Scientific Annals of Computer Science vol. 30 (2), 2020, pp. 105–166
doi: 10.7561/SACS.2020.2.105

Instruction Sequence Faults with Formal Change Justification

Jan A. Bergstra[1]

Abstract

The notion of an instruction sequence fault is considered as a theoretical concept, for which the justification of the qualification of a fragment as faulty is mathematical instead of pragmatic, the latter approach being much more common. Starting from so-called Laski faults a range of patterns of faults and changes thereof for instruction sequences is developed.
Keywords: instruction sequence, program fault, fault pattern, Laski fault, MFJ fault, change justification, software process flaw

1 Introduction

This paper aims at a description of the notion of a program fault in the context of instruction sequences. As it turns out formalizing the notion of a fault can be done in many ways and in this paper some essential definitions of fault are collected. These definitions are basic from a theoretical point of view and are not claimed to be of any immediate significance for software engineering. The faults as described in the paper each share the feature of formal justification of change. This idea requires some explanation: a

[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.

fault is understood as a fragment of an instruction sequence, which is held responsible for a deviation of the behaviour of the instruction sequence from its given functional specifications. Essential for a fault is that it admits a local improvement, which consists of replacing the fragment by another fragment, called the change of the fault, with the intended effect that a better if not perfect implementation of the specifications is obtained. Faults and changes for the better go hand in hand, and this circularity is quite hard to capture in convincing terminology. Program verification as a concept does not depend on any notion of fault, change, or improvement. Therefore program correctness is conceptually simpler, though often quite hard to establish, than the absence of faults. Whether or not absence of faults guarantees program correctness depends on the notion of fault that is being used. Changes go hand in hand with justifications for change.

A candidate fault is a fragment for which the suspicion has arisen that it may be a fault. Now the claim that replacement of a candidate fault by a candidate change eliminates the fault and thereby provides a better implementation of the specifications requires justification and I will focus on the case where such justification is obtained by comparison of the semantics of the original instruction sequence and its modified version. I will refer to the latter techniques of comparison as formal, not so much in the sense of involving formalization but rather in the sense of being mathematical and rigorous, and being independent of informal considerations, however important such considerations may be for the practice of software engineering.

A candidate change of a candidate fault is a program fragment which may be considered as a replacement of the fault and for which justification for the claim that it is sufficiently successful, or is so with a sufficiently high probability, must yet be obtained. I will simply speak of the justification of a change of a fault. Two lines of thought and practice appear about finding the justification for a change: (i) the judgement that a candidate change is sufficiently successful, i.e. is a change, is made on semantic grounds, the justification of which in turn may depend on verification by proof or on model checking, and (ii) the same judgement is made on textual grounds, or syntactic grounds if one prefers, including various forms of expert judgement, and heuristic methods involving the mining of program fragments.

In this paper I will discuss faults in the context of semantic justification. Doing so is primarily a theoretical exercise because most practical work on program faults is based on textual justification. In fact there might well be a gap between theoretical work on program faults and practice just as the

gap which [21] claimed to exist between theoretical work on testing and the industrial practice of testing.

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).*

The origin of this terminology, can be found in part in Laprie [36], while it appeared in a more definitive form in Avižienzis, Laprie & Randell [3] and in Avižienzis, Laprie, Randell, & Landwehr [4]. In Laprie [36], however, the relation between fault and error is taken to be quite generic, so that a programming mistake may be considered a fault, and the resulting (wrong) program fragment may be considered an error. In [3] errors are dynamic conditions just like failures, programmer mistakes are not considered proper faults but are merely considered to be causes of faults. In [34], however an error plays the role of a mistake, causing a fault, rather than having a faults as its cause. Rather than adopting the definition of [3] as the definition of fault we will consider the same definition as the introduction of a specific fault pattern deserving a name of its own. The notion of a fault pattern is informal, we were led to its use by the wish to accommodate a variety of fault definitions from the literature within a single framework.

Definition 1.2 (Fault Pattern) *A fault pattern (for programs, software, instruction sequences) is a class of faults or candidate faults (suspected*

faults, localized faults), which is described in terms of some or all of the following aspects: source language (program notation, instruction sequence notation), specification language and specification conventions, syntax of faults, fragmentation of faults (faults as consecutive fragments versus multi-hunk faults consisting of multiple fragments), syntax of changes, justification of changes, multiplicity of faults (multiple faults versus multi-faults).

Definition 1.3 (ALR Fault) *An ALR (for Avižienzis, Laprie & Randell) fault is a program fragment which is the cause of an error which in turn is the cause of a failure.*

For an ALR fault in a program `X` it does not matter whether or not its being a fault has actually been detected either empirically via tests of `X` and verification of modified versions of `X` where the fault has been resolved, or by other means. Causation must be understood in a counterfactual manner: if the fault would be performed during a run of the program under some circumstance it would be the case that the program fragment which is considered at fault is considered a major cause of a subsequent failure. The contrast between effective and dormant failures has been made early on, but the matter is rather informal. A similar distinction is made between temporary and persistent faults.

Definition 1.4 *An ALR fault is effective if a failure which it has caused has been observed, otherwise an ALR fault is dormant.*

Definition 1.5 *An ALR fault is temporary in a program if it becomes found and eliminated after some time, and otherwise it is persistent. (This terminology originates from [28])*

Importantly these notions are informal, indeed the notions of program, specification, execution, failure, fault, and causation each have a wide variety of possible interpretations. When formalizing these notions many different formalizations may result. As it turns out even if program, specification, and failure are chosen to have very definite meanings, as respectively instruction sequences, relations on a finite domain, and computations leading to wrong results, there is still much room for variation in the notions of causation and as a consequence there is much room for variation in the notion of an ALR-fault.

Below these ideas will be adopted for instruction sequences in the place of programs. Various difficulties arise when contemplating this terminology

in more detail. First of all the matter can be simplified by not dealing with errors, and viewing faults as causes of failures straightaway. Secondly the notion of causation requires an explanation, and providing that explanation is quite difficult.

In the context of faults some other terminology is useful, though less agreement seems to exist about it in the literature. Again it is assumed that specifications for the behaviour of a program are given. Each of these notions is dependant on, i.e. varies with, the precise meaning of fault one wishes to adopt.

All views on faults have in common that fragments of programs are considered amenable to being at fault. This view brings with it the following terminology.

Definition 1.6

Candidate fault. *A candidate fault is a fragment f of an instruction sequence which is under investigation for being faulty.*

Fault determination. *Fault determination is the process of clarifying whether or not a candidate fault is a fault.*

Fault suspicion. *A fragment f in* `X` *may be suspected, i.e. according to some criteria it is likely to be faulty. A suspected fault is assigned a level of suspicion.*

Fault localization. *Fault localization is the process of finding fragments f in a program which are suspected faults with level of suspicion l or higher.*

Fault prediction. *Fault prediction is the process of finding fragments f in a program which are likely to be suspected with level of suspicion l or higher. (In practice fault prediction and fault localization coincide for most authors).*

Fault change. *Fault change is the process of generating a fragment g which can replace a fault f in an instruction sequence* `X` *thereby removing this particular fault. (A fault change is alternatively called a fix or a patch. I prefer to use change because both fix*

and patch a connotation of lack of rigour which is not intended.)

Fault elmination. *Successful replacement of a fault with the effect that the new fragment is not at fault anymore is also called fault elimination.*

1.1 Summary of the Paper

With simple examples the framework consisting of a specification, an instruction sequence, with a (candidate) fault in it, and the plurality of its causes, each taking the form of a change, is put forward in Paragraph 1.2.1. Next, in Paragraph 1.3 some introductory remarks are made about the instruction sequence notation ISNwp[br], which combines so-called structured programming primitives and basic actions acting on single bit registers. The systematic description of fault patterns begins in Section 2 with faults for which changes are justified by way of testing.

The first definition of faults with formal change justification (our terminology) based on semantic principles can be found in Laski [37]. Laski's proposals sharpen the idea of [44] where the faults of a program which incurs failures are implicitly defined in an indirect manner. The idea is that upon introducing a change a provably correct implementation of the specifications is found.

As with ALR faults I will not adopt Laski's definition as being authoritative on software faults but I will introduce a named pattern of faults according to Laski's definition, moreover I will rephrase Laski's definition of faults in terms of semantics rather than in terms of verification. Such faults will be called called Laski faults and various fault patterns are proposed: n-Laski faults, weak n-Laski faults, multiple Laski faults, Laski multi-faults, and Laski multi-hunk faults.

A more liberal definition of faults with formal change justification has been prosed by Mili, Frias and Jaoua in [39]. I will speak of MFJ faults, and in addition several variations of MFJ faults are introduced.

After establishing some elementary results about Laski faults and MFJ faults, the paper proceeds with a description of multiple faults, mult-faults, and multi-hunk faults for the MFJ case. Members of a multi-fault need not qualify as either Laski faults or MFJ faults or as one of the variations thereof. In Paragraph 3.7 this observation leads to MFJ* faults and in Paragraph 3.8 to various forms of so-called essential faults. In Section 4 the notion of an

essential fault is considered in the context of change justification via testing. Further some remarks are made on faults in connection with spectrum based fault localization (SBFL). Most software faults are diagnosed by informal methods with justifications of change relying on software expert activity. In Section 5 under the name of "algorithm conformance oriented justification of change" further fault patterns are discussed which allow for informal justification of changes, moreover some combinations of these fault patterns with fault patterns based on formal justification of change are proposed. In the concluding remarks some attention is paid to specification faults. Although the idea of a (software) specification fault is fairly evident, there is almost no literature about such faults, and no definition of specification faults seems to exist. Beyond specification faults one might contemplate requirements faults, an idea which I consider less informative. Instead I suggest that beyond specification faults lie so-called software process flaws. Finally software process flaws are discussed in some detail, in connection with the Boeing 737 Max MCAS affair.

1.2 Examples of Faults and Corresponding Change(s)

I will work with instruction sequences which take inputs and outputs in bit valued registers. A detailed description of an instruction set and corresponding primitives for instruction sequences can be found in [18, 6] and for background material I mention [11, 20, 8, 16], and for the case of so-called polyadic instruction sequences, i.e. packages of instruction sequences, I mention [14] and [15]. I will not repeat these matters here, except for providing an ad hoc and casual explanation of instructions ad instruction sequences which are used in the text.

1.2.1 An Example of a Fault and Its Change(s)

Suppose X works on a single register `inout` which may contain values 0 or 1. We use the following total correctness specification P for an instruction sequence `X`: P requires that `X` computes the identity function $\{(0,0),(1,1)\}$, in other words it leaves the content of the single bit register unaffected.

Now consider the candidate implementation $\mathtt{X} \equiv \mathtt{inout.i/c};!$. Here `inout.i/c` is the basic action which applies the method `i/c` to the register in focus `inout`. Performing `i/c` to the register with content `b` works as follows: (i) the content is replaced by `c(b)` which stands for complementing `b` (`c` is the effect function), and the reply is determined as `i(b)` where `i` is the

identity function (i is the reply function). ! is the termination instruction. The effect of running X is that the value of inout is flipped. Considered as an implementation of P, X fails on both arguments.

Now consider $\mathtt{X}' \equiv \mathtt{inout.i/0;!}$. The method i/0 has identity i as the reply function and the function 0 (constant zero) as the effect function. The candidate implementation $\mathtt{X}'$ is better thatn X because it computes the right value on input 0. The idea that $\mathtt{X}'$ is a better implementation of P than X stems from [39] where that idea has been worked out in great detail. According to [39] the fact that the fragment $f \equiv \mathtt{inout.i/c}$ of X can be replaced in such a manner as to obtain a better implementation of P is precisely what is needed in order to substantiate the claim that the fragment f (i.e. the single instruction inout.i/c) constitutes a fault in X.

Then consider $\mathtt{X}'' \equiv \mathtt{inout.i/i;!}$ (here the effect function is identity and application of the method will not change the contents of the register), and note that this instruction sequence is once more better as an implementation of P than $\mathtt{X}'$ is. Its existence demonstrates that the first instruction of $\mathtt{X}'$ is faulty w.r.t. the specification P. Finally consider $\mathtt{X}''' \equiv \mathtt{!}$ which, like $\mathtt{X}''$ also correctly implements P and is both shorter and faster, though not better in terms of functionality. The transition from $\mathtt{X}''$ to $\mathtt{X}'''$ is an optimization and is not understood as the change of a fault in $\mathtt{X}''$ though the mere possibility of this optimization signals the presence of a certain shortcoming of $\mathtt{X}''$ when understood as an implementation of P.

1.2.2 A Plurality of Faults (as Causes) for a Single Failure

I will now assume that there is a console (that is a service representing a console) named C on which a sequence of characters can be written as subsequent outputs by means of an instruction C.p(u) which "prints" a character u. For each character u the basic action C.p(u) returns Boolean result 1 signalling a positive outcome.

Now we have the following specification for P: "X prints a single 0 and then terminates". The instruction sequence $\mathtt{X} \equiv \mathtt{C.p(0);!}$ does the job. Now consider the instruction sequence

$$\mathtt{Y} \equiv \mathtt{\#1;C.p(1);!;C.p(0);!}$$

Here #1 represents a forward jump of size 1 which is in fact a mere skip. Clearly upon execution Y first prints 1 and then terminates, which when considered from the perspective of the specification is a failure.

The following modifications involving a change of a single instruction only turn $\mathtt{Y}$ into a correct implementation $\mathtt{Y_i}$ of the specification P:

$\mathtt{Y_1} \equiv \#\mathtt{3}; \mathtt{C.p}(\mathtt{1}); !; \mathtt{C.p}(\mathtt{0}); !,$

$\mathtt{Y_2} \equiv -\mathtt{C.p}(\mathtt{0}); \mathtt{C.p}(\mathtt{1}); !; \mathtt{C.p}(\mathtt{0}); !.$

$\mathtt{Y_3} \equiv \#\mathtt{1}; \#\mathtt{2}; !; \mathtt{C.p}(\mathtt{0}); !.$

$\mathtt{Y_4} \equiv \#\mathtt{1}; \mathtt{C.p}(\mathtt{0}); !; \mathtt{C.p}(\mathtt{0}); !,$

$\mathtt{Y_5} \equiv \#\mathtt{1}; +\mathtt{C.p}(\mathtt{0}); !; \mathtt{C.p}(\mathtt{0}); !.$

Some explanation of the notation may be helpful: $\#\mathtt{2}$ represents a forward jump of size $\mathtt{2}$ and $\#\mathtt{3}$ represents a forward jump of size $\mathtt{3}$. In $\mathtt{Y_2}$ the test instruction $-\mathtt{C.p}(\mathtt{0})$ works as follows: first print $\mathtt{0}$ and receive reply $\mathtt{true}$, and then in view of the $-$ sign in front of the test, skip the next instruction to proceed with !. In $\mathtt{Y_5}$ the (i.e. termination) is performed.

It follows that upon using the given Definition 1.3 of a fault, there are at least two faults in Y, the first instruction with 2 different changes (as in $\mathtt{Y_1}$ and in $\mathtt{Y_2}$ respectively), and the second instruction with 3 different changes (as in $\mathtt{Y_3}, \mathtt{Y_4}, \mathtt{Y_5}$). I notice that the repair in $\mathtt{Y_4}$ may be considered a local fix in the terminology of [23]. Apparently the notion of a cause as meant in Definition 1.3 refers to non-exclusive causes.

1.2.3 Arrangements for an Example with Faults and Changes: Context Parameters

The above example of a fault and its various changes indicates that a significant number of parameters must be determined for the example to be comprehensible:

(i) instruction sequence notation: which instruction sequences are considered (here PGLA from [11]; before and after being changed an instruction sequence must be compliant with the chosen instruction sequence notation)

(ii) which services come into play (in the example in Paragraph 1.2.1 a single bit register, and in Paragraph 1.2.2 the console $\mathtt{C}$ which admits the writing of bits),

(iii) constraints on faults: which instructions or instruction sequences may be considered faults (here single instructions are considered as candidate faults),

(iv) constraints on changes: which instructions or instruction sequences may play the role of changes (here single instructions may serve a

changes for faults consisting of single instructions, without any additional constraints).

