Scientific Annals of Computer Science vol. 32 (2), 2022, pp. 183–209
doi: 10.7561/SACS.2022.2.183

Four Notions of Fault for Program Specifications

Jan A. Bergstra[1]

Abstract

Four notions of fault are proposed for program specifications each inspired by notions of fault for programs: symptomatic failure resolution fault, Laski fault, MFJ fault and regression test justification of change fault (RTJoC fault). Examples are provided in terms of the PGA style theory of instruction sequences. Each of the notions of fault is based on the contrast between technical specification and requirements specification. The latter contrast is discussed in detail.

Keywords: technical specification, requirements specification, Laski fault, MFJ fault.

1 Introduction

The notion of a program fault has until now received far less attention from theorists than the somehow related notion of program correctness. The relation between faults and correctness is non-obvious, however, and both notions are quite sensitive to particularities of the respective definitions. It is plausible to assume that the definitions of correctness and fault guarantee that a correct program contains no faults. Conversely, however, there is no basis for the assumption that a defective (i.e. incorrect) program will contain one or more faults.

Following [1, 2, 16] a fault in a program is a static property of it which qualifies as a cause of a failure. In the terminology of [5] an ALR fault in

This work is licensed under the Creative Commons Attribution-NoDerivatives 4.0 International License

[1]Informatics Institute, University of Amsterdam, Science Park 904, 1098 XH, Amsterdam, The Netherlands, Email: j.a.bergstra@uva.nl, janaldertb@gmail.com

a program X consists of (i) a failure for X, (ii) a program fragment of X combined with (iii) a change for said fragment, so that when the change is effectuated (iv) the particular failure will not occur anymore, and (v) the change can be understood as an improvement of the program. It is left unspecified when a change may be considered an improvement.

Laski [17] made a first proposal on how to formalise a notion of improvement that underlies ALR faults, thereby giving rise to a reasonably well-defined notion of fault, missing some quantitive information only, which is referred to as a Laski fault in [5]. Subsequently in [18] one finds a different formalisation of program faults (see also [13]), which, again following [5] is best understood as a second type of fault in a typology of faults. In [5] a third type of fault is described which is cast in terms of testing and regression testing only. The three forms of faults just mentioned differ in the form of justification which is given for a change, and in particular in the conceptualisation of a notion of the improvement brought upon by effecting a change.

A change is supposed to prevent a certain failure from taking place, while at the same time a change must preferably not introduce new failures, that is failures on inputs different from the input on which a failure was resolved by the program being repaired. The idea for a Laski fault is that a corresponding change creates a correct program, the idea for an MFJ fault is that the changed program is still working correctly on inputs where it was working correctly before the change. The latter idea can be relaxed by requiring that a given regression test suite which a program passed in advance of the change is still passed after the change has been made. The three strands of fault thus obtained merely constitute a selection from a larger variety of such notions. For a survey of such options I mention [5] and the follow-up paper [6].

Technical definitions for various kinds of faults depend on actual program notations. In [5] I made use of instruction sequences in the notation of the program algebra PGA of [8]. These notations are highly simplified and are suitable for theoretical work only. Said notions of fault each depend on a notion of failure which itself depends on the availability of a technical specification which determines which behaviours of a program are adequate and which behaviours fail.

1.1 Four Notions of Fault in a Technical Specification

The objective of this paper is to discuss notions of fault for (technical) specifications of programs rather than for programs proper. In fact we will propose four such notions: (i) symptomatic failure resolution fault as a simplest notion of specification fault, (ii) Laski fault and (iii) MFJ fault represent different refinements of the notion of a symptomatic failure resolution fault, and (iv) regression test justification of change fault relaxes the constraints imposed by the notion of an MJF fault to weaker constraints which can be checked by means of testing only. We start with a general discussion of the notion of a specification fault.

1.2 Specification Fault: General Conditions for the Existence of Such Faults

For a specification to be faulty some external criterion of validity is needed. If a program specification constitutes the only available knowledge about what the program must achieve it cannot be defective other than by being non-implementable, unreadable, too lengthy etc. The computer science literature provides little or no discussion of specification faults.

