{:}\mathtt{perm}, \mathtt{fail}{:}\mathtt{rapd}\}$ where

$$\mathtt{v_i} = \mathtt{pass} \iff P(\mathtt{b_{1,i}}, \ldots, \mathtt{b_{m,i}}) = \mathtt{X} \bullet (\mathtt{b_{1,i}}, \ldots, \mathtt{b_{m,i}})$$

A test on input $\vec{b}$ either passes, or it fails in which case the input is coupled with one of the verdicts in $\{\mathtt{fail}, \mathtt{fail}{:}\mathtt{temp}, \mathtt{fail}{:}\mathtt{perm}, \mathtt{fail}{:}\mathtt{rapd}\}$. The use of an extension to fail is a matter of maintenance policy by the author of X, and is not dictated in any manner by test outcomes. The assignment of these qualifications is explained below.

2.2 Survey of Failure Qualifications

Failure qualifications regarding X when considered as a candidate implementation of specification P are given relative to failures as represented by the inputs $\vec{b} = (\mathtt{b_1}, \ldots, \mathtt{b_m})$ on which fails to compute $P(\vec{b})$. I will say that in such cases $P(\vec{b})$ features a failure of X, or simply that X fails on $\vec{b}$.

These qualifications are given from the perspective of a single user A who is also doing maintenance on X including software fault resolution.

1. *dormant failure*: $\vec{b}$ features a dormant failure if (i) $\vec{b}$ features a failure and (ii) it is not known to A that $\vec{b}$ features a failure. (That is $\vec{b}$ is not among the input tuples in the test/use history.)

2. *detected failure* (alternatively: effective failure, activated failure, latent failure): $\vec{b}$ features a detected failure if (i) $\vec{b}$ features a failure and (ii) $\vec{b}$ occurs in the listing of the test/use history. (A may have acquired this knowledge by means of a test or during use or via theoretical analysis, or by being promised to that extent by a trusted agent; an undetected failure is the same as a dormant failure). A detected failure is available in the database with verdict fail as long as it has not been provided with one of the three possible extensions in $\{\mathtt{temp}, \mathtt{perm}, \mathtt{rapd}\}$.

3. *temporary failure*: given a detected failure the plan may be made to resolve the failure (i) finding one of its causes, in the form of a fault paired with a proposed change, and (ii) changing the faulty fragment accordingly. Once a plan to that extent has been made by A the failure is temporary, i.e. its presence is temporarily tolerated only, which is represented by means of a verdict `fail:temp` in the test/use history.

4. *permanent failure*: given a detected failure, the plan may be made not to resolve the failure by means of a modification of the instruction sequence. Once the decision has been taken not to resolve the failure featured by $\vec{b}$ that failure has become permanent (be it that it may become resolved unintentionally, as a side-effect of resolving another failure in a later stage). Permanence of a failure is indicated by means of a verdict `fail:perm`.

5. *retrospectively approved pass*: A permanent failure may become approved behaviour of `X` part of the specification, i.e. it becomes normative, and its resolution involves a change of the specification rather than to change of the instruction sequence. This condition is indicated with the verdict `fail:rapd`. (Thus a test which is retrospectively approved passes w.r.t. a different specification.)

The qualifications for instruction sequence failures suggest a life-cycle model. A failure always starts as dormant, then it may become detected and subsequently it either becomes temporary or it is considered to become permanent, in the latter case a subsequent transition to retrospectively approved status may take place, with as an effect that it counts as a pass. Yet later in the development process the status may be turned into pass, thereby forgetting a part of the history of improvement of `X` and its specification.

3 Instruction Sequence Faults: Black box Faults, Grey box Faults, and White box Faults

The above discussion of failure qualifications is unsurprising in the sense that each of the qualifications is given a reasonably convincing definition. Minor modifications of these particular definitions may be suggested but the respective definitions are well-defined assuming that one admits the use of any epistemic logic for A which allows A to speak of its knowledge about `X` and P. In these definitions I use test/use histories as a vehicle for providing the semantics of a trivialised epistemic logic. More sophistication in the design of an epistemic logic about instruction sequence processing can be imagined and the definitions can be adapted accordingly.

The primary motivation for writing this paper stems from the observation that the introduction of qualifications for faults, which is common in the literature on software faults and failures, cannot be easily accomplished in a convincing manner for any of the formal definitions of a fault as discussed in [6]. The difficulty with designing qualifications of faults is that these come with a range of different information items and that even upon choosing the simplest conceivable epistemic logic for speaking about the various options for these items the necessity arises to make design choices for which many alternatives can be imagined.

In order to make progress on the matter I will distinguish three cases for faults in connection with the availability of additional information:

White box fault. A white box fault in an instruction sequence consists of a fragment of it coupled with information sufficient to infer that it is indeed a fault of a certain modality. These information items consist of: (i) a primary symptomatic failure, (ii) a proposed change, (iii) justification of the resolution of the primary symptomatic failure upon effecting the change, and, (iv) justification of the change in accordance with the modality of the fault.

Grey box fault. A grey box fault consists of a fragment coupled with a part of the information required for a white box fault. For instance only a primary symptomatic failure may be given. Or alternatively only a change is given, but no justification of it and no primary symptomatic failure is indicated.

Black box fault. A black box fault consists of a fragment together with the claim that it can be equipped with additional information items so that a white box fault is obtained.

3.1 Terminology: Fault, Fault Location, Fault Size, Change, Change Justification

A fault in an instruction sequence is a fragment of it, i.e. a subsequence, perhaps equipped with additional information. The fault location is the number of the first instruction of the fault and the fault size is its number of instructions.

In advance of qualifying a fragment as a fault, I will speak of a candidate fault with candidate fault location and candidate fault size. So-called spectrum based fault localisation ([25, 27] and [28]) when applied to instruction sequences, is about finding candidate fault locations, from the statistics of positions in the instruction sequence which were visited during computations leading to failure. If many computations leading to a failure visit the same instruction, that provides an indication of the presence of a faulty fragment which contains said instruction.

A (candidate) multi-hunk fault consists of a collection of fragments, called the parts of the multi-hunk fault. The location of a multi-hunk fault consist of the list of locations of the respective fragments, working from left to right, and instead of a length a multi-hunk fault has a sequence of lengths, and the size of a multi-hunk faulty is the sum of the lengths of its parts. For simplicity of the discussion I will not discuss multi-hunk faults in this paper.

A fault pattern indicates what information is to be provided in addition to a fault locality and a fault length. This involves information about (i) the maximum length of a fault, (ii) the maximal length of the change, (iii) restrictions on the instruction set of a change, (iv) the criterion for justification of a change, (v) the mechanism of justification of change, (vi) naming of the pattern.

The principal idea of a fault is that a change can be found (i.e. another fragment) so that replacing the fault by its change leads to an improved implementation of the required specification. Replacing the fault by its change is supposed to remove (solve, resolve, repair) the fault. In addition this change must remove a failure (referred to as the primary symptomatic failure) so that a causal chain between the fault and at least one failure becomes apparent.

A fault pattern instance is a fault complemented with the information which is required by the fault pattern. Fault patterns comprise a perspective on causality. The ALR notion of fault (see 1.2 above) depends on a notion of causality. When understanding causality as the possibility of finding a change which brings about an improvement, defining causality is reduced to defining a notion of improvement.

Following [6] among several other options the following three different notions of improvement are distinguished, each of which may be ramified with parameters for the maximum number of instructions which a fault may contain and the maximal number of additional instructions for a change. Moreover fault patterns provide an indication on the criterion according to which the validity of the change is justified.

Laski fault: by applying the change at least one failure is resolved, and moreover it is required that after the change has been made a correct implementation is obtained,

MFJ (Mili, Frias & Jaoua) fault: improved correctness after change is required; in other words failure resolution for at least one failure must be obtained in addition to preservation of correctness of outputs for all inputs,

SRT fault: a fault whose change is merely justified using a successful regression test (SRT), which is a weaker criterion than required for an EC MFJ fault, (after the change at least one failure is removed, thereby introducing a new test, and in addition successful completion of the current regression test suite is required; the regression test accumulates all tests which have been completed during earlier phases of the design of the system).

These three kinds of faults differ in quality of improvement as well as in the work that needs to be done in order to detect a fault of that kind, i.e. to confirm that a candidate fault complies with the pattern. I will phrase this matter in the form of conceptual propositions.

For the use of the names of Laski, Mili, Frias and Jaoua in the context of fault patterns I refer to [6]. The relevant key references to their work are: [21] and [23] respectively (additional explanation is given for the latter in [17]). Both papers contribute essential ideas about how to interpret the notion of causality which underlies the intuition of a program fault. SRT faults were introduced as a variation on MFJ faults in [6]. Rather than to offer a convincing interpretation of causality SRT allows to compromise the idea of causality while achieving computational feasibility in return.

For a justification of the use of the notion of a test I refer to the work surveyed in [22]. In any case I will use testing in a more narrow sense than a broad interpretation comprising both validation and verification.

Conceptual proposition 3.1 *Detecting a Laski fault (i.e validating that a candidate Laski fault is in fact a Laski fault) requires formal verification, because correctness needs to be verified for all inputs.*

This proposition holds under the assumption that the number of input bits is so large that exhaustive testing is unfeasible. There is no guarantee that an attractive, i.e. short, proof can be found.

Conceptual proposition 3.2 *Validating that a candidate MFJ fault is an MFJ fault requires the check that for all inputs if the outcome is valid before application of the change it remains valid after application of the change. This check is of a complexity comparable to (formal) verification.*

This conceptual proposition is valid only if the number of inputs is sufficiently large and if the number of input sequences on which the original instruction sequence computes a correct result is sufficiently large.

Conceptual proposition 3.3 *Validating that a candidate SRT fault is an SRT fault requires performing a test for each element of the current test suite. For this to be feasible it is needed that: (i) computations on inputs of the test suite do not take too much time and space, (ii) that the current test suite is not too long, and (iii) that the oracle problem has been solved in a practically computable manner.*

Now faults and localisation of faults mainly occurs in circumstances where verification is either too hard or too costly so that both Laski faults and MFJ faults are probably remote from practical intuitions of fault.