Only if each of these context parameters have been determined it is possible to prove anything about faults and changes of faults. Unsurprisingly the facts that can be shown depend significantly on such choices. For instance in the case of PGLA, or PGLB (like PGLA but also allowing backward jumps), it is plausible that if an instruction u is considered faulty, and replacement by say $\mathtt{u}';\mathtt{w}$ is contemplated (with $\mathtt{u}'$ an instruction, and $\mathtt{w}$ a nonempty instruction sequence) as a change of that fault, jumps which jump over u must be increased by the number of instructions of $\mathtt{w}$. This observation renders it doubtful that for these instruction sequence notations single instruction faults can be plausibly eliminated by mere replacement of a single instruction by its change. In the absence of structured program instruction, and making use of forward and backward jumps so that instruction numbers are vital (a sensitivity for detail which can be overcome by using goto's and labels or by using structured programming instructions), the relevance of so-called multi-hunk faults (see [47]) can be noticed immediately. I will provide an example of this complication where $\mathtt{X}$ can be turned into an implementation of specification P by making two changes rather than one. In the terminology of Definition 3.7 below $\mathtt{X}$ features a multi-hunk fault when considered as as a candidate implementation of P.

The specification Q asserts that the following function $f(-,-)$ is computed on inputs $\mathtt{in{:}1}$ and $\mathtt{in{:}2}$, with outputs written on $\mathtt{C}$. $f(0,0) = 0, f(0,1) = 11, f(1,0) = f(1,1) = 010$. The following instruction sequence $\mathtt{X}$ in the instruction sequence notation PGLA is an implementation of Q:

$$\begin{aligned}\mathtt{X} = {}&+\mathtt{in{:}1.i/i};\#8;+\mathtt{in{:}2.i/i};\#3;\mathtt{C.p}(0);!;\\ &\mathtt{C.p}(1);\mathtt{C.p}(1);!;\mathtt{Cp}(0);\mathtt{C.p}(1).\mathtt{C.p}(0);!.\end{aligned}$$

The specification P deviates from Q as follows: it requires that $g(-,-)$ is computed with $g(0,0) = 00, g(0,1) = 11, g(1,0) = g(1,1) = 010$. Now $\mathtt{X}$ may be considered a faulty implementation of P with fault $f \equiv \#3$ and change $g \equiv \#4;\mathtt{C.p}(0)$ for it. But that does not quite work as in addition #8 must be changed into #9. Indeed, the following instruction sequence $\mathtt{Y}$ implements P.

$$\begin{aligned}\mathtt{Y} \equiv {}&+\mathtt{in{:}1.i/i};\#9;+\mathtt{in{:}2.i/i};\#4;\mathtt{C.p}(0);\mathtt{C.p}(0);!;\\ &C.p(1);C.p(1);!;Cp(0);C.p(1).C.p(0);!.\end{aligned}$$

Below I will focus on instruction sequences written in terms of the structured programming instructions, where, in the absence of jumps expressed

in terms of instruction counters no such obstacle exists against the use of changes which increase or decrease the number of instructions of an instruction sequence.

1.3 While Programs as Instruction Sequences with Structured Programming Instructions

The instruction sequence notation ISNwp, allows to write while programs, which, however, are written as instruction sequences with the help of some additional instructions, so-called structured programming instructions. These instructions can be translated back to the notation without such additional instructions, a transformation which is referred to as projection semantics.

- Conditional construct: $+\mathtt{a}\{;\mathtt{X};\}\{;\mathtt{Y};\}$ works as follows: perform $\mathtt{a}$, if `true` is returned then do $\mathtt{X}$, if `false` is returned then do $\mathtt{Y}$.

  And $-\mathtt{a}\{;\mathtt{X};\}\{;\mathtt{Y};\}$ works as follows: perform $\mathtt{a}$, if `false` is returned then do $\mathtt{X}$, if `true` is returned then do $\mathtt{Y}$.

  In more detail: $+\mathtt{a}\{;\mathtt{X};\}\{;\mathtt{Y};\}$ starts with method call $\mathtt{a}$ and proceeds with $\mathtt{X}$ (i.e. the next instruction) on a negative reply and with $\mathtt{Y}$, i.e. the first instruction following the next occurrence of $\}\{$ (or the corresponding occurrence of $\}\{$ or of $\}$ in the case of nested conditionals) upon a positive reply. When performed the instruction $\}\{$ works just as a jump to the first instruction after the next occurrence of $\}$ (at least in the absence of nesting). When performed $\}$ works like a skip (i.e. $\#1$).

- The one-armed conditionals $+\mathtt{a}\{;\mathtt{X};\}$ and $-\mathtt{a}\{;\mathtt{X};\}$ work as the two-armed construct assuming that $\mathtt{Y}$ works as $\#1$ (i.e. a skip).

- The while loop: $+\mathtt{a}\{\star;\mathtt{X};\star\}$ works as follows:

  REPEAT: perform $\mathtt{a}$, if `true` is returned, then perform $\mathtt{X}$, and upon termination of $\mathtt{X}$ jump back to REPEAT.

  Assuming $\mathtt{X} = \mathtt{u}_1;\ldots;\mathtt{u}_n$ this behaviour is also expressed by:
  $$-\mathtt{a};\#\mathtt{n}+2;\mathtt{u}_1;\ldots;\mathtt{u}_n;\backslash\#\mathtt{n}+2.$$

- The while loop $-\mathtt{a}\{\star;\mathtt{X};\star\}$ works as follows:

  REPEAT: perform $\mathtt{a}$, if `false` is returned, then perform $\mathtt{X}$ and upon termination of $\mathtt{X}$ jump back to REPEAT.

- termination ! and divergence #0 are allowed on any position, in particular within a loop.
- Structured programming instructions may be combined with jumps and goto's, for instance as follows:
$$+\mathtt{a}\{;-\mathtt{b};\#\mathtt{6};\mathtt{c};!;\}\{;\mathtt{b};+\mathtt{d};\};\mathtt{c};!.$$
When the forward jump #6 is performed, then a jump is made to the control position just after the instruction +d, so that the remaining instructions are c; !.
- It is assumed that instruction sequences are statically correct in the sense that for each structured instruction complementary instructions are present. This is a practical matter, not a matter of principle. In [11] the semantics of instruction sequences involving structured programming instructions is given without this constraint as well.

The semantics of ISNwp instruction sequences can be given by means of translation to the simpler notation PGLA, a semantic method called projection semantics. For instance $+\mathtt{a}\{;-\mathtt{b};\mathtt{c};!;\}\{;\mathtt{b};+\mathtt{d};\};\mathtt{c};!$ translates into: $-\mathtt{a};\#\mathtt{5};-\mathtt{b};\mathtt{c};!;\#\mathtt{3};\mathtt{b};+\mathtt{d};\mathtt{c};!$. For details on these matters I refer to [11] and [14].

1.3.1 Services for ISNwp[br]

For a complete description of an instruction sequence notation also a description of the admissible services is required. Below only single bit services (alternatively called Boolean registers) will be used with 1 denoting true and 0 denoting false, for input, for output and for auxiliary data. Instructions for services use focus method notation, in combination with a specific notation for methods on Boolean registers. I refer to [6] and [17, 18] for an introdution to these notations. Together this leads to an instruction sequence notation which will be denoted with ISNwp[br] below. Some additional constraints and conventions apply to ISNwp[br]:

- Output registers have the form out0:n and out1:n with n a decimal digit sequence. The bit 0 in out0 indicates that the initial value of the output register is chosen 0 whereas say out1:17 has initial value 1.

  The name of an output register is an instance of so-called focus, it gives access to the named register, also called a service. On output

registers only methods which write a value are allowed, in the notation of [6] (and [17]) these methods are `0/0` (reply value is `0` new content is `0`), `0/1` (reply value is `0` new content is `1`), `1/0` (reply value is `1` new content is `0`), `1/1` (reply value is `1` new content is `1`).

- Input registers have the form `in`:n and `out1`:n with n a decimal digit sequence. The name of an input register is also a focus, it gives access to the named (input) register. On output registers only methods which read (or in this case test) a value are allowed, in the notation of [6] these methods are `i/i` (reply value is `1` new content is `0`), return `i`(`b`) at content `b`, that is: `true` at content `1` and return `false` at content `1`. `c/i` (reply value is `1` new content is `1`). return `c`(`b`) at content `b`, that is: return `false` at content `1` and return `false` at content `1`.

  For both methods thee effect function is `i` for identity, i.e. the content of the register is left unchanged by application of the method.

- Auxiliary registers are `aux0`:n and `aux1`:n where (as with output registers) the initial value is part of the name. Auxiliary registers are operated on with instructions from the instruction set as specified in [6], or in [17]. These instructions are method calls of the form `f`.α/β with `f` a focus for an auxiliary register (for instance `aux1:23`) and $\alpha, \beta \in \{\mathtt{0}, \mathtt{1}, \mathtt{i}, \mathtt{c}\}$. Here α is the function which determines the reply from the content and β is the function which determines the next content from the contant at the moment of the method call. For Booleans I will identify `true` and `1`, resp. `false` and `1`. With this convention: `0` represents the function constant `0`, represents the function constant `1`, `i` represents the identity function, and `c` swaps (complements) `0` and `1`.

- In a conditional only input registers and auxiliary registers can be tested, thus +`in:7.i/i{` is an permitted instruction while −`0ut0:5.1/0{` is not.

- A repetition only auxiliary registers can be tested, (because inputs cannot change and outputs cannot be tested), thus −`aux0:15.i/c{`$\star$ is ok while +`in:23.i/i{`$\star$ is not.

- There may or may not be an additional Turing tape service with the instruction set taken from [19].

- Following the conventions introduced in [6] $\mathtt{LLOC}(\mathtt{X})$ (for logical lines of code) denotes the number of instructions of an instruction sequence $\mathtt{X}$.

1.3.2 Fault Constraints and Change Constraints

We will mainly look that the case that a (candidate) fault is a single instruction, but in principle a fault in an instruction sequences may be each subsequence of it. Faults are also instruction sequence, but faults need not comply with all criteria of the instruction sequence notation which is being used.

A change g for a fault f of an instruction sequence $\mathtt{X} = \mathtt{Y};\mathtt{f};\mathtt{Z}$ in an instruction sequence notation ISN-$\mathbb{L}$ is an instruction sequence such that the result of replacing f by g in $\mathtt{X}$, $\mathtt{X};\mathtt{g};\mathtt{Z}$ is also an instruction sequence in ISN-$\mathbb{L}$. change constraints tell which instruction sequences may be used as (candidate) changes of faults. Unless stated otherwise changes for faults of ISNwp[br] instruction sequences are supposed to be restricted by tre following constraints:

The change of a an action consists in a modification of its parameters, while the type of the action (in, out, aux) remains the same. In detail:

- if f of the form $+\mathtt{a}\{$ or of the form $-\mathtt{a}\{$ is considered a fault then its replacement g must be of the form $+\mathtt{b}\{$ or $-\mathtt{b}\{$ (sign and basic action may change),

- if f of the form $+\mathtt{a}\{\star$ or of the form $-\mathtt{a}\{\star$ is considered a fault then its replacement g must be of the form $+\mathtt{b}\{\star$ or $-\mathtt{b}\{\star$ (sign and basic action may change),

- $\}$ and $\}\{$ may not be changed.

- for basic instructions which are not conditionals termination ! as well as divergence #0 may serve as a change.

In other words structured programming primitives are turned into structured programming primitives of the same kind. All other instructions are assignments to output registers, or to auxiliary registers, or instructions for termination. The latter instructions, when considered single instruction faults (or candidate faults in more precise language) may be replaced by a change consisting of an arbitrary well-formed instruction sequence.

1.3.3 Specifications

As specifications (named P) I will only consider relations of input tuples and output tuples, which provide at least one output per input. Because there are only finitely many inputs each specification can be implemented by way of a finite instruction sequence without the use of auxiliary registers (see e.g. [6] for a proof of this fact).

2 Faults with Changes Justified by Means of Testing

An obvious idea is to use testing for the identification of faults. In spite of the immediate nature of such ideas it is hard to find proper definitions of such fault patterns in the literature.

2.1 Faults with Single Test Justification of Change

The most straightforward explanation for a fragment f being the cause of a failure is that a failure which appears on a single test can be avoided by a local change of the fragment. In most definitions I will (implicitly) make use of an instruction sequence notation ISN-$\mathbb{L}$ which itself may take many different forms, including ISNwp[br] and PGLA[br] (see [11]), PGLB[br] and related notations.

Definition 2.1 (STJoC Fault) *Given an instruction sequence notation ISN-$\mathbb{L}$, f is an STJoC (single test justification of change) fault at position p in instruction sequence $\mathtt{X} \in$ ISN-$\mathbb{L}$ w.r.t. a specification P if the following three conditions are met:*

(i) $\mathtt{X}$ *does not meet specification P, in particular it fails at a test progression α with inputs $in(\alpha)$ (that is $P(in(\alpha), out(\alpha))$ fails to hold),*

(ii) f is a subsequence of $\mathtt{X}$ *beginning at position (instruction number) p, and*

(iii) there is a (new) program fragment g such that after replacement of f by g, the resulting instruction sequence $\mathtt{X}_{g/f}$ is a correct instruction sequence in ISN-$\mathbb{L}$ which complies with P on a test β with the same inputs as α inputs as β i.e. $in(\alpha) = in(\beta)$.

Moreover, in this situation g is called an STJ (single test justified) change of the fault f, and α is a witness for the fault.

2.2 Leaving ISN-$\mathbb{L}$ Implicit, Leaving Fault Position Implicit etc.

In the sequel of this paper most definitions have an instruction sequence notation ISN-$\mathbb{L}$ as a parameter. Rather than to mention a name for the instruction sequence notation at hand, I will leave the presence of such a notation and a name for it implicit.

When writing about faults various shortcuts are useful, to mention: (i) speaking of a fault f in $\mathtt{X}$ without mentioning its position p, under the assumption that it is left implicit, if however, merely the text of f is meant it may be called the subject of the fault, (ii) speaking of a fault (f, g) instead of speaking of a pair of a fault and a change, (iii) speaking of a fault instead of a candidate fault, (iv) speaking of a change of fault f instead of speaking of its proposed change or candidate change. It appears to be rather difficult to achieve full precision on such matters while preserving readability.

2.3 Single Test Justification of Change is no Guarantee of Improvement

An obvious problem with single test justification of change is that a change may improve the outcome of a given test and at the same time introduce another fault. Let P be the specification which requires that input register $\mathtt{in{:}1}$ is copied into output register $\mathtt{out0{:}1}$. As a potential implementation consider $\mathtt{X} \equiv \mathtt{out0{:}1.1/1};!$. Testing $\mathtt{X}$ on input $\mathtt{in{:}1} = \mathtt{0}$ a failure is observed. Taking $f \equiv \mathtt{out0{:}1.1/1}$ as a fault and $g \equiv \mathtt{out0{:}1.0/0}$ as the corresponding change solves the problem, because $\mathtt{X}_{g/f}$ correctly processes the same test. Now, however, testing on input $\mathtt{in{:}1} = \mathtt{1}$ will reveal a failure. In spite of the single test justification application of the change has brought no improvement.

Below changes with semantic justification that go beyond the application of a single test or of a few tests will be considered, as a way to avoid the kind of complication, just mentioned, i.e. merely trading one fault for another one.

2.4 Metric Single Test Justification of Change

The notion of an STJoC fault allows various weaker alternatives, for instance one may assume that together with a specification P a metric d on the space of outputs is given and that an outcome of a computation is considered to be better if it is closer to a correct output. Assuming that P specifies the graph of a total function from inputs to outputs the following alternative arises:

Definition 2.2 (Metric STJoC Fault) *f is a metric STJoC fault at position p in $\mathtt{X}$ w.r.t. a specification P if the following three conditions are met:*

(i) *$\mathtt{X}$ does not meet specification P, in particular it fails at a test progression α with inputs $in(\alpha)$ (that is $\neg P(in(\alpha), out(\alpha))$ holds and writing $out_P(v)$ for the unique output v on input w: $out_P(in(\alpha)) \neq out(\alpha)$.*

(ii) *f is a subsequence of $\mathtt{X}$ beginning at position (instruction number) p, and*

(iii) *there is a (new) program fragment g such that after replacement of f by g, the resulting instruction sequence $\mathtt{X}_{g/f}$ is in better compliance with P on a test β with the same inputs as α inputs (i.e. $in(\alpha) = in(\beta)$) in the following sense:*

$$d(out_P(in(\alpha)), out(\beta)) < d(out_P(in(\alpha)), out(\alpha)).$$

Moreover, in this situation g is called a metric STB (metric single test justified) change of the fault f, and α is a witness for the fault.