I will adopt the idea that besides a technical specification for a program there may also be a set of requirements (requirements specification) which provides additional information of what is expected from a program. In the presence of a perceived mismatch between technical specification and requirements the suggestion that the technical specification is defective rather than that the requirements specification is defective can only be based on a conceptual asymmetry which assigns the requirements specification a higher status than the technical specification.

At the same time it must be explained why the requirements specification would not be included as a part of the technical specification because when doing so a defective specification would simply turn into an inconsistent specification, which is an easy notion to imagine. We are led to the following thought experiment: a context $C[-]$ is given which, when provided with a software component (program) P, which is yet to be designed, will result in an intended system $C[P]$ for which requirements specification S_{req} is known.

Now a specification S_{spec} is designed with two equally important objectives in mind:

(i) an implementation P of S_{spec} will be such that $C[P]$ satisfies S_{req};

(ii) S_{spec} is so general that its implementations can and might be used in other contexts just as well, and for that reason specific use cases such as the requirements S_{req} on $C[P]$ are to be avoided in the specification S_{spec}.

1.3 Technical Specifications Versus Requirements Specifications

I will assume that a program P implements a fairly precise (if not formal) technical specification S_{tech}. A technical specification tells what the program is supposed to achieve (compute) in a general terms, i.e. without any particular context or application of the program in mind. In view of generality a technical specification will not involve so-called use cases. If a program is supposed to compute a function then a logical description of the graph of that function may serve as a technical specification for the program, where it is assumed that the logical specification merely specifies said graph and is not biased towards any specific application or towards one or more specific use cases.

We will assume that S_{tech} has been designed with a specific application in mind. More specifically a context $C[-]$ is known such that P will be used in a system of the form $C[P]$ for which a specification is given. The latter specification, referred to as S_{req} explains how $C[P]$ is supposed behave and, w.r.t. P it has the status of a requirements specification. Typically a requirements specification may involve use cases, i.e. particular instances of desired behaviour of $C[P]$. A requirements specification for P may provide use cases for a plurality of contexts of use of P: $C_1[P], \ldots, C_k[P]$.

One may prefer to consider the requirement that $C[P]$ satisfies S_{req} to constitute a technical specification of P, therewith removing the distinction between specifications and requirements in this case. We will not adopt such a convention with the argument that $C[P]$ $\underline{\text{sat}}$ S_{req} as a property of P is not sufficiently application independent to qualify as a (technical) specification of P. Both technical specifications and requirements specifications may involve functional aspects as well as non-functional aspects.

The contrast between technical specification and requirements specification resembles the contrast made between language constraint and application constraint in [14]. A similar contrast is mentioned in [15] between local specification and system specification.

1.4 Informality for Requirements Specifications

Besides looking at a higher level of aggregation in a system, in comparison to a technical specification of a system component, a requirements specification may also sometimes use informal language. Reasons for allowing informality at the level of requirements may vary. Three main motives for tolerance of informality may be distinguished in a specific situation. Specifying these motives necessitates the introduction of a project supervisor who is supposed to be satisfied in the end with the system as developed.

(i) Informal language is deemed more readable and more concise and for that reason more effective in transmitting intentions of the project supervisor to the engineers at work.

(ii) By expecting readers to select most plausible interpretations the plurality of different interpretations of an informal specification may be limited to a lesser variety and the project supervisor may expect to be satisfied with (the consequences of) each of those readings.

(iii) The project supervisor has some additional constraints in mind which have not yet been taken into account in the requirements specification and informality of the requirements introduces the flexibility to take said constraints into account during the development of a technical specification for component P.

Below we will work with the simplifying assumption that the requirements specification is formal and complete. Whenever the requirements specification is met, an unambiguous matter in view of formality of the requirements specification, the project supervisor is satisfied, a matter of completeness.

1.5 Technical Specification Faults I: The Case of Retrospective Specifications

We consider the situation that a technical specification S_{tech} is given for a given or yet to be developed program P. If P has been developed while subsequently the specification has been written, we speak of a retrospective specification. The purpose of a retrospective specification is to clarify concisely the main properties of P. A retrospective specification may be used by a programmer who contemplates including P as a component in a system design.