Conceptual proposition 3.4 *On the approximation of programmer intuition of faults:*

(i) *SRT faults are closer to conventional programmer's intuitions than both Laski faults and MFJ faults.*

(ii) *BB faults are closer to programmer intuitions than GB faults and WB faults.*

(iii) *Programmer intuition of a fault is closer to a DFD defect (to be defined later), than to an ALR fault. In particular the archaic notion of a bug refers to a special case of DFD defects rather than to an ALR fault.*

(iv) *Programmers may often not have an intuitive grasp of the degree to which Laski faults, MFJ faults and SRT faults, each of which are instances of ALR faults, together cover the idea of an ALR fault.*

3.2 Justification Modalities of Changes

Justifications of change for Laski faults and for MFJ faults require proofs (including model checking, and including, if possible also exhaustive testing). I will distinguish the following modalities of proof:

1. MPG/MPC: manual proof generation combined with manual proof checking (MPC, where: manual = performed by a human agent),
2. MPGas/MPC: manual proof generation with automated support combined with MPC,
3. MPG/APC: manual proof generation combined with automated proof checking,
4. MPGas/APC: MPGas combined with automated proof checking,
5. APG: automated proof generation, as a matter of course combined with automated proof checking.

I will append the envisaged method of change justification for Laski faults and for MFJ faults as a postfix in brackets: Laski(MPG/MPC), Laski(APG), MFJ(MPGas, APC), etc.

For instance a Laski(MPGas/APC) fault is one for which a change exists which meets the requirements for a Laski fault and for which the corresponding correctness proof can be found by hand (with automatic support) and can then be proof checked automatically.

An MFJ(APG) fault is an MFJ fault for which a change can be found such that the restricted correctness which is supposed to hold after application of the change can be demonstrated by means of automatic proof generation.

3.3 Size Bounds for Faults and Corresponding Changes

A complication with any notion of instruction sequence fault is that without any quantification an entire instruction sequence may be considered a fault with a complete replacement of the given instruction sequence by a wholly different instruction sequence as a corresponding change. Only by imposing some numerical bounds on the sizes of faults and changes these notions are plausible. The introduction of such bounds, however, complicates the notation in an unfortunate manner. In the world of cars one speaks of total loss, and when dealing with airplanes one speaks of hull loss in order to "do away with" circumstances where local improvements will be of no help. In programming one might work under the assumption that the instruction sequence at hand (considered as a stage in a development process) is not a lost case, i.e. moving forward from there, there will be no stage at which all of it is simply thrown away, and replaced by the result of a different development process. In the absence of a workable terminology concerning lost cases and avoidance of these I will make use of numerical bounds which, when sufficiently small, imply the plausibility of the case (for the relevance of the instruction sequence at hand in spite of it being faulty) not being lost.

A $\mathtt{n}$-Laski(JM) fault with $\mathtt{n} \in \mathbb{N} \cup \{\infty\}$ and JM a justification modality is a Laski(JM) fault (of say $\mathtt{X}$) where the (candidate) faulty fragment $\mathtt{Y}$ has a size ($\mathtt{LLOC}$ for logical lines of code) of less than $\mathtt{n}$ instructions.

An $\mathtt{n}/\mathtt{m}$-Laski/(JM) fault with $\mathtt{m} \in \mathbb{N} \cup \{\infty\}$ is an $\mathtt{n}$-Laski(JM) defect where the (candidate) change $\mathtt{Z}$ has size less than $\mathtt{LLOC}(\mathtt{Y}) + \mathtt{m}$.

Similar definitions apply for MFJ(JM) instead of Laski(JM) and for SRT instead of Laski(JM).

The $\mathtt{2/1}$ changes constitute a special case for PGA notations: only a single instruction is modified, which is a significant restriction. $\mathtt{2/1}$ can be made more expressive, however, by including the unit instruction operator of [24] in the instruction sequence notation.

3.4 Conventional Vagueness of the Intuitive Notion of Program Fault

With the terminology at hand which has been discussed above it becomes clear that one may hardly speak of an intuitive notion of an instruction sequence fault. It is like the idea of a "not well-informed person", an appealing notion at first sight but highly context dependant, so much that only within a specific context this notion may be given a useful meaning. Just as "instruction sequence fault", involves quantification (over change, over means of justification of change, over a symptomatic failure to be resolved, and over its actual resolution upon change), also the notion of a "not well-informed person" involves implicit existential quantification. Considering the matter of a not well-informed person, say A, in more detail, the following questions, some of which take the form of an existential quantification arise:

1. about which topic is A supposedly not well-informed (the scope of content at hand),

2. in comparison with whom is A considered to be not well-informed,
3. is external validation of the judgement "not well-informed" required in the context where that judgement about A is put forward or is used,
4. is there an example of a detailed question in the scope of content at hand about which A produces unwarranted judgements which validate the judgement (of being not well-informed),
5. is it merely the case that A has not been actively informed about the current situation, or is it meant that A maintains wrong judgements about the subject matter.

Now these questions which arise for agent B upon being told that "A is not well-informed" are universally known among human agents so that it is unlikely that (human agent) B draws wrong conclusions from being told that "A is not well-informed".

In the case of instruction sequence faults it is also the case that merely being informed that a program fragment is faulty raises several subsequent questions, which, however merit being made explicit.

Conceptual proposition 3.5 *If B is being told by C that instruction sequence* `X` *contains one or more faults, then A only knows that replacing* `X` *by some locally engineered modification of it would constitute a step forward in the perception of C, including the resolution of at least one symptomatic failure.*

In particular in order to acquire any intuition about the fault being local, its existence amounting to more than merely conveying the incorrectness of `X`*, A needs to obtain answers to a plurality of questions including:*

(localization) *precisely which fragment of* `X` *is considered faulty,*

(proposed change) *which change can resolve the fault,*

(symptomatic failures) *which failures are caused by the fault (and for that reason are supposedly done away by its resolution),*

(primary symptomatic failure) *which symptomatic failure is marked as a symptom of the fault according to C, and must be tested for being done away by its resolution, the test being included in the regression test history, and*

(adequacy) *how is it argued for that the change is likely to do more good than harm?*

3.5 BB, GB, and WB Versions of Laski Faults, of MFJ Faults, and of SRT Faults

Let $\texttt{X}$ be a candidate implementation of specification ϕ. I will explain BB, GB, and WB as additional aspects of fault modalities via an example, for other modalities it works similarly. Consider for instance the fault modality $\texttt{6/3}$ MFJ(APG).

3.5.1 WB Faults by Example

Now a WB $\texttt{6/3}$ MFJ(APG) fault of $\texttt{X}$ is a fragment $\texttt{Y}$ of $\texttt{X}$ such that $\texttt{LLOC(Y)} < \texttt{6}$ together with 5 further pieces of information, labeled with L, C, PSF, SR, and P/E respectively.

(i) locality (L): i.e. the coordinate $\texttt{n}$ of the first instruction of $\texttt{Y}$ in $\texttt{X}$, paired with the size $\texttt{LLOC(Y)}$, i.e. the pair $(\texttt{n}, \texttt{LLOC(Y)})$ where it must be the case that $\texttt{LLOC(Y)} < \texttt{6}$,

(ii) change (C): an instruction sequence $\texttt{Z}$ (written in the same instruction sequence notation as $\texttt{X}$) constrained by $\texttt{LLOC(Z)} < \texttt{LLOC(Y)} + \texttt{3}$, (here $\texttt{X}_{[\texttt{Y/Z}]}$ results from replacing $\texttt{Y}$, that is the occurrence of $\texttt{Y}$ starting at instruction number $\texttt{n}$ in $\texttt{X}$, by $\texttt{Z}$ in $\texttt{X}$).,

(iii) failure (PSF): an input α on which $\texttt{X}$ fails (the primary symptomatic failure),

(iv) successful repair (SR): confirmation of a test run providing evidence that $\texttt{X}_{[\texttt{Y/Z}]}$ succeeds on the primary symptomatic failure (of $\texttt{X}$ w.r.t. the fault at hand), so that indeed the change resolves the failure,

(v) proof or evidence (P/E): an automatically generated proof that $\texttt{X}_{[\texttt{Y/Z}]}$ succeeds on each input where $\texttt{X}$ succeeds.

Next consider the fault modality $\texttt{5}/\infty$ SRT. Now a WB $\texttt{5}/\infty$ SRT fault of $\texttt{X}$ is a fragment $\texttt{Y}$ of $\texttt{X}$ (without a size bound on $\texttt{LLOC(Y)}$) together with 5 further pieces of information, labeled with L, C, PSF, SR, and P/E respectively.

L, C, PSF, and RS are as in the previous case. P/E consists of a report of the successful regression tests which have been carried out. For the generation of this report it is needed that a software process database is available with a history of successful past tests each of which will be included in the regression test. The possibility of carrying out regression tests depends on the ability to solve the oracle problem, which as been "secured" as one (nr. 5) of the simplifying assumptions made in Subsection 1.2.

3.5.2 BB Faults

Having clarified WB faults of various modalities, I now turn to BB and GB versions of these. BB fault modalities are WB fault modalities stripped from 4 of the 5 additional information items listed above, leaving locality (L) made public only. However, a BB fault comes with the claim that additional information items for C, PSF, RS and P/E can be found, and that the BB fault is merely an abstraction of a corresponding WB fault. The BB fault may, however, be the result of abstraction from a plurality of different WB faults. Indeed different changes, different choices for the primary symptomatic failure, different choices for the proof methods or evidence gathering may all correspond to the same BB fault.

A BB `6/10` MFJ(APG) fault of `X` is a fragment `Y` of `X` (as aa fragment it is equipped with unique locality information L) for which additional information can be found so that (as a fault) it can be extended to a WB `6/10` MFJ(APG) fault of `X`. Thus, none of the additional information (on C, PSF, RS and P/E) is made available but it is all claimed to exist by the agent claiming that "`Y` is a BB `6/10` MFJ(APG) fault of `X`".

A BB `4/3` Laski(MPG/APC) fault of `X` is a fragment `Y` of `X` together with a location (L) for it, under the constraint that a change (C), a primary symptomatic failure (PSF), a successful test for the PSF upon having brought about the change, and a manually generated proof (P/E), with automated check of it (also P/E), that the change will produce a correct inplementation of the specification at hand.

3.5.3 GB Faults

GB faults come with additional information which positions these in between of BB faults and WB faults. The notation for GB faults is as follows: a GB(C) `2/1` MFJ(MPGas-APC) fault comes with a (proposed) change but omits information about a PSF, as successful test (RS) that that PSF is resolved upon bringing about the proposed change, and an (MPCas/APC style) proof (or corresponding evidence) that the change complies with the MFJ requirement that on all inputs where `X` does not fail, the changed version of `X` does not fail either.

Similarly a GB(C,PSF) `2/1` MFJ(MPGas-APC) fault consists of an GB(C) `2/1` MFJ(MPGas-APC) fault together with a PSF (and the test report showing a failure of `X`). A GB(PSF) `2/1` SRT fault contains the claim that the PSF can be resolved by making a local change at the location L where only a single instruction is replaced and for which evidence that the change is reasonable can (could) be obtained from a successful regression test.