STBR faults are instances of ALR faults, as a reasonably plausible, though perhaps debatable, notion of causation of a failure by the fragment f comes into play. Single test change justification is considered formal justification in spite of the fact that this definition pays no attention to the possibility that other failures are introduced by the replacement of f by g. The issue is that a formal definition, in this case a definition related to a single test is decisive for the judgement that a change is valid, irrespectively of any other considerations.

2.5 Regression Test Justification of Change

Stronger guarantees, than come with the STJoC fault pattern, that a change constitutes an improvement are obtained by looking at regression tests.

Given $\mathtt{X}$ and a specification P, and a witness progression α on which $\mathtt{X}$ fails, as well as a fragment f of P and its proposed replacement g, we assume

the presence of a collection (test suite) $\beta = (\beta_1, \ldots, \beta_n)$ of tests which have been successful for X.

Definition 2.3 (RTJoC Fault) *The tuple (f, α, β, g) is a regression test justified (witnessed and resolved) fault if: (i) (f, α, g) is a STJoC fault of* X *w.r.t. P, (ii) for all $i \in [1, n]$ starting with the same inputs as β_i, $\mathtt{X}_{g/f}$ terminates and produces a result which complies with P (though the output may differ from the output of β_i in case P is non-deterministic). In this case g is called an RTJ change for f w.r.t. P and w.r.t. said test regression test suite.*

The change g for an RTJoC fault f resembles the notion of a plausible patch from [46]. Working in ISNwp[br] almost every fault admits a proper change.

Theorem 2.1 *Assume that* X *has a witnessed defect w.r.t. specification P (with witness α producing an output which is not everywhere 0), and moreover assume that* X *has been confirmed (as a candidate implementation of P) by the test suite $\beta_1, \ldots, \beta_n$ (not containing α) then there is a single instruction f of* X*, and a change g for f so that (f, α, β, g) is a fault with a test suite justified change of* X*.*

3 Fault Patterns Involving Semantic Justification of Change

Semantic justification of change refers to justifications which take the semantics of the instruction sequence and its modified version into account, and which in principle, though not always in toy examples, do so to an extent that achieving the same certainty by way of testing will give rise to a combinatorial explosion of required test cases. In general semantic justification cannot be replaced by checking a single test case or by checking a few test cases. In general a combinatorial explosion of test cases is required if semantic assertions are to be validated by way of testing.

In this Section I will survey a collection of definitions of fault each of which are directly or indirectly (the case of essential faults) based on semantic justification of changes. How the semantic justification is obtained is left untouched, and this may range from verification and model checking to systematic testing.

Definition 3.1 *A witnessed defect for* X *w.r.t. P is a terminating progression α for* X *which produces a result which is not compliant with P.*

3.1 Laski Faults

The first approach to program faults which obtains a precise, semantics based definition of a program fault is due to Laski [37], where it is required of a fault (i.e. a program fragment which is considered faulty w.r.t. a specification) that it admits a provably correct replacement. Rather than adopting Laski's definition of a fault, as defining for program faults, I will consider a specific a version of Laski's definition as defining for a specific fault pattern. Assuming that proof systems are complete, correctness may replace provable correctness w.l.o.g.

Definition 3.2 (Laski Fault) *A fragment f of* `X` *is a Laski fault at position p in* `X` *w.r.t. P if (i)* `X` *does not meet specification P, (ii) f is a subsequence of* `X` *beginning at position (instruction number) p, and (iii) there is a (new) program fragment g such that after replacement of f by g, the resulting instruction sequence $\mathtt{X}_{g/f}$ complies with P. In these conditions g is called a Laski change of the fault f.*

In some cases it is important to refer to the fragment f only, when speaking about a fault. This motivates the following terminology:

Definition 3.3 (Fault Subject) *For a fault f in* `X` *with change g the program fragment f together with its location is also referred to as the subject of the fault.*

Definition 3.4 (Proper Laski Fault) *A Laski fault f in* `X` *is proper if* $\mathtt{LLOC}(f) < \mathtt{LLOC}(\mathtt{X})$.

I will omit the explicit treatment of positions, assuming that a fragment f of `X` always comes with a position in `X`. The notion of multiple Laski faults is straightforward, but it is given an explicit definition in order to avoid confusion with the related notions of Laski multi-faults, and multi-hunk Laski faults.

Definition 3.5 (Multiple Laski Faults) *An instruction sequence* `X` *contains multiple Laski faults $f_1, \ldots, f_n$ with Laski changes $g_1, \ldots, g_n$ respectively if each pair f_i, g_i is a Laski fault for* `X` *and the fault subjects are f_i pairwise non-overlapping.*

Definition 3.6 (Laski Multi-Fault) *A set of multiple non-overlapping Laski-faults $f_1, \ldots, f_n$ with changes $g_1, \ldots, g_n$ is a Laski-multifault if after*

simultaneous replacement of f_i by g_i for $1 \leq i \leq n$, the resulting instruction sequence $\mathtt{X}_{\tilde{g}/\tilde{f}}$ complies with P on all inputs. Here n is the width of the multi fault.

Multi-hunk faults stem from [47] and adapting the same idea to Laski faults is straightforward.

Definition 3.7 (Multi-Hunk Laski Fault) *A set of multiple non-overlapping Laski-faults $f_1, \ldots, f_n$ with changes $g_1, \ldots, g_n$ is a multi-hunk Laski fault if (i) after simultaneous replacement of f_i by g_i for $1 \leq i \leq n$, the resulting instruction sequence $\mathtt{X}_{\tilde{g}/\tilde{f}}$ complies with P on all inputs, and (ii) for no proper subset V of $\{1, \ldots, n\}$ changing only the f_i by g_i for $i \in V$ produces a correct implementation of P. Here n is the width of the multi fault.*

It is useful to consider an example. X uses is a single input register in:1, and two output registers out0:1 and out0:2, as well as a 0-initialized auxiliary register aux0:.

$\mathtt{X} \equiv \mathtt{out0{:}1.0/1}; +\mathtt{in{:}1.0/i}\{; \mathtt{Y}; \}; +\mathtt{in{:}1.0/i}\{; \mathtt{Z}; \}; !$ with
$\mathtt{Y} \equiv \mathtt{out0{:}2.1/1}; \mathtt{aux0{:}1.1/1}; \}$ and
$\mathtt{Z} \equiv \mathtt{out0{:}2.1/1}; +\mathtt{aux0{:}1.i/i}\{; \mathtt{out0{:}1.0/0}; \mathtt{out0{:}2.0/0}; \}$.

The specification P requires that for all values of both inputs both outputs are set to 1. Now it is easy to see that $f_1 \equiv +\mathtt{in{:}1.0/i}\{$ (first occurrence) is a Laski fault with change $g_1 \equiv +\mathtt{in{:}1.1/i}\{$ and that $f_2 \equiv +\mathtt{in{:}1.0/i}\{$ (second occurrence), is a Laski fault with change $g_2 \equiv +\mathtt{in{:}1.0/i}\{$. But upon simultaneously changing both Laski faults the result is that both output registers end up with value 0 which deviates further from the required behaviour (that is P) than leaving both Laski faults unchanged in which case out0:2 is set to 1.

It follows from this observation that within ISwp[br] a multiple Laski fault may at the same time not be a Laski multi-fault. Using the console C from Paragraph 1.2.2 and the instruction sequence notation PGLA is is easy to provide an example of a multiple Laski fault which is not a Laski multi-fault at the same time. Let P require that a single 0 is written, and consider $\mathtt{X} \equiv \#2; \mathtt{C.p}(0); \#2; \mathtt{C.p}(0); !$. Now #2 (first instruction) with change #1 is a Laski fault for X and so is #2 (third instruction) with change #1, so the pair of these is a multiple Laski fault. The same pair, however, is not a Laski multi-fault in X because changing both Laski faults creates $\mathtt{Y} \equiv \#1; \mathtt{C.p}(0); \#; \mathtt{C.p}(0); !$ which writes 00 instead of 0 and thus fails to comply with P.

In ISNwp[br] without the use of auxiliary registers a similar example can be found: assume that P requires of $\mathtt{X}$ that either $\mathtt{out0{:}1}$ is set to $\mathtt{1}$ or $\mathtt{out0{:}2}$ is set to 1. Now consider $\mathtt{X} \equiv \mathtt{out0{:}1.1/1}; \mathtt{out0{:}2.1/1}; !$ Then the pair $\mathtt{out0{:}1.1/1}$ with change $\mathtt{out0{:}1.0/0}$ is a Laski fault (of $\mathtt{X}$ w.r.t. P) and so is the pair $\mathtt{out0{:}2.1/1}$ with change $.\mathtt{out0{:}2.0/0}$. Thus the combination of both pairs is a multiple Laski fault, while it is not a Laski multi-fault, as combining both changes turns $\mathtt{X}$ into an instruction sequence which leaves both output registers unchanged.

Upon assuming that P requires that a function is computed the situation may be different, however, a matter which I leave as an unsolved question.

Problem 3.1 *For instruction sequences in ISNwp[br] not involving the use of auxiliary registers and for an arbitrary specification P_h which requires that* $\mathtt{X}$ *computes a total function h from inputs (valuations of input registers) to outputs (valuations of output registers): is each multiple Laski fault of* $\mathtt{X}$ *(with single instruction faults and parameter changes) of* $\mathtt{X}$ *a Laski multi-fault for* $\mathtt{X}$*?*

Many variations on the Laski fault pattern can be imagined. In particular I propose to consider the following fault patterns: n-Laski fault and weak n-Laski fault.

Definition 3.8 (n-Laski Fault) *The fragment f of* $\mathtt{X}$ *is an n-Laski fault w.r.t. P if (i)* $\mathtt{X}$ *does not meet specification P, and (ii) there is a change g such that* $\mathtt{LLOC}(g) \leq \mathtt{LLOC}(f) + \mathtt{n}$ *and such that after replacement of f by g in* $\mathtt{X}$*, the resulting instruction sequence* $\mathtt{X}_{g/f}$ *complies with P.*

An unnecessary risk of divergence may also be considered a fault, though not an ALR fault. As a risk it is a deficiency, rendering an understanding of the working of the instruction sequence unnecessarily hard, rather than a fault which causes a failure when the instruction sequence is being put into effect. And such a risk may be removed against a penalty of say n additional instructions. This leads to the following derived notion (though not suggested by Laski).

Definition 3.9 (Weak n-Laski Fault) *The fragment f of* $\mathtt{X}$ *is a weak n-Laski fault at position p in* $\mathtt{X}$ *w.r.t. P if (i) f is a subsequence of* $\mathtt{X}$ *beginning at position (instruction number) p, (ii) f contains one or more iteration instructions (so its execution may diverge), (iii) there is a (new) program fragment g without iteration instructions and with (iv)* $\mathtt{LLOC}(g) \leq \mathtt{LLOC}(f) + \mathtt{n}$*,*

such that (v) after replacement of f by g, the resulting instruction sequence $\mathtt{X}_{g/f}$ complies with P.

3.2 Mili, Frias, and Jaoua Faults (MFJ Faults)

In an extensive series of papers among which [39] and [25] a group of authors including Mili, Frias, and Jaoua develop a generalization of the approach of Laski, though without reference to Laski's work, which they may not have been aware of. These authors proceed by taking into account that repairing a fault in a program must only improve it, while the modified program need not be a correct implementation of the given specification after the first fault has been eliminated by being replaced by its change. Just as for STJoC faults and Laski faults, I will take the MFJ definition of a fault for a definition of a specific fault pattern.

Definition 3.10 (*MFJ Fault.*) *f is an MFJ fault in $\mathtt{X}$ w.r.t. specification P if (i) $\mathtt{X}$ does not correctly implement P, and (ii) there is a change g such that after replacement of f by g, the resulting instruction sequence $\mathtt{X}_{g/f}$ complies with P on strictly more inputs (i.e. a strictly larger set of inputs) than does $\mathtt{X}$.*

Moreover, in this situation g is called an MFJ change of the fault f.

Definition 3.11 (n-MFJ Fault) *The fragment f of $\mathtt{X}$ is a MFJ fault in w.r.t. specification P if (i) $\mathtt{X}$ does not comply with P, and (ii) there is a (new) program fragment g with $\mathtt{LLOC}(g) \leq \mathtt{LLOC}(f) + \mathtt{n}$ and such that after replacement of f by g, the resulting instruction sequence $\mathtt{X}_{g/f}$ complies with P on strictly more inputs than does $\mathtt{X}$.*

Definition 3.12 (Weak n-MFJ Fault) *f is a weak n-MFJ fault at position p in $\mathtt{X}$ w.r.t. P if (i) f is a subsequence of $\mathtt{X}$ beginning at position (instruction number) p, (ii) f contains one or more iteration instructions (so its execution may diverge), (iii) there is a (new) program fragment g without iteration instructions and with (iv) $\mathtt{LLOC}(g) \leq \mathtt{LLOC}(f) + \mathtt{n}$, such that (v) after replacement of f by g, the resulting instruction sequence $\mathtt{X}_{g/f}$ complies with P on at least all inputs where $\mathtt{X}$ complies with P.*

A common idea is that localization, determination, and subsequent change of a fault in a program may give rise to new failures, in spite of solving the failure which led to its discovery. This problem cannot arise with either Laski faults or MFJ faults. The idea of the elimination via a change

of a fault giving rise to new faults can be modelled, however, by assuming that output registers are ordered, from the least significant one (say `out:1`) to the most significant one (say `out:n`). Assuming that the specification P is a relation which extends a function the significance of a failure may be determined as follows: if no output is produced (the computation diverges, or fails to terminate properly) the significance is ∞, if on inputs σ outputs τ are produced such that $\neg P(\sigma, \tau)$ the significance of the failure is the highest significance of a bit which must be modified in order to transform τ to a result $\tau' \in P(\sigma)$.

The notion of a relative MFJ fault is introduced in order to allow, when preventing a failure of rank k by means of the change of a fault that new failures of lower rank are introduced.

Definition 3.13 (Relative MFJ Fault) *An ordering of significance is assumed on the output registers. f is a relative MFJ fault in* `X` *w.r.t. P if (i)* `X` *does not meet specification P, and (ii) there is a change g such that after replacement of f by g, the resulting instruction sequence* $\mathtt{X}_{g/f}$ *complies with P on some inputs on which* `X` *fails to comply, (iii) let k be the highest rank of an output for X which is compliant with P while the output for the same input of* `X` *is not compliant with P then all outputs where* $\mathtt{X}_{g/f}$ *fails to comply with P while the output of* `X` *complies with P have lower rank than k.*

The definition of a relative n-MFJ fault deviates only by requiring that $\mathtt{LLOC}(g)$ is not too large. The following question, for which I have no answer at the time of writing, can be posed for each instruction sequence notation ISN-$\mathbb{L}$.

Problem 3.2 *Given natural numbers k and n and a specification P which extends a function: is there an instruction sequence* `X` *in ISN-$\mathbb{L}$ which fails to comply with specification P and for which no relative n-MFJ fault can be found of size k or less.*

3.3 Elementary Properties of Laski Faults and of MFJ Faults

In this Paragraph some basic facts about various fault patterns will established. The specification P requires that a particular function form inputs to outputs is computed, wile the final value of auxiliary registers is ignored.