A retrospective specification S_{tech} for P is defective if it is not the case that P $\underline{\text{sat}}$ S_{tech}. It is hard to imagine that a problem with a retrospective specification of a program causes a failure of that program proper, as without the specification being present the same dynamic behaviour would appear. In the case of retrospective specifications the ALR principle that a fault constitutes a repairable cause of a failure must be abandoned, so it seems. What remains is the idea that a fault constitutes a textual change of a specification which achieves two aspects: to do away with a certain deficiency in the specification while at the same time to lead to an improved specification. Unlike with program faults, however, the objective of specification is not so much to find a correct specification (taking the assertion "true" for a specification suffices for that), but to determine a sufficiently informative correct specification. The latter notion is rather informal. We conclude that the idea of a Laski fault, a change of a defective artefact upon which a correct artefact is obtained, does not generalise in a useful manner to retrospective specifications. More generally we hold that when contemplating notions of fault for retrospective specifications the ALR principle (a fault being a fragment of an artefact which can be understood as a cause of a failure) has to be abandoned and another intuition of fault must be brought to bearing. This argument motivates the following claim:

Claim 1.1 *The ALR principle that a fault constitutes a fragment of an artefact which can be understood as a cause of a failure cannot be used as a foundation for the development of notions of fault for retrospective program specifications.*

We intend to proceed along the lines of the ALR principle, i.e. to consider notions of fault which are inspired by the ALR principle. As a consequence we find that for retrospective specifications said endeavour is vacuous and our focus must be on prospective specifications instead. Returning to retrospective specifications, we propose as a second claim, that what we will call inverse ALR faults will play a role when contemplating faults.

Definition 1.1 *An inverse ALR fault is present in the retrospective specification S_{retro} of an artefact A if*

(i) the artefact does not satisfy S_{retro} (i.e. not A $\underline{\text{sat}}$ S_{retro});

(ii) a fragment of S_{retro} may be understood as a cause of the state of affairs mentioned under (i).

An inverse ALR fault blames the specification rather than the implementation for a defect consisting of a mismatch between both.

Claim 1.2 *Notions of fault for retrospective specifications of programs can be based on the idea of an inverse ALR fault.*

For the idea of an inverse ALR fault the notion of causation involved is non-obvious and merits further investigation.

1.6 Technical Specification Faults II: The Case of Prospective Specifications

It is plausible that S_{tech} has been designed with the requirement that upon developing an implementation P of S_{tech} it will be the case that $C[P]$ satisfies S_{req} in mind. In this scenario S_{req} is given in advance as well as the context $C[-]$ and subsequently S_{tech} is developed on the basis of these data as a (first) step towards the development of P. Developing P in turn is a necessary step towards the development of $C[P]$ which is the underlying engineering objective at hand. A complication which may be taken into account is that it is plausible for S_{req} to be less formal, i.e. to a lesser extent sound and complete than S_{tech}.

Now assuming that the objective of designing S_{tech} derives from the intention to develop P so that $C[P]$ $\underline{\text{sat}}$ S_{req} while insisting that S_{tech} is sufficiently abstract to qualify as a technical specification, a setting emerges where it makes sense to think in terms of the quality of S_{tech}.

It will be assumed that $C[-]$ works in such a manner that the functionality of $C[P]$ depends on the functionality of P and on no non-functional property of P and that if Q has the same functionality as P though with non-functional properties that improve those of P in some respect, the corresponding non-functional properties of $C[Q]$ may improve (in any case not degrade) on the corresponding non-functional properties of $C[P]$.

The question then arises to what extent the notion of a fault in a specification S_{tech} makes sense, as well as the related question if specifications without faults can still be defective. We will propose several notions of fault for a technical program specification S_{tech} w.r.t. a requirements specification of the form $C[-]$ $\underline{\text{sat}}$ S_{req} which is required for that same program.