3.6 Fault Modalities in General

Summarizing the above one finds an extensive family of notions of fault, which I will refer to as fault modalities. I will not try to provide any structure theory for fault modalities. Most of these options are probably devoid of practical significance and not worthy of systematic investigation. Eventually a subset of significant fault modalities may emerge which merits being investigated with an eye towards generality.

I will merely list some modalities as a definition by example: BB `5/10` MFJ(APG), BB `6/25` Laski(MPG/MPC), BB `2`$/\infty$ SRT, WB `10/10` SRT, GB(PSF) `6/10` SRT, GB(C,P/E) `6/10` SRT, and GB(C,PSF, RS,P/E) `6/10` SRT (which is the same as WB `6/10` SRT as all information items are actually exposed).

When pointing out to a fragment `Y` of instruction sequence `X` and claiming that it is faulty without providing further information this amounts to opt for of one of the following BB modalities: BB `LLOC(Y)`$+$`1`$/\infty$ Laski(PM), BB `LLOC(Y)` $+$ `1`$/\infty$ MPJ(JM), and BB `LLOC(Y)` $+$ `1`$/\infty$ SRT, with JM any of the justification modalities listed above in 3.2.

3.7 An Implicit SRT Bias?

I notice that of these modalities only SRT based modalities come without any appeal to verification (including exhaustive testing and model checking).

Conceptual proposition 3.6 *Pointing at instruction sequence faults is most plausibly done in an engineering context where verification is considered a remote option.*

Assuming that in the majority of cases where reference to faults is made the idea of verification is considered remote BB `LLOC(Y)`$/\infty$ SRT remains as the most plausible interpretation (within the context of this paper) of what is meant when a particular fragment `Y` in an instruction sequence is claimed to be faulty (without providing further information, and assuming that faults are required to be ALR faults).

3.8 Some Observations on Faults

I have collected some elementary facts about fault existence for instruction sequences, while adopting the simplifying assumptions as listed in Subsection 1.2. I assume that ϕ specifies a single valued total function. X is considered a candidate implementation of ϕ and failures of X are supposed to be deviations of the behaviour of X from computing the function specified by ϕ.

I will use output registers out0:1, out0:2, .. here the 0 indicates the initial value of the register (while out0:1, out0:2, .. denote single bit registers initialised to 1).

Example 3.1 *The instruction sequence* X *is considered a candidate implementation of specification ϕ which requires that the function constant* (1, 0) *is computed on 2 input bits accessible under focus* in:1 *and* in:2 *writing outputs in single bit services accessible under* out0:1 *and* out0:2*:*

X = −in:1.i/i; #4; out0:1.1/1; out0:2.1/1; !

Now X *features the following faults:*

1. *a WB* 2/1*-Laski fault with* L = (1, 1)*,* C = #2*,* PSF = (1) *(i.e. the input for the PSF), RS done by hand, P/E trivial for an empty regression test suite,*
2. *a GB(C)* 2/1*-Laski fault with* L = (1, 1)*,* C = #2 *(immediate consequence of item 1),*
3. *a BB* 2/1*-Laski fault with* L = 1*, (immediate consequence of item 2),*
4. *a GB(C)* 2/1*-MFJ fault with* L = (2, 1)*,* C = #1*,*
5. *a BB* 2/1*-MFJ fault (immediate consequence of item 4),*
6. *a GB(C)* 3/1*-Laski fault with* L = (1, 2)*,* C = #1*,*
7. *a BB* 3/1*-Laski fault with* L = (1, 2)*, (immediate consequence of item 6).*

Example 3.2 *In the same context as Example 3.1*

X = −in:1.i/i; out0:1.1/1; −in:2.i/i; out0:2.1/1; !

Now X *features the following faults:*

1. *a WB* 2/1*-MFJ fault with* L = (1, 1)*,* C = #1*,*

2. *a WB* 2/1*-MFJ fault with* L = (3, 1)*,* C = #1*,*
3. *a WB* 4/1*-Laski fault with* L = (1, 3)*,* C = out0:1.1/1*,*
4. *a WB* 2/1*-Laski fault with* L = (1, 1)*,* C = out0:2.1/1*.*

Fault injection is the introduction of modifications which are likely to be faults in a program. Fault injection can be helpful for understanding the probability of the presence of faults in the original program. In the case of instruction sequences the simplest form of fault injection arises if a single instruction is modified without taking any care of preservation of semantics. The following three observations can be made concerning the latter form of fault injection.

Proposition 3.1 *(Fault injection reversal.) Let* X *be a correct implementation of ϕ and let* X′ *result from* X *by replacing its* k*-th instruction by another instruction. Then, if* X′ *is not a correct implementation of ϕ it is the case that* X′ *contains a BB* 2/1 *Laski fault.*

Proof: A □ satisfactory change is to reverse the replacement of the k-th instruction, i.e. to undo the fault injection.

Proposition 3.2 *(Simultaneous fault injection I) Let* X *be a correct implementation of ϕ and let* X′ *result from* X *by simultaneously replacing its* k*-th instruction* $\mathtt{u_k}$ *by another instruction* $\mathtt{u'_k}$ *and its replacing its* l*-th instruction* $\mathtt{u_l}$ *by another instruction* $\mathtt{u'_l}$*. Then, if* X′ *is not a correct implementation of ϕ it need not be the case that* X′ *contains a WB* 2/1 *Laski fault at position* k *or at position* l *with the respective changes undoing the fault injection (i.e.restoring the instruction of* X *at the relevant position).*

Proof: ϕ represents the function constant 1 on a single bit. In the instruction sequence notation below %1,%2,... are not part of the instruction sequence but are merely comments for the reader indicating the number in X of the first instruction on its line.

```
X =
%1  out0:1/1; −in:1.i/i;
%3  out0:1.1/1; +in:1.i/i;
%5  out0:1.1/1; !
```

Then using the notation of the statement of the proposition take k = 3 and l = 5, $\mathtt{u'_3}$ = out0:1.1/0 and $\mathtt{u'_5}$ = out0:1.1/0, thus obtaining after simultaneous fault injection:

$\mathtt{X}' =$
%1 $\mathtt{out0{:}1/1}; -\mathtt{in{:}1.i/i}$;
%3 $\mathtt{out0{:}1.1/0}; +\mathtt{in{:}1.i/i}$;
%5 $\mathtt{out0{:}1.1/0}; !$

Reversing either fault injection provides a change which turns either $\mathtt{u}'_3$ or $\mathtt{u}'_5$ into a WB 2/1-MJF fault of $\mathtt{X}'$, and for that reason into a WB 2/1-SRT fault for the empty set of regression tests.

However other changes (from $\mathtt{X}'$) are possible and the possibility of changing $\mathtt{u}'_3$ into #2 demonstrates that $\mathtt{u}'_3$ is a BB 2/1-Laski fault of $\mathtt{X}'$. □

Proposition 3.3 *(Simultaneous fault injection II) Under the same assumptions as Proposition 3.2. If* $\mathtt{X}'$ *is not a correct implementation of* ϕ *it need not be the case that* $\mathtt{X}'$ *contains a WB* 2/1 *MFJ fault at position* k *or at position* l *with the respective changes undoing the fault injection (i.e. restoring the instruction of* X *at the relevant position).*

Proof: Let ϕ represent the function constant 1 on a single bit. It is immediate that X implements ϕ:
$\mathtt{X} =$
%1 #1;
%2 #1;
%3 $\mathtt{out0{:}1.1/1}; !$

Then using the notation of the statement of the proposition take $\mathtt{k} = 1$ and $\mathtt{l} = 2$, $\mathtt{u}'_1 = !$ and $\mathtt{u}'_2 = !$, thus obtaining after simultaneous fault injection:
$\mathtt{X}' =$
%1 !;
%2 !;
%3 $\mathtt{out0{:}1.1/1}; !$

Reversing either fault injection does not provide a change which turns either $\mathtt{u}'_2$ or $\mathtt{u}'_3$ into a WB 2/1-MJF fault of $\mathtt{X}'$.

One may notice that $\mathtt{u}'_1$ constitutes a BB 2/1-MJF fault of $\mathtt{X}'$ because of the option to make use of change #2 while $\mathtt{u}'_1$ is not a fault of any of the forms considered above. However change is made to the second instruction, it will not overcome the "damage" which has been inflicted by injecting termination at the first position of X. □

Proposition 3.4 *If* `X` *features a failure then* `X` *contains a BB* ∞/∞*-Laski fault.*

Proof: The entire instruction sequence `X` can be taken for a fragment to be replaced by a correct implementation of `X`. The existence of such implementations is discussed in detail in [4]. □

Proposition 3.5 *If* `X` *features a failure then* `X` *contains a BB* $\mathtt{1}/(\mathtt{2n}+\mathtt{1})$*-MFJ fault.*

Proof: This is a simple rewording of Proposition 3.6 in [6]. □

Proposition 3.6 *Given natural* `n`*, there is an instruction sequence* `X` *which features a failure but which does not feature any BB* $\mathtt{1}/\mathtt{n}$*-MFJ fault.*

Proof: Take $\mathtt{k} > \mathtt{2}^{\mathtt{n}}$ and let ϕ be the function with `0` arguments and `k` `0`-initialized output registers which sets each output to `1`. Then take $\mathtt{X} \equiv !$. Now any change which turns ! into a correct implementation must contain at least `n` instructions for setting outputs equal to `1`, apart from the termination instruction. □

3.9 Discarding Residual Faults upon Resolution of Another Fault

We assume that `X` is an instruction sequence under construction (meant to implement specification P) which has arrived in a phase of quality control involving the spotting and resolution of faults. The (relevant part of the) software process database for `X` consists of the following three components:

(1) (successful test suite) a collection $\mathtt{V_{sts}}$ containing the tests which have been performed on `X` and on which `X` has succeeded, (i.e. those test inputs on which the computation of `X` has a result compliant with P).

(2) (detected failures) a collection $\mathtt{V_{fts}}$ containing the tests which have been performed on `X` and on which `X` has failed.

(3) (detected faults) a collection $\mathtt{Fault_{det}}$ of (descriptions of) WB $\mathtt{n}/\mathtt{m}$ SRT faults for various $\mathtt{n},\mathtt{m}$. Here it is assumed that the regression test suite used for the justification of SRT faults equals $\mathtt{V_{sts}}$.