Proposition 3.1 *If* `X` *fails to compute P then the whole instruction sequence constitutes a single instruction n-Laski fault (w.r.t. P) for some n.*

Proof: Let $\mathtt{X_P}$ be a correct implementation of P such that $\mathtt{LLOC}(\mathtt{X_P}) = \mathtt{n}+\mathtt{1}$, then $\mathtt{X_P}$ constitutes an n-Laski change (w.r.t. P) for the first instruction $\mathtt{u_1}$ of $\mathtt{X}$. □

Proposition 3.2 *Suppose P determines a function with two inputs and one output. If* X *fails to compute P then the first instruction of* X *constitutes a single instruction 6-MFJ fault (w.r.t. P).*

Proof: W.l.o.g. suppose that the input registers used by X have focus in:1 and *in*:2 respectively, and that output register out:1 is used. Let $\mathtt{X} \equiv \mathtt{u_1};\mathtt{u_2};\ldots;\mathtt{u_n}$ X fails to compute the right result on input $(\mathtt{0},\mathtt{1})$ where the output prescribed by P is, say 1. Then u_1 is an MFJ fault which can be changed as follows: $+\mathtt{in{:}1.i/i};\#\mathtt{5};-\mathtt{in{:}2.i/i};\#\mathtt{3};\mathtt{out{:}1.1/1};!;\mathtt{u_1}$ □

The proof trivially generalises to the case with n inputs and m outputs, where the first instruction of a faulty instruction sequence can be shown to constitute a $2n+m+1$- MFJ fault. For practical purposes it is useless to work with changes of this form as programs get longer and longer.

The notion of an MFJ fault does not illuminate the deeper problem of this proposal for a change, i.e. that there seems to be no genuine sense in which $\mathtt{u_1}$ may be considered to be (part of) the cause of a failure.

Proposition 3.3 *Let P require that the function constant* 0 *is computed and consider the following instruction sequence:*

$\mathtt{X} \equiv +\mathtt{in{:}3.i/i}\{;\mathtt{out{:}5.0/1};\mathtt{out{:}6.0/0};\}\{;\mathtt{out{:}5.1/1};\mathtt{out{:}6.0/0};\};!.$

The following hold:

- X *is not correct w.r.t. P, (immediate,* X *fails on both input values),*
- X *does not contain a single instruction Laski fault (replacing the condition will not work because both branches contain an instruction which writes a* 1*. Replacing one of the branches does not work either, because the other branch will produce at least one wrong output.)*
- $f \equiv +\mathtt{in{:}3.i/i}\{;\mathtt{out{:}5.0/1}$ *with change* $g \equiv +\mathtt{in{:}3.1/i}\{;\mathtt{out{:}5.0/0}$ *constitutes a proper Laski fault of* X *w.r.t. P.*
- *the fragment* $f \equiv \mathtt{out{:}5.0/1}$ *is an 0-MFJ fault, with change* $g \equiv \mathtt{out{:}5.0/0}$*,*
- *the fragment* $f \equiv \mathtt{out{:}6.0/1}$ *is an 0-MFJ fault, with change* $g \equiv \mathtt{out{:}6.0/0}$*.*

Proposition 3.4 *Let P be the specification which requires that the function constant* 1 *(on all output registers) is computed. Determination of f being a proper Laski fault for* X *w.r.t. P is NP hard in the number of inputs. The results persists under the constraint that* X *cannot make use of auxiliary registers.*

Proof: We consider instruction sequences with k inputs and a single output. Let $p(u_1, \ldots, u_k)$ be a proposition in k Boolean variables. $\mathtt{Y}_{\neg\mathtt{p}};!$ is an instruction sequence which reads the k inputs and evaluates $\neg p$ on the respective values writing the result in register out:1, and then terminates with the final ! instruction. Thus, if p is satisfiable at for one input vector p holds and an output 0 will be produced, otherwise in all cases a 1 is written. For the construction of $\mathtt{Y}_\mathtt{p}$ I refer to [6].

Consider $\mathtt{X}_{\neg\mathtt{p}} \equiv \mathtt{Y}_{\neg\mathtt{p}};\#1;!$, choose $f \equiv \#1$ (at the one but last position), and consider $g \equiv \mathtt{out{:}1.1/1}$. Now one observes: f is a Laski fault in $\mathtt{X}_{\neg\mathtt{p}}$ $\iff$ f is a 1-Laski fault in $\mathtt{X}_{\neg\mathtt{p}}$ $\iff$ f is a Laski fault with change g in $\mathtt{X}_{\neg\mathtt{p}}$ $\iff$ p is satisfiable. □

Proposition 3.5 *There is an instruction sequence* X *with k inputs and $l+2$ outputs which fails to compute the function which assigns* 1 *to each output register (as is required by specification P), and which nevertheless has no l-MFJ fault.*

Proof: Let $\mathtt{X} \equiv \mathtt{out0{:}1.1/1};!$. By replacing a proper subsequence f of X by a change g of length at most $1+l$ an instruction sequence results with length at most $\mathtt{LLOC(X)}+\mathtt{1}(=\mathtt{2}+\mathtt{1})$, which has at most $l+1$ instructions able to set an output register to 1. But $l+2$ of such instructions are needed to compute a single correct output on whatever inputs, which yields a contradiction. □

Proposition 3.6 *Suppose that* X *has k input registers and l output registers, then: if (i)* X *is not correct w.r.t. P, and (ii) if in particular there is an invalid output (w.r.t. P) on some inputs, which is not* 0 *on all output registers, then* X *contains a single instruction $(2 \cdot k + l)$- MFJ fault.*

Proof: Suppose that X computes on inputs γ outputs γ' such that $\neg P(\gamma, \gamma')$ holds and such that γ' assigns at least to one of the output registers, say out0:i the value 1. Let α be the progression which X generates on γ, then there must be at least one instruction of X, say $\mathtt{u}_\mathtt{j}$ such that $\mathtt{u}_\mathtt{j} \equiv \mathtt{out0{:}j.1/1}$ which occurs in α.

We take $f \equiv \mathtt{u_j}$, and g as follows: $g \equiv \mathtt{w_1};\ldots;\mathtt{w_k};\mathtt{v_1};\ldots;\mathtt{v_l};!$ where: for $t \in [1,k]$: if $\gamma_t = 0$ then $\mathtt{w_t} \equiv -\mathtt{int.i/i};\#(\mathtt{1}+\mathtt{2}\cdot(\mathtt{k}-\mathtt{t})+\mathtt{l})$ and if $\gamma_t = 1$ then $\mathtt{w_t} \equiv +\mathtt{int.i/i};\#(\mathtt{2}\cdot(\mathtt{k}-\mathtt{t})+\mathtt{l}+\mathtt{2})$, and $\mathtt{v_i} \equiv \mathtt{out0{:}i}.\gamma_\mathtt{i}/\gamma_\mathtt{i}$. We find that $\mathtt{X_{f/g}}$ has the result γ' on input γ and the same result as X for other inputs. Therefore $\mathtt{X_{f/g}}$ is correct w.r.t. P for the inputs for which X computes correctly plus one.

The proof is imprecise because g is not written by means of structured programming instructions. That can be done as follows (with an replacement of the same LLOC:

$g' \equiv \mathtt{w'_1};\ldots;\mathtt{w'_k};\mathtt{v_1};\ldots;\mathtt{v_l};!;\};\ldots;\}$ where for $t \in [1,k]$: if $\gamma_t = 0$ then $\mathtt{w_t} \equiv -\mathtt{int.i/i}\{$ and if $\gamma_t = 1$ then $\mathtt{w_t} \equiv +\mathtt{int.i/i}\{$. □

We now imagine that X is able to make use of a service H_{tmt} which embodies the behaviour of a Turing tape (see [19] for such services). Thus besides method calls for single bit registers X may also involve method calls for operating H_{tmt}. The tape is initially empty and plays no role for input and output, it has an auxiliary status only. Now the following Proposition can be asserted.

Proposition 3.7 *For instruction sequences with a single input register* in:1 *and a single output register* in:1 *we assume as the specification P that the input is copied into the output. Now for* X *making use of H_{tmt} the following is undecidable:*

- *f is a Laski fault in* X*,*
- *f is a proper Laski fault in* X*,*
- *(f,g) is a Laski fault with change in* X*,*
- *f is a MFJ fault in* X*,*
- *f is a proper MFJ fault in* X*,*
- *(f,g) is a MFJ fault with change in* X*,*

Proof: We notice that X is a Laski fault in X $\iff$ X is a proper Laski fault in in:1.i/i;X $\iff$ X correctly implements P (assuming total correctness) $\iff$ either X diverges on one of its (two) arguments, or if it changes at least one of the inputs when writing to the output register. Now let W_e be a non-computable computably enumerable set. $\mathtt{Y_{e,n}}$ works as follows for an integer n: (i) write e and n in binary notation into H_{tmt}, with

an appropriate marker in between (this work is done by $\mathtt{LOAD_e};\mathtt{LOAD_n};!$, (ii) apply a universal TM program, say $\mathtt{X_{utm}};!$ to the contents of H_{tmt} which terminates if and only if $n \in W_e$.

Now choose $\mathtt{X_n}$ as follows:

$\mathtt{X_n} \equiv \mathtt{LOAD_e};\mathtt{LOAD_n};\mathtt{X_{utm}};+\mathtt{in{:}1.i/i}\{;\mathtt{out{:}1.0/0};\}\{;\mathtt{out{:}1.1/1};\};!.$

We notice that $\mathtt{X_n}$ correctly implements P if and only $n \in W_e$.

This proves the first item, other items have similar proofs. □

3.4 Multiple MFJ Faults, Multi-Hunk MFJ Faults and MFJ Multi-Faults

Candidate faults f and g of X are non-overlapping if they do not share any instructions.

Definition 3.14 (Multiple MFJ Faults) *An instruction sequence* $\mathtt{X}$ *contains multiple MFJ faults* $f_1, \ldots, f_n$ *with MFJ changes* $g_1, \ldots, g_n$ *respectively if each pair* f_i, g_i *is an MFJ fault for* $\mathtt{X}$ *and the fault subjects are* f_i *pairwise non-overlapping.*

It is intuitively clear that an instruction sequence may contain several different MFJ faults. Following the terminology of [47] multi-hunk faults require simultaneous changes in different locations. The idea of a multi-hunk fault can be specialized to MFJ faults as follows.

Definition 3.15 (MFJ Multi-Hunk Fault) *A set of disjoint (i.e. non-overlapping) fragments* $f_1, \ldots, f_n$ *is an MFJ multi-hunk fault in* $\mathtt{X}$ *w.r.t.* P *if (i)* $\mathtt{X}$ *does not meet specification* P*, (ii) there are changes* $g_1, \ldots, g_n$ *such that after replacement of* f_i *by* g_i *for* $1 \leq i \leq n$*, the resulting instruction sequence* $\mathtt{X}_{\tilde{g}/\tilde{f}}$ *complies with* P *on strictly more inputs than does* $\mathtt{X}$*, and (iii) no proper subset of* $f_1, \ldots, f_n$ *is an MFJ multi-hunk fault. Here* n *is the width of the multi-hunk fault.*

In contrast with the case of Laski faults, for MFJ faults, the idea of a multi-fault which is not a multi-hunk fault is plausible.

Definition 3.16 (MFJ Multi-Fault) *A set of disjoint (i.e. non-overlapping) fragments* $f_1, \ldots, f_n$ *is an MFJ multi-fault in* $\mathtt{X}$ *w.r.t.* P *if (i)* $\mathtt{X}$ *does not meet specification* P*, and (ii) there are (new) program fragments* $g_1, \ldots, g_n$ *such that each* g_i *is an MFJ change for* f_i*, and (iii) for each nonempty set* $U \subseteq \{1, .., n\}$*, after simultaneous replacement of* f_i *by* g_i

for $i \in U$, the resulting instruction sequence $\mathtt{X}^{\mathtt{U}}_{\tilde{g}/\tilde{f}}$ complies with P on strictly more inputs than does $\mathtt{X}$*. Here n is the width of the multi-fault.*

The following fault pattern matches with the plausible intuition that upon eliminating all (MFJ) faults a correct implementation of P is obtained.

Definition 3.17 (Laski Strength Multiple MFJ Fault) *A set of disjoint (i.e. non-overlapping) fragments $f_1, \ldots, f_n$ is a Laski strength MFJ multi-fault in* $\mathtt{X}$ *w.r.t. P if (i)* $\mathtt{X}$ *does not meet specification P, and (ii) there are (new) pairwise non-overlapping changes $g_1, \ldots, g_n$ such that each g_i is an MFJ change for f_i (i.e. these faults and changes are a multiple MFJ fault for* $\mathtt{X}$*), and (iii) after simultaneous replacement of f_i by g_i for $i \in U$, the resulting instruction sequence $\mathtt{X}^{\mathtt{U}}_{\tilde{g}/\tilde{f}}$ complies with P (i.e the f_i and g_i constitute a multi-hunk Laski fault.*

Definition 3.18 (Orthogonal MFJ Multi-Fault) *A set of disjoint (i.e. non-overlapping) fragments $f_1, \ldots, f_n$ is an orthogonal MFJ multi-fault in* $\mathtt{X}$ *w.r.t. P if (i)* $\mathtt{X}$ *does not meet specification P, and (ii) there are (new) program fragments $g_1, \ldots, g_n$ such that each g_i is an MFJ change for f_i, and (iii) for each pair of sets U, V with $\emptyset \neq U \subsetneq V \subseteq \{1, .., n\}$, after simultaneous replacement of f_i by g_i for $i \in V$, the resulting instruction sequence $\mathtt{X}^{\mathtt{V}}_{\tilde{g}/\tilde{f}}$ complies with P on strictly more inputs than does $\mathtt{X}^{\mathtt{U}}_{\tilde{g}/\tilde{f}}$. Here n is the width of the multi-fault.*

3.5 An Example with Multiple Faults of Size One

Let specification P require that for all inputs all outputs (which are initially set to 0) are set to 1 Consider $\mathtt{X} = +\mathtt{in{:}1.1/i}\{;\mathtt{out0{:}1.1/1};\mathtt{out0{:}2.1/1};\};!$. We will now look at 0-Laski faults and 0-MFJ faults of size 1 only. We can apply single fault injection (of such faults) in three different ways as follows:

$\mathtt{X_1} = +\mathtt{in{:}1.0/i}\{;\mathtt{out0{:}1.1/1};\mathtt{out0{:}2.1/1};\};!$.
$\mathtt{X_2} = +\mathtt{in{:}1.1/i}\{;\mathtt{out0{:}1.1/0};\mathtt{out0{:}2.1/1};\};!$.
$\mathtt{X_3} = +\mathtt{in{:}1.1/i}\{;\mathtt{out0{:}1.1/1};\mathtt{out0{:}2.1/0};\};!$.

And double fault injection as follows:

$\mathtt{X_{1,2}} = +\mathtt{in{:}1.i/i}\{;\mathtt{out0{:}1.1/0};\mathtt{out0{:}2.1/1};\};!$.
$\mathtt{X_{2,3}} = +\mathtt{in{:}1.1/i}\{;\mathtt{out0{:}1.1/0};\mathtt{out0{:}2.1/1};\};!$.
$\mathtt{X_{2,3}} = +\mathtt{in{:}1.1/i}\{;\mathtt{out0{:}1.1/1};\mathtt{out0{:}2.1/0};\};!$.

And triple fault injection by combining each of these options:

$\mathtt{X_{1,2,3}} = +\mathtt{in{:}1.i/i}\{;\mathtt{out0{:}1.1/0};\mathtt{out0{:}2.1/0};\};!$.

Counting fault patterns as defined above, however, does not support this informal counting of faults. It is easy to check the following observations

- $f \equiv$ +in:1.0/i{; out0:1.1/0 with change $g \equiv$ +in:1.1/i{; out0:1.1/1 is a Laski fault with subject f and change g w.r.t. P.
- that $\mathtt{X}_{1,2}$ contains a single proper MFJ fault only, which is a Laski fault at the same time,
- $\mathtt{X}_{2,3}$ contains a multi-hunk MFJ fault of size 2: $f_1 \equiv$ +in:1.i/i{ $g_1 \equiv$ +in:1.1/i{ and $f_2 \equiv$ +out0:2.1/0, $g_2 \equiv$ + + out0:2.1/1 This MFJ multi-hunk fault is not an MFJ multiple fault, neither is it an MFJ-multi-fault of $\mathtt{X}_{2,3}$. These two faults can be eliminated in either order, in both cases turning the other fault into a 0-Laski fault.
- $\mathtt{X}_{1,2,3}$ contains no single instruction 0-MFJ fault and no single instruction 0-Laski fault. In order to turn $\mathtt{X}_{1,2,3}$ into a correct implementation of P the transition to $\mathtt{X}_{2,3}$ must be made by undoing the first of the fault injections, which, however, in $\mathtt{X}_{1,2,3}$ does not have the status of a fault.