The discussion will be illustrated below a very simple example presented in terms of the PGA style theory of instruction sequences. Instruction sequence notations as introduced in [8] are devoid of practical significance. Nevertheless the example allows illustration in some detail of the various

notions of fault understood as theoretical concepts rather than as notions ready made for use in a practical setting.

2 Specification Faults

In this section we will discuss specification faults leading up to the notion of a Laski fault for specifications.

2.1 Prospective Technical Specification as the Default for Specification

Unless stated explicitly a specification for a program is assumed to be a technical specification rather than a requirements specification. This default convention for the understanding of specification is in line with the widespread convention not to drop "requirements" from the phrase requirements specification. Said default is also in line with the practice to think in terms of specifications of software components, where a specification is understood to be independent of any actual or potential use of a software component. Nevertheless, for the sake of clarity of exposition, I will often use the phrase "technical specification" and not rely on the suggested default. In the title of the paper, however, the mentioned default is supposed to be taken into account.

Moreover, a program specification is supposed to be prospective by default, i.e. it has an independent status which might serve or have served as the starting point of program development, rather than that it has come about by way of reverse engineering from a given software component.

2.2 Quality Assessments for Technical Specifications Including Laski Faults

Below we list some quality assessment options for a technical specification S_{tech}, serving as a specification for a program P (yet to be developed) all w.r.t. a given requirements specification S_{req} which expresses behavioural constraints (on the same program) of the form $C[X]$ $\underline{\text{sat}}$ S_{req}:

1. A requirements specification S_{req} w.r.t. context $C[-]$ is feasible if for some program P, $C[P]$ $\underline{\text{sat}}$ S_{req}, otherwise the requirements specification is infeasible.

2. implementable $\iff$ an implementation exists (alternatives: consistent, not self-contradictory).

   More specifically: S_{tech} is implementable if there exists a program P such that P $\underline{\text{sat}}$ S_{tech}.

   Constructing a program that satisfies S_{spec} is also called program synthesis, and (non)synthesizable may be used as an alternative for (non)implementable. The later terminology can be traced back at least to Church [12].

3. requirements compatible $\iff$ an implementation P of S_{tech} exists such that $C[P]$ $\underline{\text{sat}}$ S_{req}.

4. sufficient $\iff$ implementable and moreover each implementation meets the requirements specification.

   More specifically: S_{tech} is sufficient if S_{tech} is implementable and if for each program P such that P $\underline{\text{sat}}$ S_{tech} it is the case that $C[P]$ $\underline{\text{sat}}$ S_{req}.

5. insufficient $\iff$ requirements compatible while not sufficient.

   More specifically: S_{tech} is insufficient if there are programs P and Q such that P $\underline{\text{sat}}$ S_{spec} and Q $\underline{\text{sat}}$ S_{spec} while $C[P]$ $\underline{\text{sat}}$ S_{req} and $\neg(C[Q]$ $\underline{\text{sat}}$ $S_{req})$.

6. wrong $\iff$ implementable while not requirements compatible.

   More specifically: S_{tech} is wrong if it is implementable while none of the implementations P of S_{tech} enjoys $C[P]$ $\underline{\text{sat}}$ S_{req}.

7. defective $\iff$ not sufficient.

8. Specification S_{tech} is Laski k-faulty $\iff$ S_{tech} is defective and there exists a textual change γ of S_{tech} with edit distance k or less that produces a specification $\gamma(S_{tech})$ which is sufficient w.r.t. the requirements specification S_{req} and context $C[-]$.

   More specifically: given requirements specification S_{req}, a context $C[-]$, and a specification S_{tech} a Laski k-fault in S_{tech} relative to S_{req} and $C[-]$ is a change γ (with edit length k or less), which, when applied to S_{tech}, produces an implementable specification $\gamma(S_{tech})$ such that all implementations P of $\gamma(S_{tech})$ enjoy $C[P]$ $\underline{\text{sat}}$ S_{req}.

The idea of a Laski fault is simple: the specification is defective (either not implementable or one of its implementations fails to meet the given requirements specification), and by applying a limited textual change a sufficient specification is obtained. We make use of the eponym Laski because it conforms the approach of Laski [17] to speak of a fault of a program in case of a failing program in combination with a corresponding change leading to a correct program. We notice that the notion of a Laski fault may apply just as well if the specification at hand has no implementation.