It is assumed as an invariant for the software process (at least in the final stage of successive fault removal, the so-called debugging phase) that the (i) all tests in $\mathtt{V_{sts}}$ when applied once more end up successfully for `X`), (ii) the successful test suite is increasing during the design stages of `X`, while (iii) the collection of detected failures may both decrease (when a fault is resolved and its symptomatic failures disappear) and increase (upon performing a novel test thereby detecting a dormant failure).

The dynamics of the collection of detected faults is somewhat complicated. Clearly the collection increases if a new fault is detected. The collection of detected faults decreases first of all by resolving a fault. However, the collection of detected failures may decrease also, if, upon resolving a fault, its primary symptomatic failure constitutes a new test which is taken as an addition to the successful test suite. It may then happen that one ore more known (detected) faults cease to be valid faults (are "undetected" so to speak) because the proposed change fails on the new test. Assuming that a fault was invented (located and change proposed) in order to remedy its (detected) primary symptomatic failure the removal of the fault from the set of detected faults must be considered disappointing rather than reassuring because the search for a cause of its primary symptomatic failure may have to start again.

Given a stage of the software process including a collection pairwise disjoint detected faults (WB or GB with at least L and P made explicit), upon choosing one of the faults and resolving that fault according to the suggested change al other faults have the status of residual faults. Residual faults are candidate faults which may or may not on closer inspection qualify as (proper) faults. What has been argued in principle above is that upon resolving one of the faults another fault may turn into a residual fault which fails the (updated) regression test and for that reason does not qualify as a fault. In that case the residual fault is said to have been discarded as a consequence of the resolving another fault. Discarding a fault upon resolution of another fault is not a merely hypothetical phenomenon, as is illustrated by the following example.

Example 3.3 *The instruction sequence* `X` *below, considered as a candidate implementation of specification* ϕ*, features two disjoint GB(C)* `2/1`*-SRT faults such that after improving* `X` *according to the first fault the second fault does not pass the new regression test.*

For this example I will assume that ϕ requires that the function constant `1` is computed on a single input bit available under focus `in:1` and the output is delivered under focus `out0:1`. Assume that $\texttt{V}_{\texttt{sts}} = \emptyset$ and take `X` as follows:

```
%1  +in1.i/i;
%2  out0:1.1/1;
%3  #2;
%4  +out0:1.0/1;!;+in:1.i/i;+out0:1.1/0;!
```

Now `out0:1.1/1` is a WB `2/1`-SRT fault with location `2`, change `out0:1.1/0` and PSF on input `1`, moreover `#2` is a WB `2/1`-SRT fault with location `3` and change `#1` and a PSF on input `0`. Then upon resolution of the first fault (i.e. the one at position `2`) by effecting its change one obtains $\texttt{X}'$ as the instruction sequence under construction in the subsequent stage of the software process: $\texttt{V}_{\texttt{sts}} = \{1\}$.

The second fault then returns in the next stage of the software process with the status of a candidate fault: the PSF `0` is still valid because $\texttt{X}'$ fails on it, while subsequently changing according to the candidate fault (thereby obtaining $\texttt{X}''$ resolves that particular failure. However, the regression test fails for $\texttt{X}''$ on its (only) test at input `1`.

The disappearance of the second fault is no step forward from an engineering point of view because its primary symptomatic failure persists and another change, i.e. another GB(C) `2/1`-SRT fault for the same location should be found for its resolution (for which changing the 4th instruction to `+out:.0/1` suffices in this particular case).

Problem 3.1 *Is there an instruction sequence* `X` *which, when considered as a candidate implementation of a specification ϕ, features two disjoint BB* `2/1`*-SRT faults F and G for which it is the case that upon refining fault F to a WB* `2/1`*-SRT fault F', independently of how this is done (there may be several options), and after resolving fault F', fault G has lost its status of a detected fault because it has then become impossible to enrich it to a WB* `2/1`*-SRT fault (for the modified version of* `X`*).*

The class of `2/1` changes (that is replacing a single instruction by another single instruction) can be restricted by requiring that only parameters of an instruction are modified but the type of the instruction (termination, forward jump, backward jump, positive test, void, negative test) is left the same. I will refer to such changes as `2/1[pco]` changes (for parameter change only).

Now the following is easily checked:

Proposition 3.7 *For* `X` *and* ϕ *as in Example 3.3 both faults disjoint and are GB(L)* `2/1[pco]`*-SRT faults and upon resolving the first of both faults an instruction sequence results in which the second (now candidate fault) is not a GB(L)* `2/1[pco]`*-SRT fault anymore.*

4 Fault Qualification for SRT Based Fault Modalities

I will consider fault qualifications only for the case of SRT based fault modalities, which I consider to best connected to the original intuitions regarding these qualifications.

4.1 Qualification of GB(L,C,P) SRT Based Faults: Preliminary Remarks

WB fault modalities are the simplest among WB, GB, and BB because both CB and BB modalities involve non-trivial existential quantification and it is not immediately clear how to deal with those in a (rudimentary model of the) software process database.

Besides using the WB `n/m`-SRT fault modalities for various `n` and `m` for the formalisation of detected faults, I will consider the BB `n/m`-SRT fault modalities for modelling dormant faults. I hold that a BB `n/m`-SRT fault comes close to what a practitioner may have in mind when thinking of a yet undetected software fault (when specialised to instruction sequences), while a WB `n/m`-SRT fault is a fruitful approximation of what a practitioner may have in mind when contemplating a detected fault.

If one hopes to detect a treasure of coins in a backyard, then upon actually digging up the treasure, additional information about it becomes available. The notion of a dormant treasure is intuitively clear and so is the idea that upon detecting a treasure additional information about is discovered. In other words: the BB to WB transition which (given the definitions of dormant fault and of detected fault as presented below) takes place when detecting a fault, is to be expected rather than to be rejected.

4.2 Qualifications for BB/WB SRT Faults

The following qualifications apply at some stage `s` in the debugging phase of the software process for `X` understood as a candidate implementation of specification ϕ and with regression test set $\mathtt{V}^{\mathtt{s}}_{\mathtt{sts}}$, collection of detected faults $\mathtt{V}^{\mathtt{s}}_{\mathtt{fts}}$, and a collection $\mathtt{Fault}^{\mathtt{s}}_{\mathtt{det}}$ of WB $\mathtt{n}/\mathtt{m}$-SRT faults (for $'\mathtt{X}$ relative to ϕ) for various natural numbers $\mathtt{n} > 1, \mathtt{m} > 0$.

1. *non-detected fault*: a non-detected SRT fault is a BB $\mathtt{n}/\mathtt{m}$-SRT fault such that for no $\mathtt{n}' < \mathtt{n}$, a WB $\mathtt{n}'/\mathtt{m}$-SRT fault with the same location is contained in $\mathtt{Fault}^{\mathtt{s}}_{\mathtt{det}}$. (In other words: no WB version of the fault, or of any fault prior to such a fault, is included in the collection of detected faults.)

2. *effective fault*: A BB $\mathtt{n}/\mathtt{m}$-SRT fault F is said to be effective (i.e. to have been activated) if some enrichment F' of it to a GB(C) $\mathtt{n}/\mathtt{m}$-SRT fault exists with the property that at least one of the failed tests in $\mathtt{V}^{\mathtt{s}}_{\mathtt{fts}}$ counts as a symptom for F' (i.e upon effectuation of the change of F' that particular failure, which had been detected already, is resolved).

3. *dormant fault*: a BB $\mathtt{n}/\mathtt{m}$-SRT fault F is dormant if it is both non-detected and not effective.

4. *detected fault*: a WB $\mathtt{n}/\mathtt{m}$-SRT fault is detected if either it is included in $\mathtt{Fault}^{\mathtt{s}}_{\mathtt{det}}$ or a fault prior to it is included in $\mathtt{Fault}^{\mathtt{s}}_{\mathtt{det}}$. It is assumed that only effective faults are considered to be detected (i.e. detection requires a test which demonstrates that the input of the primary symptomatic failure of the fault indeeds leads to a failure).

5. *temporary fault*: given a detected fault, the plan may be made to resolve the fault by making an appropriate change. If such a plan has been made the fault is considered temporary.

6. *permanent fault*: given a detected fault, the plan may be made not to resolve the fault by means of a modification of the instruction sequence. Once the decision has been taken not to resolve the fault one either acknowledges that the fault is permanent, or otherwise that the fault will be discarded, i.e. that the specification is updated rather than its candidate implementation.

7. *retrospectively discarded fault*: A permanent fault may become included in the approved form of $\mathtt{X}$ by changing the specification and the design design in hindsight accordingly.

These qualifications are given in the absence of any explanation of how fault detection actually works. In fact fault detection is unlikely to be primarily based on the above definitions.

Claim 4.1 *Fault detection is the process of finding WB* `n`/`m`*-SRT faults which are qualified as detected (in the sense of the descriptions just given). Fault detection will need to make use of some higher level of description than mere instruction sequences (that is, of some form of design or specification), and either specifications (formalisations) of ϕ or designs (for* `X`*) or both will play a critical role in the process of fault detection.*

All instruction sequence faults are instruction sequence defects, but not the other way around. I will next proceed to complement the discussion of defects which are faults with a discussion of defects which may not qualify as faults, that is defects which may but need not cause one or more failures. For so-called deviation from design defects (DFD defects), a particular class of defects which is elaborated below, useful heuristics concerning the design of defect resolution methods can be obtained. Faults which are DFD defects at the same time admit promising detection strategies.

5 Potentially Non-Faulty Instruction Sequence Defects; DFD-defects

Given a specification ϕ which determines a unique partial function P_ϕ, and a candidate implementation `X` of it. A defect of `X` is any feature (property) of it which can be held against in its intended quality of being an instruction sequence which computes the partial function P_ϕ. Clearly the presence of failures indicates the existence of one or more defects. In fact "the existence of failures" itself constitutes a defect. However, "the existence of failures" does not qualify as a fault because there is no causal relation between such a general claim and the occurrence of any particular failure.

I will assume that `X` has been constructed with a design $\mathtt{D}_\phi$ as an intermediate stage. $\mathtt{D}_\phi$ may have many forms, among which a pseudocode in some high level specification notation. Now it may be the case that it is obvious upon inspection of `X` and $\mathtt{D}_\phi$ and given an instruction counter `n` that some fragment `Y` of length `k` of `X` beginning with the `n`-th instruction $\mathtt{u_n}$ of `X` is "problematic" in the following sense: `Y` does not reflect the intentions embodied in $\mathtt{D}_\phi$ to such an extent that it is clear how to change `X` by replacing the fragment $\mathtt{Y} = \mathtt{u_n};\ldots;\mathtt{u_{n+k}}$ of `X` by a fragment $\mathtt{Y'} = \mathtt{v_n};\ldots;\mathtt{v_{n+k'}}$ in such a manner that this particular mismatch between `X` and its purported design $\mathtt{D}_\phi$ does not occur anymore after the change has been made.