This example suggest that fault injection may lead to "faults" which lie outside any of fault patterns which have been introduced thus far in this paper and which cannot be detected via testing in a completely straightforward manner.

3.6 Too Short Instruction Sequences Have No Laski Faults

Suppose P is a specification for a relation from n input registers to m output registers, and suppose that instruction sequences are written in ISNwp[br], where 0-initialized auxiliary single bit registers are admitted. Let l_{min} be the LLOC of (a) shortest implementation of P in ISNwp[br].

Proposition 3.8 *Now suppose that* X *is a ISwp[br] instruction sequence with LLOC below l_{min}, where auxiliary registers are admitted. Then (i)* X *cannot contain any Laski faults (because even changing all instructions is not enough), (ii)* X *has no Laski multi-faults, and (iii) the failure of* X *w.r.t. P is a non-local defect of* X*.*

In the above proposition X may contain MFJ faults of depth 0 or of depths above 0.

3.7 Better Definitions of "Better Programs"

The last observation in Paragraph 3.5 implies that multiple fault injection can result in an instruction sequence which contains no MFJ faults. This somewhat implausible consequence can be remedied by adapting the definition of an MFJ fault in such a manner that instead of asking that upon change of a fault the resulting instruction sequence is "better", that is producing P-compliant output on strictly more arguments, it is required that the resulting instruction sequence is potentially better: it contains a fault which can be resolved so that a better (or even a potentially better) instruction sequence is obtained.

Definition 3.19 (MFJ* Fault) *f is an MFJ* fault in $\mathtt{X}$ w.r.t. P if (i) $\mathtt{X}$ does not meet specification P, and (ii) there is a change g such that after replacement of f by g, either (i) the resulting instruction sequence $\mathtt{X}_{g/f}$ complies with P on strictly more inputs than does $\mathtt{X}$ (i.e.f is an MFJ fault in $\mathtt{X}$), or (ii) the instruction sequence $\mathtt{X}_{g/f}$ complies with P on the same inputs as does $\mathtt{X}$ (but perhaps with different outcomes) and $\mathtt{X}_{g/f}$ contains an MFJ* fault.*

In this definition h is a descendant of f and an MFJ* fault has one more more chains of descendants of maximal length, while an MFJ fault is an MFJ* fault without descendants. The depth of an MFJ* fault is size of the shortest non extendable chain of descendants for it. Thus an MFJ fault is an MFJ* fault with depth 0.

Definition 3.20 *An MFJ* fault f in $\mathtt{X}$ is proper if it has a non-extendable chain of descendants such that the sum of the $\mathtt{LLOC}$'s of these is below $\mathtt{LLOC}(\mathtt{X})$.*

For the following fault pattern many variations can be found, for instance $\mathtt{Y}$ (as in the definition) may be required to contain an MFJ$\star$ fault with subject disjoint from any of the f_i, or even an MFJ* multi-hunk fault

Definition 3.21 (MFJ* Multi-Hunk Fault) *A multi-hunk MFJ* fault in an instruction sequence $\mathtt{X}$, relative to the specification P is a set of disjoint (i.e. non-overlapping) fragments $f_1, \ldots, f_n$ with changes $g_1, \ldots, g_n$ such that upon replacement of f_i by g_i for $1 \leq i \leq n$, for the resulting instruction sequence $\mathtt{Y} \equiv \mathtt{X}_{\tilde{g}/\tilde{f}}$ the following holds: (i) $\mathtt{X}$ does not meet specification P, (ii) either $\mathtt{Y}$ complies with P on strictly more inputs than does $\mathtt{X}$ (but perhaps*

with different outcomes) or Y *contains an MFJ fault w.r.t. P, and (iii) no proper subset of $f_1, \ldots, f_n$ is an MFJ* multi-hunk fault.*

Proposition 3.9 $\mathtt{X}_{\{1,2,3\}}$ *in Paragraph 3.5 has three disjoint MFJ* faults of size 1 and non-zero depth.*

If one assumes that the domain of the transformation computed by X is finite then another option for MFJ comes available: requiring that for a larger fraction of inputs the output conforms with P.

Definition 3.22 (MFJ$_{fd}$ Fault) *It is assumed that computed functionalities have a finite domain D. The fragment f of* X *is an MFJ$_{fd}$ fault in* X *w.r.t. P if (i)* X *does not meet specification P, and (ii) there is a change g such that after replacement of f by g, the resulting instruction sequence $\mathtt{X}_{g/f}$ complies with P on a larger number of inputs than does* X.

This alternative definition may also be adapted to non-zero depth, then obtaining the notion of an MFJ$^\star_{fd}$ fault.

3.8 Essential Faults, Including ALR Faults which Are Not MFJ Faults

Not all instructions which need to be changed in order to turn an instruction sequence into compliance with its specification P are faults of the kinds mentioned above.

Using the notion of a multi-hunk fault it is possible to determine faults which can only indirectly be seen to be in need of revision. In this Paragraph so-called essential faults are introduced, in fact three patterns of such faults. Although essential faults need not be Laski faults, MFJ faults or variations thereof, an essential fault qualifies as an ALR fault, as its presence blocks compliance with the given specification, though the argument for that blockade is less direct than in the case of Laski faults or in the case of MfJ faults.

Definition 3.23 (Laski Essential Fault) *Given an instruction sequence* X *and a specification P for which at least one Laski multi-hunk fault is known, a Laski essential fault is a candidate fault f with change g the subject of which is a member, or is a fragment of a member, of each Laski multi-hunk fault of* X *w.r.t. P.*

Definition 3.24 (MFJ Essential Fault) *Given an instruction sequence* X *w.r.t. a specification P for which at least one MFJ multi-hunk fault is known, an MFJ essential fault is a candidate fault f the subject of which is a member of all MFJ multi-hunk faults of* X *w.r.t. P.*

Definition 3.25 (MFJ* Essential Fault) *Given an instruction sequence* X *w.r.t. a specification P for which at least one Laski multi-fault is known, an MFJ* essential fault is a candidate fault f the subject of which which is a member of all MFJ* multi-hunk faults of* X *w.r.t. P.*

For this Paragraph I will assume that P requires that for all inputs on registers 3 and 4 the output registers 1, 2, and 3 are set to 1. Now consider the following IS:

$\mathtt{X} \equiv +\mathtt{in}{:}3.1/\mathtt{i}\{;\mathtt{X}_1;\}\{;\mathtt{X}_2;\};!$ with

$\mathtt{X}_1 \equiv \mathtt{out}{:}1.1/1$ and

$\mathtt{X}_2 \equiv +\mathtt{in}{:}4.1/\mathtt{i}\{;\mathtt{out0}{:}2.1/1;\}\{;\mathtt{out0}{:}1.1/1;\mathtt{out0}{:}2.1/1;\mathtt{out0}{:}3;1/1;\}$.

About X the following observations can be made:

1. X is not correct w.r.t. P as it leaves registers 2, and 3 at their initial value 0.

2. Taking $\mathtt{f}_1 \equiv +\mathtt{in}{:}3.1/\mathtt{i}\{$, $\mathtt{g}_1 \equiv +\mathtt{in}{:}3.0/\mathtt{i}\{$, and $\mathtt{f}_2 \equiv +\mathtt{in}{:}4.1/\mathtt{i}\{$, $\mathtt{g}_2 \equiv +\mathtt{in}{:}4.0/\mathtt{i}\{$, constitutes a Laski multi-fault of X, and also a multi-hunk Laski fault.

3. No simultaneous change of any combination of output instructions constitutes a Laski multi-fault.

4. Each Laski multi-fault must contain (i.e. modify) the first test instruction. Indeed otherwise unavoidably at least one output register is left unchanged, because the unchanged test with lead to the execution of $\mathtt{X}_1$ which, even after changes can perform only one assignment to a bit valued register. Here it is used that we only consider changes consisting of a single instruction, otherwise the situation is entirely different.

5. In particular $f_1 \equiv +\mathtt{in}{:}3.1/\mathtt{i}\{$ is an essential fault because X cannot be corrected without changing the instruction sequence at that location.

6. No MFJ fault which changes the first test (f_1) can be found, i.e. no choice of a change g for it creates an MFJ fault. Indeed either the

change leaves the control flow unchanged, in which case the resulting instruction sequence does the same and is not better (in the sense of taking the correct value on a strictly larger set of inputs), or the modification changes the flow of control, in which case an instruction `out:1.1/1` is not performed so that the resulting output becomes wrong for an input for which it was already correct in advance of the change.

7. f_1 is an essential fault, but it is not a Laski fault, not an MFJ fault, and not an MFJ fault of depth > 0.

8. f_1 is an ALR fault because its presence stands in the way of compliance of the behaviour of `X` with P.

9. f_1 is an example of a fault which must be eliminated. though for which any change introduces at least one new failure.

3.9 A Laski Multi-Hunk Fault, None of the Members of which Are MFJ* Faults

For this Paragraph I will work with ISNwp[br]. The specification P is supposed to require that for all inputs on registers `in:0`, ..., `in:4` the (0-initialized) output registers `out0:1`, ..., `out0:4` are set to the content of the corresponding input registers. Now consider the following IS:

$\mathtt{X} \equiv \mathtt{out0{:}1.1/1}; +\mathtt{in{:}0.0/i}\{; \mathtt{X_1}; \}; +\mathtt{in{:}0.0/i}\{; \mathtt{X_2}; \};$
$-in{:}1.i/i\{; out{:}1.0/0; \}; -in{:}2.i/i\{; out{:}2.0/0; \};$
$-in{:}3.i/i\{; out{:}3.0/0; \}; -in{:}4.i/i\{; out{:}4.0/0; \}; !$

with

$\mathtt{X_1} \equiv \mathtt{out{:}1.0/0; out0{:}2.1/1; out0{:}3.1/1; aux0{:}1.1/1}$ and
$\mathtt{X_2} \equiv +\mathtt{aux0{:}1.i/i}\{; \mathtt{out0{:}1.1/1}; \}\{; \mathtt{out{:}1.0/0}; \}; \mathtt{out0{:}4.1/1}.$

About `X` the following observations can be made:

1. `X` fails to comply with P as, on input vector $(0, 1, 1, 1, 1)$ it produces $(1, 0, 0, 0)$ instead of $(1, 1, 1, 1)$.

2. Taking $f_1 \equiv +\mathtt{in{:}0.0/i}\{$ (the first test) , $g_1 \equiv +\mathtt{in{:}0.1/i}\{$, and $f_2 \equiv +\mathtt{in{:}0.0/i}\{$ (the second test), $g_2 \equiv +\mathtt{in{:}0.1/i}\{$, the pair $((f_1, g_1), (f_2, g_2))$ constitutes a Laski multi-fault of `X`.

3. (f_1, g_1) is not an MFJ* fault of `X` because applying $\mathtt{X}_{f_1/g_1}$ to $(0, 1, 0, 0, 0)$, the result of $\mathtt{X}_{f_1/g_1}$ is $(0, 0, 0, 0, 0)$ which is not correct (w.r.t. P) while `X` produces $(0, 1, 0, 0, 0)$ which is correct.

4. (f_2, g_2) is not an MFJ* fault of $\mathtt{X}$ for because $\mathtt{X}_{f_2/g_2}$ will cause failure (with output $(0,0,0,0)$), on input $(0,1,0,0,0)$, (which as is mentioned above is correctly processed by $\mathtt{X}$), this time because $\mathtt{aux0{:}1}$ still has its initial value 0.

5. (f_1, g_1) is a Laski essential fault, because unless f_1 is modified $\mathtt{out0{:}2}$ and $\mathtt{out{:}3}$ cannot each be assigned 1 (irrespective of any changes made inside $\mathtt{X}_2$, which can perform only two assignments (although it contains three assignments).

6. (f_2, g_2) is not a Laski essential fault. Indeed the combination (f_1, g_1), (f_3, g_3) with $f_3 \equiv (\mathtt{out{:}1.0/0}, \mathtt{out{:}4.1/1})$ where f_3 is located inside $\mathtt{X}_1$, constitutes a Laski multi-fault which does not contain (f_2, g_2).

The following questions are left open.

Problem 3.3 *Is there an example of a Laski multi-fault with two members such that neither member is an MFJ* fault and both of which are Laski essential faults?*

The same question can be raised with MFJ or with MFJ* instead of Laski.

Problem 3.4 *Is there an example of a Laski multi-fault with two members such that neither member is an MFJ* fault without the use of auxiliary registers?*

Again the same question can be raised with MFJ or with MFJ* instead of Laski.

4 Fault Patterns Linked to Regression Testing and Fault Localization

Testing may create confidence in the correctness of $\mathtt{X}$ being an implementation of a specification P. Testing may also create confidence in the correctness of $\mathtt{X}_{g/f}$ resulting from replacing a candidate fault f by a candidate change of it. Rather than generating a test suite for the latter purpose one may use the test suite for which $\mathtt{X}$ has already been observed to be in compliance with P. This leads to the idea of regression testing for justification a proposed change. Because the text of f and g plays no role in this form of justification it is qualified as a formal justification of change. These suggestions have already been taken into account in Section 2 above. The idea of an essential fault can be adapted to a regression testing pattern as well.

4.1 Essential Faults in Connection with Regression Testing

The idea of an essential fault transfers to regression test justified changes. The example of Paragraph 3.9 is of relevance for regression testing as well. Assume that the current regression test suite $(\beta_1, \ldots, \beta_n)$ is a test progression suite which contains no progression starting from an input of the form $(b_0, 1, b_2, b_3, b_4)$, and from input $z = (0, 0, 1, 0, 0)$ then the following may noticed, using the notations of paragraph 3.9:

- `X` fails on z by producing $(0, 0, 0, 0, 0)$ instead of $(1, 0, 0, 0)$.
- $\mathtt{X}_{f_1/g_1}$ succeeds on z by producing $(0, 1, 0, 0)$.
- $\mathtt{X}_{f_1/g_1}$ succeeds on all regression tests, as the only test cases where it would fail when `X` works well, take `1` as the input for `in:1`, and such test cases are, by assumption, not represented in the test suite at hand.
- Therefore (f_1, g_1) is a regression test justified resolved fault.
- `X` can succeed on tests on inputs $(b_0, 0, 0, 0, 0)$ so it may be concluded that under the given assumptions the regression test suite contains two progressions at best.

If on the other hand the test suite contains a test starting from input say $(1, 1, 0, 0, 0)$ then g_1 is not a regression test justified change of the fault f_1.

4.1.1 Ranked Regression Justification of Change

The notion of a regression test suite compliant fault can be adapted by labeling the test progressions with a level of relevance chosen from say $l_1, \ldots, l_r$, and assigning α a level of relevance, say $l \in \{l_1, \ldots, l_r\}$ and requiring only that progressions β_i with relevance above l are protected (i.e. transformed into progressions with the same output or in any case with an output compliant with P) against the modification of f into g.

Returning to the example of Paragraph 3.9 now assume that `out:1` contains the least significant output and `out:4` the most significant one and that a progression is considered more relevant than another progression if the disagreement with desired outputs (i.e. with P) occurs in output registers of lower significance. The following fact can be noticed:

- Suppose at some stage regression the test suite has four elements $(\beta_1, \ldots, \beta_4)$ with respective inputs $(0,0,0,0,0)$, $(1,0,0,0,0)$, $(0,1,0,0,0)$, and $(1,1,0,0,0)$. $\mathtt{X}$ works correctly on each of these.

- Consider $z' = (1,0,1,0,0)$, and let α be the progression in this input of $\mathtt{X}_{f_1/g_1}$. $\mathtt{X}_{f_1/g_1}$ succeeds on z' producing z' but it fails on (the inputs of) β_3 and on β_4. However, as α agrees with P on more significant outputs than β_3 and on β_4 it is ranked higher than both.

- (f_1, g_1) is a ranked regression test justified resolved fault.

4.2 Spectrum Based Fault Localization (SBFL) for Instruction Sequences

Following e.g. [54] given an instruction sequence $\mathtt{X} \equiv \mathtt{u}_1; \ldots; \mathtt{u}_\mathtt{n}$ and a test suite $t_1, \ldots, t_m$ for it, consisting of progressions labeled with $\mathtt{p}$ for pass (i.e. succeed) or $\mathtt{f}$ for fail, the following numbers are defined: $a^i_{ef}, a^i_{ep}, a^i_{nf}, a^i_{np}$, which respectively denote the following quantities: the number of progressions in the suite that visit (effectuate) u_i and fail, the number of progressions in the suite that visit u_i and pass, the number of progressions in the suite that do not visit u_i and fail, the number of progressions in the suite that do not visit u_i and pass. The sequence of such 4-tuples for all instructions in an instruction sequence is called its spectrum for the test suite.

The relevance of this idea comes from the objective to automate the search for candidate faults. The idea is to generate a test suite and then for each instruction u_i to compute a value R_i denoting the risk that u_i is faulty, where, remarkably, no definition of fault needs to be assumed. The so-called assumption of "perfect bug detection" is made which guarantees that if an instruction is inspected for being faulty, this judgement can be made and is made in a reliable manner. There is a remarkable variety in formulae for risk determination which has been proposed. A survey can be found in [54] where comparisons are made on theoretical grounds.

In the context of ISNwp[br] with single instruction faults, restricted to parameter faults, we may relate SBFL to the various definitions of fault and change We consider a single example only in order to highlight the idea.

Proposition 4.1 *Assuming that* $\mathtt{LLOC}(\mathtt{X}) = \mathtt{n}$*, and that* $\mathtt{X}$ *works on* k *single bit input registers, and that a test suite* $t_1, \ldots, t_{2^k}$ *is given which contains a test for each of the possible inputs, the following holds:*

(i) *if u_i is a Laski fault (with a single instruction faults, parameter faults only, and single changes), then $a^i_{ef} > 0$.*

(ii) *if u_i is an MFJ fault (with a single instruction faults, parameter faults only, and single changes), then $a^i_{ef} > 0$.*

(iii) *if* `X` *contains a unique Laski fault u_j different from u_i then it need not be the case that the spectrum shows a difference between i and j.*

Proof: (i) follows from the definition of a Laski fault as there must be a failed progression which does not fail anymore after a replacement of u_i by a change for it, form which it follows that the failed progression must visit u_i. The proof of (ii) is similar. For (iii), let $\mathtt{X} \equiv \mathtt{out0{:}1.1/0; in{:}1.i/i; !}$, assume that P requires the only output register to be set to 1. Now u_1 is a fault because it can be replaced by $\mathtt{X} \equiv \mathtt{out0{:}1.1/1}$ which resolves the problem of non-compliance with P. But u_2 is not a fault as no replacement for it makes `X` compliant with P. However, as each progression visits both u_1 and u_2 there is no difference between the spectrum of the two instructions.

Remark. If one adopts single instruction changes but drops the constraint that only parameter faults are considered the situation changes because then both u_1 and u_2 are Laski faults and it is more plausible that both instructions cannot be distinguished in the spectrum. □

Proposition 4.2 *Working in* ISNwp[br] *with a fixed and finite number of input registers and a fixed and finite number of 0-initialised output registers, and an arbitrary number of 0-initialized auxiliary registers and assuming that faults must be single instructions while changes may be arbitrary instruction sequences from* ISNwp[br]*, and moreover assuming that specification P requires that some function is computed, and that $\mathtt{X} \equiv \mathtt{u_1; \ldots; u_n}$ does not satisfy P, the following holds:*

(i) *if for some test suite $A^i_{ef} > 0$ then u_i is an MFJ-fault in* `X` *w.r.t. P.*

(ii) *u_1 is a Laski fault in* `X`*.*

The above results have as an important consequence that the very assumption that SBFL is a useful technique, cannot be taken for granted as it presupposes constraints on the context parameters as mentioned in Paragraph 1.2.3. The perfect bug detection assumption implies that given a

candidate fault it is relatively easy to determine if it is indeed a fault and how it can be resolved. None of of the definitions of special cases of ALR faults, as presented above, seem to support this assumption. One is led to believe that SBFL is not about ALR faults, and is instead aiming at finding and resolving some other kind of defects, for which I have not yet been able to find a convincing definition.

4.2.1 CPH and PBD as Cornerstones of SBPL

The CPH (competent programmer hypothesis), see e.g. [29] asserts that the program being searched for faults is rational and is close to a working implementation of its specification. CPH explains why it is reasonable to expect the detection of a series of faults and changes thereof to lead to software of improved quality. CPH defeats any analysis in terms of elementary theoretical concepts. Together with the perfect bug detection hypothesis PBD, a context arises in which the usefulness of SBPL is plausible, a state of affairs which is hardly explainable when thinking in terms of ALR faults only. CPH and PBD are not part of the formal description of faults, and can only work if non-formal definitions of fault come into play.

5 Algorithm Conformance Oriented Justification of Change

Remarkably and importantly none of the formally justified definitions of instruction sequence faults as discussed in the above Sections explains, or helps to understand, the use of program faults in practice, in fact not even in the the majority of papers on faults in software engineering research.

Apparently a definition of a program fault, or of an instruction sequence, must exist or must be sought, which differs from the definitions based on formal justification as listed above.

5.1 Defining Program Faults Rather Than Instruction Sequence Faults

In this approach we will not make use of instruction sequences as substitute for programs, instruction sequences will however be used as a substitute for pseudocode. We take for granted that the listed definitions of instruction sequence fault classes have plausible counterparts for each imperative program notation. The reason not to consider instruction sequence faults but

rather program faults for programs in a known program notation lies in the necessity to make use of the notion of an experienced programmer in the given notation. Working with a theoretical framework such as instruction sequences, assuming the existence of experienced programmers is too much of a hypothetical idea.

Definition 5.1 (Algorithm)

(i) *An algorithm is an algorithmic method.*

(ii) *An algorithmic method, is a method, say M, for solving a certain problem (i.e. for achieving a promised outcome given adequate inputs) in a stepwise manner.*

(iii) *Moreover, and more precisely, an algorithmic method is a method which can in principle be documented by means of a program where a program is an entity the meaning of which is primarily determined by a uniform translation (also calleed projection, which is defined for all elements of a given program notation) into a sequence of instructions.*

For the notion of an instruction sequence and the consequences of requiring finiteness thereof we refer to [16]. An instruction sequence which implements the documenting instruction sequence is referred to as an implementation of the algorithm. Our definition of an algorithm is somewhat more specific than most definitions conventionally used in computer science.

In fact all parts (phases, branches) of the method as well as its overall architecture can be documented by means of programs.

(iv) *The existence of an all-encompassing documenting instruction sequence is hypothetically assumed. An execution of the method corresponds to putting the documenting instruction sequence into effect.*

(v) *An implementation of M is a computer program, meant for use on a known computer architecture, which conforms the documenting instruction sequences.*

(vi) *In modern public language "algorithm" also refers to an agent (a computer system or a cyber-physical system) which operates under software control and which in a certain phase is under control of a running implementation of an algorithm as referred to in (v). This*

derived meaning of algorithm focuses on a black box view of the mentioned agent, while dropping information about its internal details of an implementation or of a documentation (if available). The latter understanding of algorithm renders it amenable for usage in debate among software non-specialists.

Some remarks on this definition are in order:

1. It is only required that the algorithm can be documented by means of one or more instruction sequences, it is not required that such is actually done, if a family of documenting instruction sequences is available which documents core phases of the method, it is not required that an integration of these in an integrating polyadic instruction sequence is known or available.

2. It is tempting to identify an algorithm with a documenting instruction sequence of it, if available. The idea, however, is that an algorithm is essentially more abstract, and that it may be documented by many different instruction sequences, and just as well in terms of a plurality of other formalisms.

3. Given a program one may, or may not, claim the presence of an algorithm of which the program is an implementation. With the algorithmic background hypothesis (ABH) the claim is expressed that a program is in fact an implementation of an algorithm which was known to its designers in advance.

4. Assuming ABH for a program `X`, calling its underlying algorithm A, and in the absence of any documentation for A (which `X` supposedly implements), it is plausible to view `X` as the (best available) documentation of A. Now an incentive arises to identify A with `X`. Nevertheless in conceptual terms algorithms are not programs, in fact algorithms are more abstract than programs.

5. It is conventional to assume that a program complies with given specifications, say P, and that specifications express how a program (when being executed) may contribute to a system meeting given requirements, say R. Given specifications a programmer (software designer) may develop an algorithm and provide actual documentation for it. Subsequently a program may be developed which implements the algorithm and in that manner complies with the specifications. It is

conventional to approach the question whether or not a program X implements a specification P directly, that is without contemplation of an algorithm A from which X may be thought to have been derived.

5.2 Defining Program Faults when Assuming the Algorithmic Background Hypothesis

An interesting consequence of adopting the concept of an algorithm, as defined above, as well as the resulting connection between algorithms and programs is that an informal definition of a program fault can be given, or rather that an informal fault pattern can be defined with some rigour.

In the following fault pattern, a fault f is considered together with a change g for it, and justification of the assumption that g is an adequate change for f is provided in an informal manner. Thus the following definition of program faults is based on an informal justification of a change in which the justification takes the form of an expert judgement.

Definition 5.2 (ACOJoC Fault; Fault with Algorithmic Compliance Oriented Justification of Change) *Let program* X *implement algorithm A. A fragment (subprogram or smaller part of the text) f of* X *is an ACO fault in* X *if it can be replaced by a change g, thus obtaining* $\mathtt{X}_{f/g}$ *so that according to some experienced programmers knowledgeable of A and its possible documentations* X *does not fully reflect the meaning of a plausible documentation of the relevant part of A, while* $\mathtt{X}_{f/g}$ *is reflects the meaning of A more faithfully. In this case g is called an ACOJoC change of* X*.*

In [47] it is stated that "We classify a patch as correct, if it is semantically equivalent to the developer-provided patch, based on a manual examination. This is consistent with the definition used in previous work [....[46]]. An incorrect patch is a patch that is not correct." This notion of change correctness fits with the idea of an ACOJoC fault.

The notion of a ACOJoC fault generalizes in an unproblematic manner to multi-faults, where it is natural to expect orthogonality, though in an informal setting

Definition 5.3 (ACOJoC Multi-Fault) *Let program* X *be a candidate implementation of algorithm A. A disjoint sequence of fragments (subprograms or smaller part of the text) $f_1, \ldots, f_n$ of* X *is an ACOJoC multi-fault in* X *if each of the f_i can be replaced by a change g_i, thus obtaining* $\mathtt{X}_{f/g}$

so that according to some experienced programmers knowledgeable of A and of its possible documentations known to these programmers $\mathtt{X}$ *does not fully reflect the meaning of a plausible documentation of the relevant part of A, while for each nonempty U and V with* $U \subsetneq V \subseteq \{1,\ldots,n\}$ *the result of simultaneous replacement* $\mathtt{X}^{\mathtt{V}}_{f/g}$ *reflects the meaning of A more faithfully than* $\mathtt{X}^{\mathtt{U}}_{f/g}$*. In this case* $g_1,\ldots,g_n$ *is called an ACOJoC change of* $\mathtt{X}$*.*

The idea of a multi-hunk fault can be adapted to the ACOJoC pattern as follows:

Definition 5.4 (ACOJoC Multi-Hunk Fault) *Let program* $\mathtt{X}$ *be a candidate implementation of algorithm A. A disjoint sequence of fragments (subprogram or smaller part of the text)* $f_1,\ldots,f_n$ *of* $\mathtt{X}$ *is an ACOJoC multi-hunk fault in* $\mathtt{X}$ *if each of the* f_i *can be replaced by a change* g_i*, thus obtaining* $\mathtt{y} \equiv \mathtt{X}_{\tilde{f}/\tilde{g}}$ *so that according to some experienced programmers knowledgeable of A and of its possible documentations known to these programmers the following three claims are justified: (i)* $\mathtt{X}$ *does not fully reflect the meaning of a plausible documentation of the relevant part of A, (ii)* $\mathtt{Y}$ *reflects the meaning of A more faithfully than* $\mathtt{X}$*, and (iii)* $f_1,\ldots,f_n$ *is a minimal collection candidate faults combining properties (i) and (ii).*

Informal proposition 5.1 *An ACOJoC fault need not be an ALR fault at the same time.*

Proof: An ACOJoC fault f of $\mathtt{X}$ may manifestly fail to implement an instruction sequence $\mathtt{Y}$ which supposedly documents the relevant part of an algorithm A which underlies $\mathtt{X}$. However it may just be the case that the documenting instruction sequence is overly specific and that the inconsistency between the documentation and f is irrelevant for the overal working of A and of $\mathtt{X}$. In such circumstances it cannot be plausibly claimed that f causes a failure in the sense of not implementing a specification P, because "by accident" it might just be the case that $\mathtt{X}$ happens to work in compliance with P. □

Definition 5.5 (ACOJoC/ALR Fault) *An ACOJoC/ALR fault of* $\mathtt{X}$ *w.r.t. specification P is a fragment f of* $\mathtt{X}$ *for which there exits a change g such that (i) g is an ACOJoC change for f and g is an ALR change for f w.r.t. P.*

I recall that an STJoC (single test justification of change) fault (of $\mathtt{X}$ w.r.t. P) is defined as a (candidate) fault f for which a change g is known such that for at least one test $\mathtt{X}$ fails on the test inputs while $\mathtt{X}_{f/g}$ succeeds on the same inputs. A practical definition of a fault arises by combining the the requirements of an ACOJoC fault and an STJoC fault w.r.t. the same proposed change.

Definition 5.6 (ACOJoC/STJoC Fault) *An ACOJoC/STJoC fault of* $\mathtt{X}$ *w.r.t. specification* P *is a fragment* f *of* $\mathtt{X}$ *for which there exits a change* g *such that (i)* g *is an ACOJoC change for* f *and (ii)* g *is an STJ change for* f *w.r.t.* P*.*

A different definition of fault arises by combining ACOJoC fault with RTJ (regression test justified fault).

Definition 5.7 (ACOJoC/RTJoC Fault) *An ACOJoC/RTJoC fault of* $\mathtt{X}$ *w.r.t. specification* P *and a test suite* $\alpha_1, \ldots, \alpha_n$*, is a fragment* f *of* $\mathtt{X}$ *for which there exits a change* g *such that (i)* g *is an ACOJoC change for* f *and (ii)* g *is an RTJoC change for* f *w.r.t.* P *and the test suite* $\alpha_1, \ldots, \alpha_n$*.*

Similar combinations may be made with other classes of fault with semantic justification of changes, for instance: ACOJoC/Laski fault, ACOJoC/MFJ fault, AHB/essential Laski fault, ACOJoC/essential MFJ fault, to mention only four options for such combinations. However, we have not found any interpretation of program fault in the literature which calls for another combination than ACOJoC/STJoC fault and ACOJoC/RTJoC fault as defined above, and a preliminary conclusion from that observation may be that other combinations are of insufficient practical value.

6 Specification Faults and Software Process Faults

In this Section I will discuss faults at a higher level of abstraction, in particular specification faults and software process flaws. Software process flaws take the place of requirements faults which I consider to be a less convincing notion.

6.1 Specification Faults

In the paper have made use of a specification P of a program, and in particular of an instruction sequence, without further clarification of what is

supposed to be specified: what outputs are generated on which inputs, including "how fast" and "how efficient with memory usage". This view corresponds with classical texts e.g [5]. I will refer to such specifications as extended functional (EF) specifications: functionality of the program (instruction sequence) optionally extended with specifications of speed, of memory usage, perhaps including bounds on code compactness (LLOC), possibly also extended with bounds on energy consumption, and in on any conceivable performance related criterion which may be applied to the program.