Here it is understood that given instruction counter `n` (i.e. a location in `X`) a corresponding location in $\mathtt{D}_\phi$ is easily spotted and moreover that a mere inspection of X and D_ϕ at the respective locations provides the insight that something is wrong for which a local fix in the form of a change Y′ is easy to find.

The fragment X constitutes, or contains, a defect which, however, need not necessarily be the cause of a failure. This type of defect may arise even when the candidate implementation is correct. For that reason the defect is not necessarily a fault, though it may well be. I will refer to a defect as mentioned above as a DFD defect (deviation from design defect).

Definition 5.1 *A BB DFD defect of an instruction sequence* X *w.r.t. a design* D_ϕ *is a fragment* Y *of* X *which causes the presence of a deviation of* X *understood as a candidate implementation of* D_ϕ *from the construction requirements as embodied in* D_ϕ*. Here causality is understood in terms of allowing a remedy by enacting a change of the fragment.*

If in addition to the fragment which is considered non-compliant with the design a proposed change is available which resolves that particular instance of non-compliance I will speak of a WB DFD defect. Both BB DFD defects and WB DFD defects come in three flavours: Laski, MFJ, and SRT.

To emphasise that the BB DFD defect is apparent even without contemplating options for change or improvement it may be referred to as a TMO DFD defect (TMO for textual mismatch, with the design, only).

5.1 Design Notations

Design notations exist in many forms, with UML diagrams as a famous example which admits many dialects and versions. Close to the instruction sequence I am using are flow-charts, and algorithms as understood in [13]. I will comment on both options.

5.1.1 Flow-charts as a Design Notation

Flow-charts may contain basic instructions `f.m` with a single outgoing arrow, in case the returned Boolean value is not made use of, or with two outgoing arrows labeled with $+$ and $-$ respectively. Termination is a box with ! in it and arrows represent jumps.

A flow-chart may be turned into a PGLB instruction sequence in many different ways. The textual order of basic instructions provides a degree of freedom, and so does the sign used for test instructions. Given a PGLB instruction sequence $\mathtt{X}$ it is straightforward to assign a unique flow-chart $\mathtt{D}_{\mathtt{X}}$ to it. Given a design $\mathtt{D}$ in the form of a flow-chart I will say that $\mathtt{X}$ implements $\mathtt{D}$ if $\mathtt{D} \cong \mathtt{D}_{\mathtt{X}}$. Important aspects of this example of a design format are: (i) there are many ways to implement a design, (ii) if an instruction sequence does not implement a design then an instruction can be found which does not implement the corresponding fragment of the flow-chart, (ii) in some cases a local change may resolve the problem, in part (MFJ style or SRT style) or completely (Laski style).

5.1.2 Algorithm Documenting Instruction Sequences

In [13] the same assumptions are made as in 1.2 above, except that auxiliary single bit registers may be used. Consider two instruction sequences $\mathtt{X}$ and $\mathtt{Y}$ which compute the same functions from bit sequences to bit sequences. Now it is claimed in [13] that if after appropriate renaming of auxiliary registers of $\mathtt{X}$, and after swopping the content of one or more of the auxiliaries, an instruction sequence $\mathtt{X}'$ is obtained so that $|\mathtt{X}'| = |\mathtt{Y}|$, i.e. $\mathtt{X}'$ and $\mathtt{Y}$ determine the same thread then $\mathtt{X}$ and $\mathtt{Y}$ may be considered algorithmically equivalent.

In this case it may be appropriate to consider $\mathtt{Y}$ a design for $\mathtt{X}$ in the sense that it documents an algorithm for which $\mathtt{X}$ is supposed to be an implementation. It would be too optimistic to regard this definition of algorithmic equivalence as the last word on that matter, because, quite obviously this equivalence relation is too coarse to capture the intuition of algorithmic equivalence (whatever that intuition may be, it will be too coarse for anyone).

Suppose that $\mathtt{X}_1$ and $\mathtt{X}_2$ are candidate implementations of the design $\mathtt{Y}$. Now it is not yet assumed that these instruction sequences compute the same function. The following notion of improvement is plausible in this setting. (1) find $\mathtt{X}_1'$ using renaming and content swapping of auxiliary single bit service so that in for as large as possible $\mathtt{n}$ (if not for all $\mathtt{n}$) $\pi_{\mathtt{n}}(|\mathtt{X}_1'|) = \pi_{\mathtt{n}}(|\mathtt{Y}|)$, and do the same for and $\mathtt{X}_2$. Now if for some $\mathtt{k}$, say $\pi_{\mathtt{k}}(|\mathtt{X}_2'|) = \pi_{\mathtt{k}}(|\mathtt{Y}|)$ while not $\pi_{\mathtt{k}}(|\mathtt{X}_1'|) = \pi_{\mathtt{k}}(|\mathtt{Y}|)$ then $\mathtt{X}_1$ constitutes an improvement (as an implementation of the design $\mathtt{Y}$) than $\mathtt{X}_2$. I will say that the improvement is an improvement of DOAA (depth of algorithmic approximation). Thus DOAA is a notion of improvement for implementations of a certain class of designs.

DOAA is by no means the only notion of improvement available, and contemplating these matters in more general terms makes sense.

Claim 5.1 *Each notion IMPR of improvement for implementations of de-*

signs comes with a corresponding notion of BB `n/m`*-IMPR DFD defect: a fragment of a candidate implementation* `X` *of the design with at most length* `n` *where a change can be applied, introducing at most* `m` *additional instructions, so that the resulting candidate implementation* $\mathtt{X}'$ *is an improvement in the sense of IMPR of* `X`*.*

So one is led to contemplate BB `n/m`-IMPR DFD defects. Apart from the technical merits of this defect modality, about which one can only say that it remains to be seen, its very existence and plausibility indicates that IMPR is a parameter of defect modalities which can hardly be left implicit.

5.2 Classification of Functional Improvement Based DFD Defects

Repair in case of Laski faults, MFJ faults and SRT faults consider improvement in terms of achieving an improved input output behaviour. When repairing an SRT fault the improvement may be merely observed by lack of a sufficiently extensive regression test suite, but the objective to achieve better input output behaviour is present in the removal (resolution) of at least the primary symptomatic failure. I will label such improvements as functional. DOAA in contrast is an algorithmic notion of improvement.

For various DFD defects the question arises how the validity of changes is confirmed, in excess of the change providing better compliance with the given design. I will distinguish three options, the naming of which follows the naming of fault modalities mentioned above:

TMO DFD defect: a TMO DFD defect merely consists of a fragment of an instruction sequence for which it is considered implausible that it constitutes a proper implementation of the corresponding part of the design. The implausibility is, however, not argued for by the proposal of a change.

WB Laski DFD defect: a WB Laski DFD defect comes with a change (C) which is supposed to be inferred in a straightforward manner from the design, and which when applied, in addition to removing the known DFD defect, creates a correct implementation of the given specification (where the component (P/E) informs about the inference method).

WB MFJ DFD defect: a WB MJF DFD defect comes with a change (easily inferred from the design and the location and size of the defect) such that, in addition to removing the known DFD defect, the changed instruction sequence computes a correct output whenever the original instruction sequence does so (where again the P/E component of the fault informs about the inference method).

WB SRT DFD defect: A WB SRT DFD defect comes with a change (easily inferred from the design and the location and size of the defect) which, in addition to removing the known DFD defect, passes all tests in the current regression test suite.

In addition a WB SRT DFD comes with a new test, which is added to the test suite, the computation of which passes through the modified instruction(s).

BB Laski DFD defect: a BB Laski DFD defect results from a WB Laski DFD defect by forgetting/hiding the change and related P/E. A BB Laski DFD defect comes with the claim that its enrichment to a WB Laski DFD defect is possible.

BB MFJ DFD defect: a BB MFJ DFD defect results from a WB MFJ DFD defect by forgetting/hiding the change and related P/E. A BB MFJ DFD defect comes with the claim that its enrichment to a WB MFJ DFD defect is possible.

BB SRT DFD defect: a BB SRT DFD defect results from a WB SRT DFD defect by forgetting/hiding the change and related P/E. A BB SRT DFD defect comes with the claim that its enrichment to a WB SRT DFD defect is possible.

I will assume that information about how to resolve a defect is not essential for confirmation of a defect. This assumption indicates a huge difference with faults where any fault must allow an improvement by definition.

Assumption 5.1 *Each BB or WB (Laski or MFJ or SRT) DFD defect from the above listing contains an underlying TMO DFD defect.*

5.3 Size Bounds for Defects and Corresponding Changes

Parameter for bounds on the size of a defect and of a change for its resolution are indicated just as for faults. Thus a WB 5/5-SRT DFD defect has size at most 5 and involves a change of length at most 5 instructions more than the defect.

An $\mathtt{n}$-Laski/MFJ/SRT DFD defect with $\mathtt{n} \in \mathbb{N} \cup \{\infty\}$ is a defect (of say $\mathtt{X}$) where the (candidate) defective fragment $\mathtt{Y}$ has a size ($\mathtt{LLOC}$ for logical lines of code) of less than $\mathtt{n}$ instructions.

An $\mathtt{n}/\mathtt{m}$-Laski/MFJ/SRT DFD defect with $\mathtt{m} \in \mathbb{N} \cup \{\infty\}$ is an $\mathtt{n}$-Laski/MFJ/SRT defect where the proposed change $\mathtt{Z}$ has size less than $\mathtt{LLOC}(\mathtt{Y}) + \mathtt{m}$.

5.4 Commented Examples

A vast space of defects modalities now arises, I will give some examples.

- an WB $\infty/\mathtt{10}$-Laski(MPG/MPC) DFD defect is: a deviation from the design of arbitrary length for which a change is known (and can be inferred from its location and the design) which is at most 10 instructions longer, and which resolves the design fault and which in addition turns the instruction sequence at hand to a correct implementation the latter fact being supported by a manually generated proof which has been checked manually as well.

- A BB $\mathtt{10}/\mathtt{10}$-SRT DFD defect is: a fragment of size $\mathtt{k} < \mathtt{10}$ with a known but undisclosed change of size below $\mathtt{k} + \mathtt{10}$, where after application of the change regression testing confirms compliance with the current regression test suite. Moreover it is required that given the location of the defect the change was found from inspection of the design with a reasonable effort.

- A TMO $\mathtt{10}$-DFD defect is: a fragment of size below $\mathtt{10}$ which manifestly does not comply with the corresponding fragment in the design.