Some authors, however, claim that a specification determines for a software component `X` "how it works", and might prefer to refer to an EF-specification as a requirements specification instead. The understanding of software specification as being informative about the how rather than the what of a program is mentioned as the second option in [51], and is also mentioned in [32]. If, however one understands a specification as a description of how a program works, the notion of a failure of compliance with the specification acquires a different meaning and becomes detached from testing. In that case a failure of compliance may be caused by an ACOJoC failure rather than by an ALR failure. Therefore in the paper it is assumed that a specification P is in fact a functional specification or more generally an EF-specification.

I prefer to use requirements for another purpose: to express what the system $S[\mathtt{X}]$ of which `X` is an intended component is expected (i.e. required) to achieve thanks to the contribution of `X`. In order to avoid confusion with other interpretations of the notion of requirements I will speak of COSC requirements (COSC for contribution of software component) on the software component `X`.

COSC requirements are specific for a context in which a program is embedded and in which its controlling ability comes to expression. Having available the notion of a COSC- equirement, the notion of a fault in an EF-specification P can be defined as follows:

Definition 6.1 (Specification Fault) *A fragment f of a specification P is a fault in P w.r.t. COSC requirements R if (i) it can be demonstrated (for instance by way of a test with a prototype implementation of P) that a system compliant with P will not meet the requirements in R (in an intended execution environment) while (ii) there is a change g for f so that a program compliant with the adapted specification $P_{g/f}$ is demonstrably compliant with R in a wider range of conditions.*

Definition 6.4 allows many variations connected with the various forms of justification for the adequacy of a proposed change. Just as for program faults a spectrum of options for precise notions of a specification fault arises, working out the details of this matter is left for future work.

In the presence of a specification fault in P which can be improved to P' w.r.t. COSC requirements R a new kind of instruction sequence fault arises.

Definition 6.2 (Quasi-Phantom Fault) *A fragment f of an instruction sequence* $\mathtt{X}$ *is a quasi-phantom fault with change g if (i)* $\mathtt{X}$ *complies with P, (ii)* $\mathtt{X}$ *does not comply with P', and (iii)* $\mathtt{X}_{f/g}$ *complies with P'.*

For COSC requirements I won't speak of faults but it ma be the case that at some stage during design and software production, or after delivery, an improvement R' is deemed necessary for R. In the eyes of those who favour R' over R, the consequence of this preference may be that yet another fault pattern emerges.

Definition 6.3 (Phantom Specification Fault) *A fragment f of a specification P is a fault in P w.r.t. COSC requirements R and its known improvement R' if (i) compliance with P guarantees for an instruction sequence* $\mathtt{X}$ *that it will meet requirements R, (ii) it can be demonstrated (for instance by way of a test with a prototype implementation of P) that a system compliant with P will not meet the requirements in R' (in an intended execution environment) while (iii) there is a change g for f so that a program compliant with the adapted specification $P_{g/f}$ is demonstrably compliant with R' in a wider range of conditions.*

If a specification fault is in fact a phantom specification fault then a corresponding quasi-phantom fault is a phantom fault. Again it is assumed that R' improves upon R. And it is assumed that P' improves P w.r.t. R' by eliminating a phantom specification fault. Under these conditions the pattern of a phantom fault emerges.

Definition 6.4 (Phantom Fault) *Assuming that P' results from P by elimination of a phantom fault w.r.t. COSC requirements R and known improvement$'$ thereof: a fragment f of an instruction sequence* $\mathtt{X}$ *is a phantom fault with change g if (i)* $\mathtt{X}$ *complies with P, (ii)* $\mathtt{X}$ *does not comply with P', and (iii)* $\mathtt{X}_{f/g}$ *complies with P'.*

6.2 Software Process Flaws

It is hardly plausible to define a requirements fault, by pointing to a fragment f of it (i.e. of a textual requirements specification) and suggesting a potential improvement of g of that fragment because there is no yardstick available against which to justify the conviction that the change from f to g within R constitutes an improvement of the requirements. I hold that it is unrealistic if not impossible to consider the quality of requirements in isolation, that is without a wider context of system design. In [9, 10] it is proposed to speak of a software process flaw if a software process has not been carried out in accordance with the operational rules that have been set out for the software development method at hand.

A software process flaw need not result in the creation of a program containing a fault, or in the preparatory creation of a specification for a program which contains a fault as defined in Paragraph 6.1. However, the result of a flawed software process may be a system that does not work. In particular if the software process happens to be aiming towards the solution of a problem for which no adequate software solution exists (a state of affairs which may not have been detected during the development process), it is not necessarily the case that the resulting software is faulty. More precisely, the resulting may well but need not contain MFJ faults while it cannot contain any Laski faults. The very notion of correct software implicitly depends on assumptions about the principled existence of software solutions t te problem at hand.

For instance one may imagine that a car model offers too few sensors, or not quite the right sensors, to allow the writing of an adequate software component for automated parking. Suppose that the software process for such a software component starts out in a state where it has not yet been recognized that the mission is impossible. In that case it is irrelevant to think of faults in the resulting software (if any software happens to be delivered) be it ALR faults or ACOJoC faults, or some variation of these. Instead the problem may lie somewhere else, for instance in a software process failing to produce a warning to the software engineers that a software component is supposed to contribute to the functionality of a system in an implausible manner.

6.2.1 An Example of a (Potential) Software Process Flaw

Much has been written about what might have caused the problems with the Boeing 737 MCAS configuration which are deemed to have been implied in two successive disastrous crashes, the first one in 2018 and the second one in 2019.

In particular in the popular literature there have been many suggestions that the software was at fault, for instance by not taking the outputs of both AOA sensors into account in circumstances where, according to the critics, that should have been done. The academic literature on the matter is very limited and to the best of my contains no single paper which focuses in an unprejudiced manner on the question whether or not the embedded software running the MCAS feature was at fault, and if so, what kind of fault that would be. Equally absent is any academic literature about the objectives of MCAS. The latter stands in sharp contras with the aerodynamics of airframes, and the material science involved, both of which have led to numerous scientific contributions through many years. So what I am writing is speculative and is based on limited information and may very well not adequately represent the state of affairs with the Boeing 737 MCAS configuration, as it was understood, in advance of the accidents. on purpose I am writing from the standpoint of someone who has no access to inside information.

Claim 6.1 *From the widely available information about the Boeing 737 MCAS algorithm affair one may extract examples of several candidate software process flaws, each of which may have occurred during the design of MCAS related programs and the occurrence of which might have contributed to overall system failure during one or both of the accidents. These examples have the virtue of clarifying the notion of a software process flaw, even if none of these candidate software process flaws can actually be held against the software process which actually took place during design and implementation of said MCAS algorithm.*

Claim 6.2 *Claim 6.1 is argued for in [10], where four so-called candidate software process flaws are listed in connection with (an uninformed abstraction from an external perspective of) MCAS (like) software.*

Claims 6.1 and 6.1 must both be read with care because I do not and cannot commit to the related but vastly different and much stronger claim that the following question has an affirmative answer.

Question 6.1 *Is it the case that at least one of the four candidate software process flaws as discussed in [10] actually featured in the MCAS software design and that fact may be considered as being among the causes of the catastrophic failures which have occurred.*

I am inclined to believe that Question 6.1 would be positively answered upon thorough investigation with prior odds, say 50%, which I consider to be so high that I expect others to have lower prior odds on the matter. Prior odds in subjective probability theory don't come with justifications. My perspective on how to apply subjective probabilistic reasoning has been outlined in detail in [7]. The following claim, I hold (with a high subjective probability, say 0.95), and its formulation is (on purpose and by design) independent of the critical Claim 6.1.

Claim 6.3 *Assuming one believes Claim 6.1, and more specifically Claim 6.2, and in addition one believes that Question 6.1 can be answered affirmatively, then one may have no compelling reasons grounded in publicly available information to believe that the (pre-disaster) MCAS software contained either Laski faults or MFJ faults. In other words from a theoretical perspective it is not obvious that these programs are (were) faulty.*

Claim 6.1 is merely an expression of ideas on software engineering and more specifically on requirements engineering. And also Claim 6.2 may be considered as (potentially) belonging to the theory of software engineering. Both claims are unrelated to the actual practice of software engineering with Boeing and its IT suppliers and to the various practices of certification regarding aviation software.

A summary of the argument for Claim 6.1 is as follows. In [10] it is argued that the Boeing 737 MCAS software component is supposed to provide a solution for a problem which is unlikely to have a fully satisfactory software solution in the presence of two AOA (angle of attack) sensors only. In [10] we disagree with the claim (implicitly suggested in [49]) that it is a software fault of MCAS to inspect only a single sensor in advance of triggering a stabilizer position intervention, and that a change must be applied which ensures that AOA sensor agreement is required as a precondition for an intervention. The mentioned software fault would find its cause in a specification fault which Boeing software engineers could (and should) have noticed. However, [10] draws a different conclusion: according to [10] there is indeed a design problem with the Boeing 737 Max, a problem which,

however, does not take the form of a program fault (and an underlying specification fault) in MCAS which, unaware of an AOA sensor mismatch which it could have detected (had the program fault not been present), has caused in the course of both tragic accidents that an unnecessary nose-down command has (repeatedly) been issued with adverse consequences. Instead the design problem stems from the invalid assumption that the trim wheel provides a generic solution to all stabilizer runaway-like problems, in the light of the fact that (unexpectedly) the use of the manual trim wheel turned out to be an insufficient technique to handle a (novel type of) stabilizer runaway condition (a condition now arising from an MCAS intervention based on a false positive reading of its AOA sensor).

6.2.2 Unconvincing Suspicion of a Specification Fault

With a drastic simplification one may imagine that the MCAS specification contains these assertions, P_l and P_r with `t` a threshold above which MCAS interventions are supposed to be triggered:

$$P_l \equiv \texttt{if}\ (\texttt{even\_cycle} \wedge \texttt{AOA.left} > \texttt{t})\ \texttt{then}\ \texttt{start\_intervention}$$
$$P_r \equiv \texttt{if}\ (\texttt{odd\_cycle} \wedge \texttt{AOA.right} > \texttt{t})\ \texttt{then}\ \texttt{start\_intervention}$$

This specification indicates that only a single AOA sensor will be inspected as a precondition for an intervention and that the assignment of that to one of the sensors role alternates with each cycle.The specification is to be implemented by means of an instruction sequence (i.e. the MCAS program), which is supposed to be running concurrently with other systems in control of the aircraft, in a multithread which is deterministically scheduled by way of strategic interleaving as described in [12, 13].

Now the mentioned fragment of the specifications might be considered faulty and instead the MCAS specifications might include the following changed assertion about intended MCAS behaviour:

$$P_{l,r} \equiv \texttt{if}\ (\texttt{AOA.left} > \texttt{t} \wedge \texttt{AOA.right} > \texttt{t})\ \texttt{then}\ \texttt{start\_intervention}$$

Regardless of whether or not the specification fragment $P_l \wedge P_r$ as mentioned is considered faulty, an which of both alternatives has been chosen, it is quite likely that the MCAS program provides a faithful implementation of it and for that reason is not at fault, at least not in this matter. more importantly, however, in [10] it is argued that there is no compelling reason to suspect a specification fault which can or should be (have been) eliminated by its replacement by the suggested change.

6.2.3 A Design Change as a Precondition for a Specification Fault

A conceivable design change for the aircraft is to introduce a third AOA sensor and to perform majority voting in advance of signalling that the AOA is too high and triggering a stabilizer repositioning intervention. In the presence of three AOA sensors, (or perhaps in the presence of two AOA sensors complemented with a synthetic airspeed sensor as is mentioned in [49]), there is increased credibility that a satisfactory MCAS component can be developed. Once a third AOA sensor (or any device which supports taking a statistically useful decision in cases of a sensor mismatch) is available, the suspected (i.e. candidate) specification fault as mentioned in Paragraph 6.2.2 advances to the status of a specification fault with a justified change as follows:

$$\begin{aligned} P_{l,r,3th} \equiv \texttt{if}\ ((&\texttt{AOA.left} > \texttt{t} \\ &\wedge AOA.right > t) \vee\ (AOA.left > t \wedge AOA.third > t) \\ &\vee (AOA.right > t \wedge AOA.third > t))\ then\ start\_intervention \end{aligned}$$

The argument of [10] is not based on any considerations regarding MCAS being safety critical, (such considerations being reserved for the handling by MCAS of false negatives rather than for the handling of false positives) and instead the argument focuses on the design of simulators and simulator training schemes as a part of the overall system engineering problem. In [10] it is argued that in the presence of two AOA sensors, which upon disagreement disable MCAS (in a proposed redesign), an unreasonable amount of training is needed for prospective Boeing 737 Max (new version) pilots to master being in control of the Boeing 737 Max (new) without MCAS, which is the same as piloting the Boeing 737 Max (grounded version) without MCAS, a task considered too problematic by Boeing designers and test pilots. The argument put forward for the latter is that if an observed AOA sensor disagreement disables MCAS (an MCAS change which has been announced by Boeing) a single sensor's failure (which occurs so often, and mainly in the initial stage of a flight, that it's handling must be trained) suffices to switch off MCAS for the rest of the flight so that pilots must be trained in a simulator to perform almost a full cycle in a Boeing 737 Max without the support of MCAS, a counterintuitive state of affairs given the claimed importance of MCAS.

6.2.4 Software Non-Delivery as a Positive Outcome of the Software Process

Suggestions are made in [10] concerning which software process flaws may have occurred and may be taken into consideration in order to explain how the delivery of a system (Boeing 737 Max control) could take place which turned out to be problematic, by containing a software component (MCAS) which supposedly contributes to the overall system in an (in hindsight) implausible manner. Awareness of software process flaws can turn the non-delivery of software at the end of a software process into the best way of delivering a result: non-delivery avoids the delivery of a problematic product. The software process becomes total (always producing a result) if a negative outcome (the project failed) is considered an acceptable outcome as well. The occurrence of one or more software process flaws may serve as a justification for non-delivery of a software component.

In [43] the Ariane 5 crash is discussed and it is indicated how each group of technical specialists may have their own way to diagnose and then locate (if not blame) the mistakes that were made during design, development, and production of the Ariane 5 missile. The different views are quite divergent and perhaps it is not possible to get any further than such a listing of options. In the Boeing 737 Max MCAS affair, however, it appears that the presence of a software fault, as an explanation for either of both disasters is implausible, while the occurrence of a software process flaw that went undetected may be suspected.

6.2.5 Presence of a Quasi-Phantom Fault in the MCAS Software, a Controversial Issue

To those who claim that the modified specification which asks for two AOA sensors being checked for exceeding a threshold rather than one is an improvement which meets the COSC requirement of adequately assisting the pilot(s) in controlling AOA during the flaps down flight phases, it is obvious that (i) a specification fault exists, and (ii) a corresponding quasi-phantom fault is present in the MCAS program.

The assessment of [10] however is different: said change does not provide an implementation of the mentioned COSC requirements on `X`. And for that reason it is unwarranted to claim the existence of a quasi-phantom fault in MCAS. Moreover, given the hardware configuration (2 AOA sensors) the COSC requirements cannot be met, and no improved version of

the requirements can be captured and therefore no specification P can be designed which guarantees of `X` that it implements R, so that no phantom fault is present either in MCAS software or in the MCAS specification.

Moreover, according to [10] the fact that an implementation MCAS was delivered during the software production process while no satisfactory solution was likely to be found may be explained by the occurrence of one or more software process flaws during the software process at hand and four options for such flaws (called candidate software process flaws) are indicated. Following [10] the MCAS program features no fault, no quasi-phantom fault, and no phantom fault, and its specification features no fault and features no phantom fault. Still MCAS does not satisfy the COSC requirement. It is, however, a defect of the COSC requirement that it does not admit a satisfactory implementation.

6.2.6 On the Role of Defeasible Claims in Theoretical Work

As we have noticed the theoretical literature about program faults is quite limited. This fact is the more remarkable given the extensive literature on program correctness. As it stands "program fault" is a practical notion and writing about it in a theoretical context constitutes a confrontation with practice, just as much as it may be a confirmation of practice in some cases.