An algorithmic improvement need not be a functional improvement. Let $\mathtt{Y} = \mathtt{out0{:}1.1/1}; \mathtt{out0{:}1.1/0}; !$ be a documenting instruction sequence serving as a design with $\mathtt{X} = +\mathtt{out0{:}1.0/1}; \mathtt{out{:}.1/1}; !$ as a candidate implementation for it.

Consider $\mathtt{X}' = \mathtt{out0{:}1.1/1}; \mathtt{out{:}.1/1}; !$ which results by replacing the first instruction of $\mathtt{X}$ thereby obtaining $\mathtt{X}'$. This situation features a WB $\mathtt{2}/\mathtt{1}$-DOAA DFD defect (the first instruction as the fragment and $\mathtt{out0{:}1.1/1}$ as its change) but the same fragment and change do not provide a a WB $\mathtt{2}/\mathtt{1}$-SRT DFD defect for any non-empty regression test set. Clearly $\mathtt{X}$ contains a WB $\mathtt{3}/\mathtt{2}$-SRT DFD defect, the change of which consists of replacing the first two instructions of $\mathtt{X}$ by the first two instructions of $\mathtt{Y}$.

One may take an interest in WB `n`/`m`-SRT DFD defects which are WB `n`/`m`-DOAA DFD defects at the same time (w.r.t. the same change). Such defects belong to yet another fault modality: WB `n`/`m`-SRT/DOAA defect

5.5 Impact of Functional DFD Defect Resolution on Other DFD Defects

Performing a change in an instruction sequence may have upgrading of defects as a consequence. A defect is upgraded if a more conclusive resolution is possible, where resolving a Laski defect is the most conclusive resolution because it leads to a correct implementation of the given specification.

Proposition 5.1 *Given design* `D` *and a candidate implementation* `X` *of it which contains no overlapping TMO DFD defects and assuming that* `X` *contains an TMO DFD defect* `d`*, then upon making the change which removes* `d` *each of the following side-effects on another defect may take place:*

(i) another TMO DFD defect becomes upgraded to BB SRT DFD defect status (or to BB MFJ DFD defect status or to BB Laski DFD defect status),

(ii) a BB SRT DFD defect is upgraded to a BB MFJ DFD defect status or to BB Laski DFD defect, or

(iii) a BB MFJ DFD defect is upgraded to a BB Laski DFD defect.

Proof: I will assume that the design notation extends the instruction sequence notation so that an instruction sequence can be considered to be a design of itself. Consider a situation where the constant function `1` is computed on a single input bit placed in `in:1`. Let this requirement be formalised with specification ϕ. Let $\texttt{X} = !; +\texttt{in:1.0/i}; \texttt{out0:1.1/1}; !$. I notice that `X` does not implement ϕ because it fails on input 1. Now consider $\texttt{D} = \#\texttt{1}; +\texttt{in:1.1/i}; \texttt{out0:1.1/1}; !$ which is taken for a design for implementations of ϕ, and `X` is contemplated as a candidate implementation for it. Manifestly the first instruction ! of `X` constitutes an TMO DFD defect in `X`. At the same time none of the other instructions constitute faults of `X`, as no change undoes the initial termination instruction. The obvious resolution of the TMO DFD defect results by replacing it by $\#\texttt{1}$ thus following exactly the given specification, thereby obtaining $\texttt{X}' = \#\texttt{1}; +\texttt{in:1.0/i}; \texttt{out0:1.1/1}; !$. $\texttt{X}'$ fails even on both inputs. However, given ϕ it is now the case that the instruction $+\texttt{in:1.0/i}$ has become a BB Laski DFD defect because it can be changed to $+$`in:1.1/i` upon which the resulting instruction sequence computes both outputs correctly.

One may also hold that $+$`in:1.0/i` has become an MFJ DFD defect because it can be improved to $+$`in:1.i/i` which computes one of the outputs correctly (i.e. for input `1`) which is better than nothing, and which does not introduce any new failures in comparison to `X`$'$. □ The following question depends on the details of the notion of SRT DFD at hand.

Question 5.1 *Let* `V` *be a given regression test suite. Suppose that* `X` *is a candidate implementation of design* `D` *for specification* ϕ *which contains (at least) two disjoint WB* `2/1`*-SRT DFD defects (w.r.t.* `V`*). Can it be the case that upon changing* `X` *according to one of the two defects, the other defect ceases to be a WB* `2/1`*-SRT DFD defect (i.e. upon performing the second change, compliance with the design will be improved, but nevertheless the regression test then fails on the new test which was included in the test suite upon repairing the first defect)?*

One may take instruction sequences for designs (trivial designs in fact) which are preferably to be implemented by the same instruction sequence. Better implementations are identical (with the instruction sequence as the designs) on more positions and have the same number of instructions (the difference in length is counted as different instructions). I will refer to this notion of improvement as TRIV-DES. Consider design `Y` as follows:

```
Y =
%1   #1; aux0:1.1/1; −in:1.i/i; #3; out0:1.1/1; !;
%6   + aux0:1.i/i;
%7   !; !
```

`Y` computes the identity function: `in:1` is copied into `out0:1`. For `Y` consider as a candidate implementation `X` with:

```
X =
%1   #2; aux0:1.1/1; −in:1.i/i; #3; out0:1.1/1; !;
%6   − aux0:1.i/i;
%7   out0:1.1/0; !
```

and regression test set $\{1\}$. Now `X` contains three WB `2/1`-SRT/TRIV-DES DFD defects:
d1 on the `1`-th instruction with change #`1`,
d2 on the `6`-th instruction with change $-$`aux0:1.1/1`,and
d3 on the `7`-th instruction with change !,

After the change according to d1, the residue of d2 is not a WB `2/1`-SRT/TRIV-DEFS DFD defect anymore because it will fail on input `1`.

The third defect is necessary for the example because after two repairs there must still be a failure on input `1`, which can only be the case if there are still one or more TMP TRIV-DES DFD faults left.

5.6 Bugs are Defects Which May be Faults

Causation of failures is a curious matter. If a programmer makes a mistake while constructing an instruction sequence, thereby creating an instruction sequence which contains a fault which in turn (and by definition of fault) causes a failure, said mistake is also a cause of the failure, provided one agrees that causation is transitive. It appears that not all causes of failures are faults. Instruction sequence faults are a feature of an instruction sequence proper, not of the way it has come about.

Conceptual proposition 5.1 *An instruction sequence fault is an instruction sequence defect which causes at least one failure.*

Dijkstra famously wrote in 1970 [18] (p7):

> Program testing can be used to show the presence of bugs, but never to show their absence.

I will first turn this quote into a claim which is supported by the definition of failure.

Claim 5.2 *Instruction sequence testing can be used to show the presence of failures, but almost never provides a feasible method to show their absence.*

That testing may demonstrate the presence of a failure, assuming that the oracle problem has been solved is obvious, and the second part of the claim states that what is unfeasible because of a combinatorial explosion of different inputs is qualified as impossible. Most programs are written in order to process so many different inputs that exhaustive testing is out of the question.

Returning to Dijkstra's quote, I notice that the notion of a bug comes with the intuition of locality: the bug is somewhere in an instruction sequence. I will assume that bugs are defects, i.e. that the informal notion of a bug is made precise by the somewhat less informal notion of a (DFD) defect. The following claim simply defines bugs as faults.

Claim 5.3 *(Bug) (i) All instruction sequence bugs are instruction sequence defects, (ii) some bugs are faults as well, (iii) once a bug has been spotted* *its resolution is easy (so bugs areTMO DFD defects with strong guidance on how to improve design compliance).*

For bugs which are not faults (i.e. do not cause a failure, a state of affairs for which a resolution by way of a local change can be found) testing is uninformative about presence or absence.

For bugs which are faults at the same time, however, the plausibility of detection via testing is highly debatable too. Following idea of an ALR fault, it is a critical feature of a fault that it causes a failure. Now it is far from clear how to demonstrate causation by means of testing. At first sight that cannot be done: evidence for causation of a specific failure by a fault is obtained by demonstrating that a resolution of the fault can remove the failure while doing not much harm to the processing of other inputs. Implicitly assuming the ALR ideology about faults the following claim results:

Claim 5.4 *The claim that instruction sequence testing can demonstrate the presence of instruction sequence faults requires a perspective on testing which involves mechanisms for the following tasks:*

1. *spotting a failure (called the symptomatic failure),*
2. *finding a change for resolving a fault in case the fault does not come with a proposed change,*
3. *fault resolution by applying the change at hand, including obtaining confirmation of the fact that the symptomatic failure has been dealt with adequately (i.e. has been removed), and,*
4. *obtaining satisfactory validation for the changed instruction sequence for inputs different from the input for the symptomatic failure.*

The literature on program testing is insufficiently uniform to find out whether or not it is reasonable to give testing the scope and breath as indicated in Claim 5.4. Some authors subsume all activity focused on asserting or improving program quality under testing, some authors consider the software process rather than the delivered software as the target of improvement for testing. I will provide a definition for a brand of testing which may be far to limited for the taste of some, and perhaps even too liberal for the taste of others.

Definition 5.2 *(White box [for instruction counting] output oriented testing) Instruction sequence testing as applied to a candidate implementation* `X` *of specification ϕ consists of putting* `X` *into effect on a family of inputs and finding out (i) whether or not the resulting computation complies with the given requirements, (ii) whether or not during the run a certain instruction in* `X` *has been reached.*

Using white box [for instruction counting] output oriented testing amounts to (i) generating testsuites. (ii) experimenting with the various tests in the test suites, (iii) finding and including additional tests in order to obtain sufficient coverage (that requires some white box permission), and (iv) drawing conclusions (either about `X`, or about its construction process) on the basis of the results thus obtained.

Now finding a change, applying the change to an instruction sequence, and testing the changed instruction sequence to the point of obtaining validation for results outside the symptomatic failure each constitute activities which arre not subsumed under testing as defined in Definition 5.2. Under these assumptions, in contrast with Dijkstra's quote, the following claim has become plausible:

Claim 5.5 *White box [for instruction counting] output oriented testing cannot, in general, be used to demonstrate the presence of instruction sequence faults.*

5.7 DFD Defect Qualifications for Instruction Sequences

Defect qualifications regarding `X` as a candidate implementation of the function P are given relative to a design `D` for such implementations. Defects are about knowledge of a designer/programmer rather than about mathematical fact. Just as for failures and for faults a very simple model of a knowledge base is helpful. For each of the kinds of defects listed above one one may imagine a format for the description of such defects. I will assume that defects are collected about an instruction sequence with name `X`. I also restrict attention to SRT modalities, just as was done in the case of faults

As stage `s` of the software proces for the construction of `X` the software process database is supposed to a collection $\mathtt{Defect}^{\mathtt{s}}_{\mathtt{d}}$ of textual descriptions `text(d)` of WB `n`/`m`-DFD SRT defects `d` for various `n`, `m`. Moreover the database contains a collection $\mathtt{V}^{\mathtt{s}}_{\mathtt{sts}}$ of (inputs for) successful test which will serve as the regression test suite. It is assumed that it can be and has been confirmed in each case that $\mathtt{d}$ describes an TMO DFD defect of $\mathtt{X}$. It is possible that defects overlap in which case resolving one defect by changing a fragment of $\mathtt{X}$ may or may not resolve (or do away with in another manner) the other defect as well.

Definition 5.3 *Given two defects* $\mathtt{d}_1$ *and* $\mathtt{d}_2$ *where* $\mathtt{d}_1$ *is an TMO* $\mathtt{n}_1$*-DFD defect of* $\mathtt{X}$ *and* $\mathtt{d}_2$ *is an TMO* $\mathtt{n}_2$*-DFD defect of* $\mathtt{X}$ *with* $n_1 < n_2$ *then defect* $\mathtt{d}_1$ *is said to be prior to defect* $\mathtt{d}_2$ *if the corresponding fragments of* $\mathtt{X}$ *have the same initial position.*

These qualifications are given from the perspective of a single user A who is also doing maintenance on $\mathtt{X}$ including defect resolution. Let $\mathtt{X}$ be the $\mathtt{ISuc}$.

1. *dormant defect*: a dormant TMO DFD defect is an TMO DFD defect which is not the TMO version of a WB SRT DFD defect that is contained in $\mathtt{Defect}^{\mathtt{s}}_{\mathtt{det}}$, and such that no TMO version of a WB SRT DFD defect in $\mathtt{Defect}^{\mathtt{s}}_{\mathtt{det}}$ is prior to it.
2. *detected defect*: a detected WB $\mathtt{n}/\mathtt{m}$-SRT DFD defect is an WB $\mathtt{n}/\mathtt{m}$-SRT DFD defect which either is contained in $\mathtt{X}$ in $\mathtt{Defect}^{\mathtt{s}}_{\mathtt{det}}$ or for which another defects which is prior to $\mathtt{d}$ is contained in$\mathtt{Defect}^{\mathtt{s}}_{\mathtt{det}}$.
3. *temporary defect*: given a detected defect the plan may be made to resolve the defect by making an appropriate change. If that happens the defect is considered temporary.
4. *permanent defect*: given a detected defect, the plan may be made not to resolve the defect by means of a modification of the instruction sequence. In this case the defect is permanent.
5. *retrospectively approved permanent defect*: A permanent defect may become part of the approved form of $\mathtt{X}$ by changing the design in hindsight accordingly.

I refrain from defining qualifications for DFD defects with either explicit or implicit change information. For instance it is far from clear what a dormant BB $\mathtt{5/10}$-DFD Laski (MPG/MPC) defect might be. This notion introduces an existential quantifier over changes and over the existence of corresponding proofs of the MFJ property and checks of such proofs, all carried out manually.

6 Design Related Aspects of the Software Process

I imagine an instruction sequence `X` which is in the proces of being constructed. Once a first complete version of `X` has been completed, i.e. a candidate implementation of the given specification has been developed, the process of finding and repairing changes and defects begins. Changes modify candidate implementations step by step to better candidate implementations until the construction project is either abandoned or is finalised with a first version of a product ready for use, with the understanding that the proces of fault/defect finding and resolution will be continued during use of the software product at hand.

It may be so that a programmer has technical design at hand. Otherwise it is reasonable to suppose that the programmer avails of a mental design, that is a design which has not been constructed as an independent artefact and yet exists in the mind of the programmer. DFD defects can be imagined with respect to a mental design as well.

A plausible way for a programmer to find a (functional) SRT DFD defect as well as a suitable proposed change for it is as follows:

(i) Use some method to spot one or more locations where a defect or a fault may be found. For instance code walk-through may help to spot locations of DFD defects and so may automated detection of code smells. If a failure has been observed the computation may be followed step by step (so-called debugging) and one or more locations visited by the computation may be considered candidates for the location of defects or of faults. Instead of looking through the entire computation for a single failure, spectrum based fault localisation may be used to focus more quickly on "promising" locations for candidate faults,

(ii) Then one expects to find a TMO DFD defect at or briefly after the first of the spotted (candidate) locations. (If, unexpectedly, a TMO DFD defect is not spotted, the design may be called into question depending on the evidence of the presence of a fault on or near that location).

(iii) Subsequently the design is used to derive a change which remedies the DFD defect, so that a candidate WB SRT DFD defect has been found,

(iv) Finally to run the regression test suite on the changed version of `X` in order to confirm that the candidate defect is a proper WB SRT DFD defect.

6.1 Syntactical/Grammatical Faults Treated as Ordinary Faults

The situation with semantic faults differs from syntactic (grammatical) faults as follows: for a text which is considered to be syntactically faulty it is usually possible to determine one or more smallest fragments which are (syntactically) incorrect by any standards, i.e. in no context said fragment would be part of a correct expression. Unlike semantical faults grammatical faults are very serious problems which persist in each context. Speaking of a grammatical fault rather than a grammatical defect is justified if one assumes that running a program with a grammatical fault simply terminates with an error message, say by setting a designated bit to `1` leaving all others unchanged.

6.2 Inadequate Designs and Local Design Inadequacies

I will not be specific about a syntax or form of designs, but merely assume the existence of designs for instruction sequences, which are human readable texts from which human software engineers may construction instruction sequences, possibly with the help of various software tools. The notion of a design enters the discussion of faults because DFD defects, a notion which requires some idea of design, play a key role in the detection and resolution of faults.

An inadequate design is a design from which a correct implementation cannot be obtained, where an implementation of a design is understood as follows:

Definition 6.1 *(Implementation of design) Given a design* `D`*, an implementation of* `D` *is an instruction sequence* `X` *which has been constructed according to design* `D` *in such a manner that* `X` *contains no TMO DFD defects (w.r.t.* `D`*).*

A design need not uniquely determine its implementations. However, I will assume that the existence if implementations of designs is given by the syntax of designs. Existence of implementations of designs is independent of any notion of specification, and merely indicates that by construction expressions in the design language which is used are consistent.

Conceptual proposition 6.1 *Every design has at least one implementation. A professional software engineer (team of engineers) is always able to construct an implementation of a given design.*

I recall that, under the simplifying conditions as listed in Paragraph 1.2 above, a specification is supposed to determine a relation which specifies a class of partial functions (maximal functional relations contained in ϕ) one of which is to be computed by any implementation of ϕ. I think of a specification as a formal specification, for instance a first order formula over the two-element structure of bits, perhaps making use of expressive power brought about by admitting auxiliary sorts for various parametrised data types.

Definition 6.2 *(Adequate design.) Given a specification ϕ, a design $\mathtt{D}_\phi$ for an implementation of ϕ is adequate if there is an instruction sequence* $\mathtt{X}$ *which (i) is an implementation of* $\mathtt{D}_\phi$ *and (ii)* $\mathtt{X}$ *computes* $\mathtt{P}_\phi$*.*

Definition 6.3 *(Complete design) A design $\mathtt{D}_\phi$ is a complete design if all implementations of it compute the same partial function, as specified by P_ϕ.*

In practice it is implausible for a design to be complete. In particular if a design focuses on architecture while leaving the details of various components for being worked out in the stage of instruction sequence construction, then completeness is not to be expected.

Definition 6.4 *(Inadequate design) Given a specification ϕ, the design $\mathtt{D}_\phi$ is inadequate if there is no instruction sequence* $\mathtt{X}$ *which (i) is an implementation of* $\mathtt{D}_\phi$ *and (ii)* $\mathtt{X}$ *computes* $\mathtt{P}_\phi$*.*

Definition 6.5 *(Local design inadequacy.) Given a specification ϕ, and an inadequate design $\mathtt{D}_\phi$ for an implementation of it, a local inadequacy in the design $\mathtt{D}_\phi$ is a fragment of it which can be replaced by another fragment in such a manner that the resulting design is not flawed anymore.*

From a practical perspective the notion of an informal specification is missing from our discussion. One may think of that omission as being of marginal importance. However, we were definitely unable to find a role for informal specifications in addition to this account of artefacts which enter the process of instruction sequence construction.

Problem 6.1 *How to include the notion of an informal specification in the account of (formal) specifications, designs, and implementations.*

The difficulty seems to be connected with the observation that coexistence of formal specifications and informal specifications is uncommon, so that the introduction of informal specifications would plausibly go hand in hand with forgetting about formal specifications. But in the absence of formal specifications and of derived formal semantics, the various notions of fault and defect as discussed in this paper cannot be defined. Notions of design adequacy and inadequacy as introduced above also depend on the availability of formal semantics.

I will refer to the activity of constructing an instruction sequence as instruction sequencing. Upon constructing an implementation of the design a local design inadequacy can give rise to (cause the existence of) a fault in the resulting instruction sequence. The act of faithfully implementing the (locally inadequate) design is a mistake which causes the presence of a fault. The mistake itself is caused by the presence of said local inadequacy in the design. In other words:

Conceptual proposition 6.2 *Faithfully following a locally inadequate design during instruction sequencing may lead to making a mistake (of instruction sequencing), which in turn may cause the presence of a fault in the resulting instruction sequence.*

Qualifications of local inadequacy admit convincing informal definitions. Indeed any specific local inadequacy of a design `D` can be qualified as follows:

1. dormant: the designer is unaware of the specific local inadequacy of the design.

2. detected: the designer is aware of the specific inadequacy of the design, which will be argued for by fairly informal arguments,

3. temporary: the designer knows about the local inadequacy and has made the plan to locally modify the design in such a manner that said local inadequacy disappears,

4. permanent: the designer accepts the local design inadequacy as a fact of life for the future,

5. retrospectively approved: the designer concludes that after all it would be better to modify the formal specification in such a manner that what was a local design inadequacy, is not anymore an inadequacy.

6.3 Design Correctness and Design Defects

In the absence of a notation for designs and a definition of refinement which explains how design and implementation are to be related it seems to be impossible to provide valid definitions of: design correctness, and design defect. At the same time these notions seem to make perfect sense intuitively, perhaps these notions are meaningful for all artefacts with an instrumental status.