In particular there is the risk of putting on paper claims which may eventually turn out to be wrong. This holds for the claims listed in (sub)section 6.2 but it would also hold for the following conceivable claims which I am not actually proposing: (i) Laski faults occur in practice (there is no evidence of this), (ii) MFJ faults occur in practice (I see no evidence either), (iii) STJoC faults occur in practice (again I see no evidence).

Of the fault patterns mentioned in the paper I claim that RTJoC faults occur in practice, and so do ACOJoC faults and the intersection of both fault patterns: ACOJoC/RTJoC faults. I do not claim to have found, in this paper, a fault pattern which faithfully represents the intuitions of fault as used by programmers in practice. The theoretical notions which have been contemplated above, perhaps with the exception of ACOJoC/RTJoC faults, are merely rough approximations of that intuitive of notion of fault.

In particular if one or more of the claims I have made in Subsection 6.2.1 turn out to be defeasible and are (in the future) proven to be wrong that fact constitutes progress. Assertions about the role of fault patterns in cases studies or in practice at large do not take the form of mathematical theorems. I expect that no (theoretical) work on software faults with potential

practical implications will consist of pure mathematics only. In my perception it is not the case that theory consists of 100 % true mathematics only, there is always a lot of interpretation around and such interpretations may turn out to be mistaken, a state of affairs which subsequently may be brought to light as a part of the development of the literature. I hold that theoretical work in computer science includes defeasible propositions and claims. And therefore it is plausible and to be expected that theoretical work sometimes leads to disagreement. It is well-known that for disagreement there is enormous scope in philosophy and theoretical economics, each of which have room for defeasible claims. I hold that similarly potentially wrong claims have a considerable place in informatics as well.

Stated differently: in my view a theorist runs a real risk of getting things wrong. My work is not aiming at bringing the risk of getting things wrong back to zero. It is about getting theoretical analysis of faults and failures forward, which invariably may come with steps which in hindsight may turn out to have been problematic from a methodological point of view. This happens in economics, in physics, in philosophy, and so on. I oppose the very idea that theoretical computer science equals the pure (and risk of failure avoiding) mathematics based on a package of classical and formal definitions.

Finally I may illustrate my viewpoint on the relevance of defeasible claims in informatics with a famous claim made by E. W. Dijkstra in 1970:

Claim 6.4 *Program testing can be used to show the presence of bugs, but never to show their absence.*

Now Claim 6.4 is wrong in case of a program working on a finite domain which admits exhaustive testing. And in practice all domains are finite. So the claim is defeasible, but the very defeasibility of the claim lies uncovers its strength. Claim 6.4 turns a judgement about asymptotic computational complexity, into a qualitative judgement about software engineering at large. And now it may be turned around: software engineering takes place in a world "where Claim 6.4 holds without any doubt". It is an axiom of software engineering which can be successfully formulated without making any attempt to develop an axiomatic framework for software engineering. Moreover Claim 6.4 strongly suggests that program testing centers on bugs, a perception which constitutes a bias on testing which one need not accept. This suggestion, however, as Dijkstra intended to achieve, helps a programmer to focus the mind on alternative ways of obtaining knowledge about program behaviour.

7 Concluding Remarks

In [24] it is argued that both program testing and program verification are to be understood as forms of defeasible reasoning. A similar position is put forward in [27]. Adopting defeasible reasoning as a basis for program verification, it is plausible that change justification is just as much a matter of defeasible reasoning, an idea which makes the difference between formal change justification and ACO based change justification less significant.

Next, I will mention some uses of fault, error, and failure in the software engineering literature. The classification of software faults of [30] seems to aim at a subdivision of the class of STJoC faults in two classes, Bohrbugs (which reproducibly cause failures) and Mandelbugs (which may or may not cause a failure depending on one or more other system components which lie(s) outside the control of a software tester), with Heisenbugs (an attempt to observe a failure caused by the flaw may suffice tho prevent a failure caused by the fault at hand from occurring) as a subclass of the Mandelbugs. Faults may percolate through a formalized life-cycle, or in the wording of [33] a defect cycle. In the proposed life-cycle it is for instance an explicit step to accept a candidate fault for repair, or alternative to remove it from the bag of potential (candidate) faults. The terminology of Bohrbugs, Heisenbugs and Mandelbugs seems not to be in use in recent literature anymore.

In [1] a failure is called an "incorrect behaviour". Further [1] assumes that program defects can be localized, which suggests that according to [1] "program defect" and "program fault coincide". In [31] the term error is considered "a problematic term used in different ways in different standards". According to [53] an error is a disguised failure, i.e. a state which is reached during a computation which should not have been reached but with the additional property that only an inspection of data internal to the computation may reveal the problematic state of affairs. In [41] soft errors, such as communication problems caused by radiation, are viewed as potential causes of logical failures, that is incorrect evaluation of Boolean values.

The notion of a software failure occurs in [40] where a software failure is meant to be a system failure caused by a software fault. Because faults are defined in terms of causation of failures, defining a software failure in this manner is quite indirect, but it seems to be the only way to define software failures. In [42] the remark is made that "unfortunately there is no particular definition of what a software fault is". In addition it is claimed in [42] that a definition of a software fault must make that notion quantifiable,

that is both the number of faults in a program and the size of each fault must admit objective measurement. Working in the context of instruction sequences makes the latter objective perhaps more easily achievable. [50] states that "(a)ssuming the bugs users report occur in a software product that really is in error, ...". I assume that in [50] "bug" equals fault and that users report failures rather than bugs, the product being faulty rather than being in error. In terms of the terminology that I am using this statement would translate into: "(a)ssuming that the failures users report in a program are non-phantom, and assuming that a fault causes such a failure, ...". In [22] an error (fault, failure) which needs to be repaired by changing the requirements is called a phantom error. With the latter terminology it is plausible to say that a requirements fault causes a phantom failure.

In [55] one finds detailed information on fault interference and fault localization in multi-fault programs, without any indication of a definition of faults. For the extensive survey of fault localization techniques in [52] it suffices to know that a fault is the underlying cause of an error, which itself is a precondition for a fault, without further explanation of causation in this context. In [35] (p9), however it is argued that "Whether faults cause failures is important, but strongly depends on what types of failures one is interested in." The latter is hard to reconcile with the idea that causation of failures is a defining criterion for a fault. The literature on testing is formidable, see [38] for the state of affairs up to 2010. Most testing papers view testing as a technique that is useful, if not necessary, in the perpetual battle against software faults. The literature which addresses faults as the primary topic is much less extensive than about testing, however, and is often written in terms of how to process the results of testing.

The notion of an ALR fault is essentially informal because it does not explain in detail what is meant by causation. Causation may be approximated form above (counting too many conditions as causes) and from below (accepting too few conditions as causes). Many different notions of fault stem from different interpretations of causation. To the best of my knowledge notions like program fault and program error have not been dealt with in the theoretical literature, while program failures have been studied widely, though not with the objective to trace failures back to errors and faults. In [45] a survey is presented of classification systems for software faults. None of these classifications suggests definitions of program faults with formal change justification.

Repairing a software fault by means of a change, also called software

fault elimination, is not the only way to go ahead when software faults are expected. Instead working towards software fault tolerance is an option. For instance one may create three programs each implementing specification P, written by independent teams. Then outputs may be determined by way of majority voting. In this manner the impact of a single fault in one of the implementations may be reduced. A comparison between software fault elimination and the use of n-programming based software fault tolerance can be found in [48].

7.1 Open Issues

As it stands it is hard to determine which of the fault patterns that have been defined in the paper have practical relevance. Orthogonal ACOJoC multi-faults are ubiquitous. Unclear is in how many applications of the STJoC fault pattern or of the RTJoC fault pattern can be found in practice. MFJ multi-faults are probably quite rare. The mathematics of MFJ multi-faults requires further attention, for instance it may be asked under which conditions is the union of two MFJ multi-faults again an MFJ multi-fault. Developing a theory of specification faults is an open theme altogether so it seems. From a methodological perspective detection and elimination of specification faults is essential, and the idea that specifications are designed without faults is just as implausible (or impractical) as the conception of faultless software in all but the most formalistic approaches to software design and development.

Acknowledgement. I profited from illuminating discussions with Mark Burgess on software faults and system failures, and I acknowledge the contribution of a reviewer whose remarks led to Proposition 3.2 and its subsequent generalization, and to a more explicit form of the discussion of software process flaws in Subsection 6.2.1.

References

[1] Z.A. Alzamil. Application of Redundant Computation in Program Debugging. *The Journal of Systems and Software* 81, 2024–2033, 2008. doi:10.1016/j.jss.2008.02.024.

[2] P. Ammann, J. Offutt. *Introduction to Software Testing. Cambridge University Press*, 2008. doi:10.1017/CBO9780511809163.

[3] 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.

[4] 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.

[5] R. Balzer, N. Goldman. Principles of Good Software Specification and Their Implications for Specification Language. *AFIPS'81*, 393-400, 1981. doi:10.1145/1500412.1500468.

[6] 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/.

[7] 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.

[8] J.A. Bergstra, I. Bethke. On the Contribution of Backward Jumps to Instruction Sequence Expressiveness. *Theory of Computing Systems* 50 (4), 706–720, 2012. doi:10.1007/s00224-011-9376-x.

[9] J.A. Bergstra, M. Burgess. A Promise Theoretic Account of the Boeing 737 Max MCAS Algorithm Affair. 2019. arxiv:2001.01543v1.

[10] J.A. Bergstra, M. Burgess. Candidate Software Process Flaws for the Boeing 737 Max MCAS Algorithm and a Risk for a Proposed Upgrade. 2020. arxiv:2001.05690v1.

[11] 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.

[12] 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.

[13] J.A. Bergstra, C.A. Middelburg. Distributed Strategic Interleaving with Load Balancing. *Future Generation Computer Systems* 24 (6), 530–548, 2008. doi:10.1016/j.future.2007.08.001.

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

[15] 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.

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

[17] 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.

[18] 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

[19] J.A. Bergstra, C.A. Middelburg. Program Algebra for Turing-Machine Programs. *Scientific Annals of Computer Science* 29 (2), 113–139, 2019. doi:10.7561/sacs.2019.2.113.

[20] J.A. Bergstra, A. Ponse. Execution Architectures for Program Algebra. *Journal of Applied Logic* 5 (1), 170–192, 2004. doi:10.1016/j.jal.2005.10.013.

[21] A. Bertolino. The (Im)Maturity Level of Software Testing. *ACM SIGSOFT Software Engineering Notes*, 29 (5), 1–4, 2004. doi:10.1145/1022494.1022540.

[22] I.B. Bourdonov, A.S. Kossatschev, V.V. Kulianin. Formalization of Test Experiments. *Programming and Computer Software* 33 (5), 239-260, 2007. doi:10.1134/s0361768807050015.

[23] M. Christakis, M. Heizmann, M.N. Mansur, C. Schilling, V. Wüstholz. Semantic Fault Localization and Suspiciousness Ranking. In T. Vojnar, L. Zhang (Eds.), *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2019)*, *Lecture Notes in Computer Science* 11427, 226–243, 2019. doi:10.1007/978-3-030-17462-0_13.

[24] T.R. Colburn. Program Verification, Defeasible Reasoning, and Two Views of Computer Science. *Minds and machines* 1, 97–116, 1991. doi:10.1007/978-94-011-1793-7_17.

[25] W. Ghardallou, A. Mili, N. Diallo. Relative Correctness: A Bridge Between Testing and Proving. In M. Ghazel, M. Jmaiel (Eds.), *Proceedings of the 10th Workshop on Verification and Evaluation of Computer and Communication System (VECoS 2016)*, *CEUR Workshop Proceedings* 1689, 141-156, 2016.

[26] M. Dyer. A Formal Approach to Software Errors Removal. *The Journal of Systems and Software* 7, 109-114, 1987. doi:10.1016/0164-1212(87)90015-x.

[27] J.H. Feltzer. Philosophical Aspects of Program Verification. *Minds and Machines* 1, 197-216, 1991. doi:10.1007/978-94-011-1793-7_18.

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

[29] R. Gopinah, C. Jensen, A. Groce. Mutations: How Close Are They to Real Faults? *25th International Symposium on Software Reliability Engineering*, 2014. doi:10.1109/ISSRE.2014.40.

[30] M. Grottke, K.S. Tivedi. A Classification of Software Faults. *16th IEEE Symposium on Software Reliability Engineering*, 19–20, 2005.

[31] L. Hatton. Programming Technology, Reliability, Safety and Measurement. *Computing and Control Engineering Journal* 9 (1), 23–27, 1998. doi:10.1049/cce:19980105.

[32] J.J. Horning. Program Specification: Issues and Observations. In J. Staunstrup (Ed.), *Program Specification (ProgSpec 1981)*, *Lecture Notes in Computer Science* 134. 5–24, 2005. doi:10.1007/3-540-11490-4_2.

[33] K. Hewett, P. Kijsanayothin. On Modeling Software Defect Repair Time. *Empirical Software Engineering* 14 (2), 165–186, 2009. doi:10.1007/s10664-008-9064-x.

[34] B. R. Kidwell. *MiSFIT: Mining Software Fault Information and Types*. University of Kentucky, Theses and Dissertations–Computer Science. 33, 2015. https://uknowledge.uky.edu/cs_etds/33.

[35] E. van der Kouwe.*Improving Software Fault Injection*. Ph. D. Thesis, Vrije Universiteit Amsterdam, 2016. https://www.cs.vu.nl/~ast/Theses/kouwe-thesis.pdf.

[36] 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.

[37] 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.

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

[39] 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.

[40] A. Mockus, D.M. Weiss. Predicting Risk of Software Changes. *Bell Labs Technical Journal* 5 (2), 169–180, 2000. doi:10.1002/bltj.2229.

[41] S. Mukherjee, J. Emer, S.R. Reinhardt. The Soft Error Problem: An Architectural perspective. *11th International Symposium on High-Performance Computer Architecture*, 2005. doi:10.1109/HPCA.2005.37.

[42] J.C. Munson, A.P. Nikora, J.S. Sherif. Software Faults: A Quantifyable Definition. *Advances in Engineering Software* 37 (5), 327–333, 2005. doi:10.1016/j.advengsoft.2005.07.003.

[43] B. Nuseibeh. Ariane 5: Who Dunnit? *IEEE Software* 14 (3), 15–16, 1997. doi:10.1109/ms.1997.589224.

[44] A.J. Offut, J.H. Hayes. A Semantic Model of Program Faults. *ACM SIGSOFT Software Engineering Notes* 21 (3), 195–200, 1996. doi:10.1145/226295.226317.

[45] J. Ploski, M. Rohr, P. Schwenkenberg, W. Hasselbring. Research Issues in Software Fault Categorization. *ACM Software Engineering Notes* 32 (6), 1–8, 2007. doi:10.1145/1317471.1317478.

[46] Z. Qi, F. Long, S. Achour, M. Rinard. An Analysis of Patch Plausibility and Correctness for Generate-and-Validate Patch Generation Systems *Proceedings of the 2015 International Symposium on Software Testing and Analysis (SSTA 2015)*, 24–36, 2015. doi:10.1145/2771783.2771791.

[47] S. Saha, R.K. Saha, M.K. Prasad. Harnessing Evolution for Multi-Hunk Program Repair. *IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019. doi:10.1109/ICSE.2019.00020.

[48] J. Shimeall, N.G. Leveson. An Empirical Comparison of Software-Fault Tolerance Elimination and Software Fault Elimination. *IEEE Transactions on Software Engineering* 17 (2), 173–183, 1991. doi:10.1109/32.67598.

[49] The House Committee on Transportation and Infrastructure: *Boeing 737 Max Aircraft: Costs, Consequences, and Lessons from Its Design, Development, and Certification : Preliminary Investigative Findings.* (2020). https://www.govinfo.gov/app/details/GOVPUB-Y4_T68_2-fb0f3812fefe3515ebcf3f4170fce64b

[50] J.A. Whittaker. What is Software Testing? And Why is it so Hard? *IEEE Software* 17 (1), 70–79, 2000. doi:10.1109/52.819971.

[51] J.M. Wing. Program Specification. *Encyclopaedia of Computer Science* 1454–1458, 2003.

[52] 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.

[53] 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.

[54] 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.

[55] 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.