Nevertheless given a notion of correctness some notion of a Laski defect for a design may be proposed (a local change leading to a correct design), and given a notion of improvement for designs some notion of an MFJ defect can be proposed (a local change leads to an improved design), while SRT errors make sense only if the design is executable itself. An informal definition of a design defects which provides no explanation of the notion of causation reads as follows:

Definition 6.6 *(ALR style design defect) A design defect is a fragment of a design which, upon implementation of the design causes a mistake to be made, that is it causes a fault to come about in its implementation. (This definition requires that a specification is available which implementations of the design are supposed to comply with, and which allows to speak of faults in candidate implementations.)*

However, this paper relies on the following simplifying assumption.

Conceptual proposition 6.3 *It is possible and plausible to develop a theory of instruction sequence faults and defects making use of a notion of design that is equipped with a notion of local design compliance which applies to fragments of an instruction sequence serving as a candidate implementation of the design, as on that basis one may speak of a TMO DFD defect.*

However, for the stated purpose (that is: developing a theory of instruction sequences faults and defects), it is not required that the notion of design is itself equipped with notions of correctness and defect.

7 Concluding Remarks

In the words of one of the reviewers: "..often we think of some instruction as a fault because we think we know the programmer's intention, and we recognize that what is written does not reflect the programmer's intention.

Such a definition is only as good as our ability to second-guess the programmer." By consequence there is no way around a more rigid approach to the notion of a faulty instruction. I made some progress in that direction in the paper.

The relation between instruction sequence faults and design methods requires further attention. A classical idea is top-down design, and this idea directly relates to the concept of fault. Indeed one may hold that top-down design and systematic use of software architecture give rise to the intuition of a fault. If one knows that refining a certain architecture must do the job, and as a matter of fact it does not, then local changes must suffice to obtain a correct program. In other words, structured programming and top-down design, nowadays cast in terms of designing on the basis of an architecture, so to say create the concept of a fault (understood as a local option for solving the occurrence of one or more failures) by giving rise to ever smaller units within which changes ought to suffice to obtain a program (instruction sequence) which meets the given requirements. The relation between structured programming and "debugging" (understood as removal of faults) may turn out to be quite strong after all, a perspective which merits further work.

I assume that in principle the definitions of notions given for the special case of instruction sequences can be carried over to high level program notations. Doing so may yet be non-trivial in the case of a conventional program notation and it remains to be seen to what extent the results of this paper admit generalisation to practical context.

I do not know whether or not the notion of a BB SRT fault provides a satisfactory interpretation of the intuitive notion of a dormant fault and whether or not the notion of a WB SRT fault provides a satisfactory interpretation of the notion of a detected fault. For both questions the choice of a methodology for making progress on the respective issue constitutes a challenge.

7.1 Options for Further Work

It is non-trivial to describe a rationale of the use of the concept of program fault in software development. In the absence of a design it is hard to understand how a human programmer can find changes which resolve failures and are compliant with extensive regression test suites. It is plausible to use spectrum based testing to find locations for candidate TMO DFD defects which are candidate SRT faults at the same time. Actually spotting a TMO

DFD defect, however, requires inspection of a design, and upon recognition of a TMO DFD defect the design provides subsequent heuristics or designing a suitable change. What makes these matters hard to grasp, however, is that on the one hand the design must be sufficiently detailed to allow determination of a TMO DFD defect for each fault which is to be detected and resolved, while on the other hand the distance between the design and its candidate implementation apparently is so large that an automatic check of compliance (which would render spectrum based techniques useless) cannot be done, or does not compete with the undeniably significant efforts required for spectrum based fault localisation.

Developing a theoretical account of software processes with a proper role of faults, defects, bugs, as well as (regression) testing and verification is still a challenge.

One may imagine that instruction sequences are designed and constructed against the background of a large database of past work. Then given an instruction sequence `X` one may extract from the database prior odds for the probability that an instruction sequence of a given length contains a fault of a given modality M. Upon obtaining the information that a failure has been detected the subjective probability of the presence of a fault of modality M will increase and likelihood ratio transfer mediated reasoning comes into play (see e.g. [5]). Legal reasoning in court about the presence of faults in programs may be grounded in part on statistical information about program construction. It is an intriguing challenge to work out the "laws" of legal reasoning about software faults from first principles.

References

[1] H. Mushtaq, Z. Al-Ars, K. Bertels. Survey of Fault Tolerance Techniques for Shared Memory Multicore/Multiprocessor Systems. 6th IEEE International Design and Test Workshop (IDT 2011), 12–17, (2011). doi:10.1109/IDT.2011.6123094.

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

[3] A. Avižienzis, J.C. Laprie, B. Randell, C. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE*

Transactions on Dependable and Secure Computing, 1 (1), 1–23, 2004. doi:10.1109/TDSC.2004.2.

[4] J.A. Bergstra. Quantitative Expressiveness of Instruction Sequence Classes for Computation on Single Bit Registers.. *Computer Science Journal of Moldova* 27 (2), 131-161, 2019. http://www.math.md/publications/csjm/issues/v27-n2/12969/.

[5] J.A. Bergstra. Adams Conditioning and Likelihood Ratio Transfer Mediated Inference. *Scientific Annals of Computer Science* 29 (1), 1–58, 2019. doi:10.7561/sacs.2019.1.1.

[6] J.A. Bergstra. Instruction Sequence Faults with Formal Change Justification. *Scientific Annals of Computer Science* 30 (2), 105–166, 2020. doi:10.7561/SACS.2020.2.105.

[7] J.A. Bergstra. Sumterms, summands, sumtuples, and sums and the meta-arithmetic of summation. *Scientific Annals of Computer Science* 30 (2), 167–203 (2020). doi:10.7561/SACS.2020.2.167.

[8] J.A. Bergstra, M.E. Loots. Program Algebra for Sequential Code. *Journal of Logic and Algebraic Programming* 51 (2), 125–156, 2002. doi:10.1016/s1567-8326(02)00018-8.

[9] J.A. Bergstra, C.A. Middelburg. Thread Algebra for Strategic Interleaving. *Formal Aspects of Computing* 19 (4), 445–474, 2007. doi:10.1007/s00165-007-0024-9.

[10] J.A. Bergstra, C.A. Middelburg. Thread Extraction for Polyadic Instruction Sequences. *Scientific Annals of Computer Science* 21 (2), 283–310, 2011.

[11] J.A. Bergstra, C.A. Middelburg. Thread Algebra for Poly-Threading. *Formal Aspects of Computing* 23 (4), 567–583, 2011. doi:10.1007/s00165-011-0178-3.

[12] J.A. Bergstra, C.A. Middelburg. Instruction Sequence Processing Operators. *Acta Informatica* 49 (3), 139–172, 2012. doi:10.1007/s00236-012-0154-2.

[13] J.A. Bergstra, C.A. Middelburg. On Algorithmic Equivalence of Instruction Sequences for Computing Bit String Functions. *Fundamenta Informaticae* 138 (4), 411–434, 2014. doi:10.3233/FI-2015-1219.

[14] J.A. Bergstra, C.A. Middelburg. On Instruction Sets for Boolean Registers in Program Algebra. *Scientific Annals of Computer Science* 26 (1), 1–26, 2016. doi:10.7561/sacs.2016.1.1.

[15] J.A. Bergstra, C.A. Middelburg. Instruction Sequence Size Complexity of Parity. *Fundamenta Informaticae* 149 (3), 411–434, 2014. doi:10.3233/FI-2016-1450.

[16] J.A. Bergstra, C.A. Middelburg. A Short Introduction to Program Algebra with Instructions for Boolean Registers. *Computer Science Journal of Moldova* 26 (3), 199–232, 2019. http://www.math.md/files/csjm/v26-n3/v26-n3-(pp199-232).pdf

[17] N. Diallo, W. Ghardallou, A. Mili. Relative Correctness: A Bridge Between Testing and Proving. *10th Workshop on Verification and Evaluation of Computer and Communication System (VECoS 2016)*, 141–156, 2016. CEUR-WS:Vol-1689/paper11.

[18] E.W. Dijkstra. Notes on Structure Programming. Technological University Eindhoven, Department of Mathematics and Computing Science. Vol. 70-WSK-03, 1970. TUE-TR:EWD249.

[19] R.L. Glass. Persistent Software Errors. *IEEE Transactions on Software Engineering* 7 (2), 162–168, 1981. doi:10.1109/tse.1981.230831.

[20] J.C. Laprie. Dependable Computing and Fault Tolerance: Concepts and Terminology. In *Twenty-Fifth International Symposium on Fault-Tolerant Computing, Highlights from Twenty-Five Years*, 2–11, 1995. doi:10.1109/FTCSH.1995.532603.

[21] J. Laski. Programming Faults and Errors: Towards a Theory of Software Incorrectness. *Annals of Software Engineering* 4, 79–114, 1997. doi:10.1023/A:1018966827888.

[22] C.A. Middelburg. Searching Publications on Software Testing. 2010. arxiv:1008.2647v1.

[23] A. Mili, M.F. Frias, A. Jaoua. On Faults and Faulty Programs. In: P. Höfner, P. Jipsen, W. Kahl, M.E. Müller (Eds.), *Relational and Algebraic Methods in Computer Science (RAMICS 2014)*, *Lecture Notes in Computer Science* 8428, 191–207, 2014. doi:10.1007/978-3-319-06251-8_12.

[24] A. Ponse. Program Algebra with Unit Instruction Operators. *Journal of Logic and Algebraic Programming* 51 (2), 157–174, 2002. doi:10.1016/S1567-8326(02)00019-X.

[25] W. E. Wong, R. Gao, Y. Li, R. Abreu, F. Wotawa. A Survey on Software Fault Localization. *IEEE Transactions on Software Engineering* 42 (8), 707–740, 2016. doi:10.1109/tse.2016.2521368.

[26] L. Xiaojian, J. Ting, D. Xiaofeng. Formal Definition of Program Faults and Hierarchy of Program Fault-Tolerant Abilities. *4th International Conference on Information Science and Control Engineering (ICISCE)*, 2017. doi:10.1109/ICISCE.2017.78.

[27] X. Xie, T.Y. Chen, F.C. Kuo, B. Xu. A Theoretical Analysis of the Risk Evaluation Formulas for Spectrum-Based Fault Localization. *ACM Transactions on Software Engineering and Methodology* 22 (4), article no. 31, 2013. doi:10.1145/2522920.2522924.

[28] A. Zakari, S.P. Lee, C.Y. Chong. Simultaneous Localization of Software Faults Based on Complex Network Theory. *IEEE Access* 6, 23990–24002, 2018. doi:10.1109/access.2018.2829541.