However an important distinction with the case of program faults arises. If a (deterministic) program contains a Laski fault it must fail on some input, and it will fail always on that same input, while upon the corresponding change having been applied failure on the same input will not take place. This state of affairs warrants viewing the fault as a cause of the failure.

In the case of a specification S_{req} with a Laski fault, however, the specification may well have one or more implementations P such that $C[P]$ $\underline{\text{sat}}$ S_{req}. Thus viewing the Laski fault in the specification S_{req} as a cause of failure is unwarranted. Rather the fault may be understood as an explanation of a failure of $C[P]$ on some input s for some implementation P of S_{tech}which happens to feature such a failure. Stated mor formally:

Proposition 2.1 *If a specification S_{tech} (relative to requirements S_{req} and context $C[-]$) is faulty and contains a Laski k-fault γ then either*

(i) the specification S_{tech} is wrong and γ may be understood as a cause of the presence of failures in $C[P]$ for whatever implementation P of S_{tech}, or otherwise

(ii) the specification S_{tech} is insufficient and γ may be understood as an explanation of the presence of one or more failures in the behaviour of $C[P]$ for one or more implementations P of S_{tech}.

The idea of an ALR fault as a fragment of a (program) text which causes a failure is replaced by what we call an E/C-ALR fault: a fragment of a (specification) text which for some of its implementations P explains the presence of a failure in the target system $C[P]$. The explanation indicates that S_{tech} allows too much freedom of implementation thereby introducing the risk of certain failures in the target system.

Definition 2.1 *An E/C-ALR fault ("explain instead of cause" ALR fault) in an artefact is present if the artefact contains a fragment which can be*

understood as the explanation of a failure which may arise but which need not arise (thereby not qualifying as a cause).

Claim 2.1 *A plurality of plausible notions of fault for prospective specifications of programs can be obtained as detailed instances the notion of an E/C-ALR fault.*

The notion of a Laski fault for a technical specification seems to be the first and most plausible idea on the matter of specification faults. A complication with the notion of a Laski fault is that spotting a Laski fault involves a mathematical proof that allows to take into account all implementations of the modified specification.

2.3 Generalising Laski Faults: MFJ Faults and RTJoC Faults

In the case of program faults two generalisations of Laski faults have been elaborated: an MJF fault is present if a so-called symptomatic failure can be removed (repaired) by applying a change which in addition is supposed to provide valid results whenever the original program did so. This generalisation of the notion of a Laski fault stems from [18] and has been named an MJF (for Mili, Frias and Jaoua) fault in [5].

A generalisation of the notion of an MJF fault has been outlined in [5] where instead of asking for an improvement of the modified program it is required that the modified program, besides performing better on the input for the symptomatic failure complies with a given regression test suite. The latter generalisation is referred to as a regression test justification of change fault (RTJoC fault) in [5].

Below we will define counterparts for MFJ faults and for RTJoC faults for technical specifications.

2.4 Proposal for the Notion of an MFJ Fault in a Technical Specification

We start with an auxiliary type of fault which is then modified into the definition of an MFJ fault for a technical specification.

Definition 2.2 *(Symptomatic failure resolution k-fault.) Given:*

- *a requirements specification S_{req} concerning a system $C[X]$ with program parameter X, and*

- *an implementable specification S_{tech} for programs X,*

a symptomatic failure resolution k-fault in S_{tech} relative to S_{req}, consists of

- *a change γ (with edit length k or less), which is applicable to S_{tech} (then obtaining $\gamma(S_{tech})$), and*
- *an element s (symptomatic failure case) of the input domain of systems of the form $C[X]$,*

where the following conditions are satisfied:

(i) *$\gamma(S_{tech})$ is implementable,*

(ii) *for some program P that satisfies (implements) S_{tech}, $C[P]$ fails on s w.r.t. S_{req},*

(iii) *for no program Q that implements $\gamma(S_{tech})$, $C[Q]$ fails on s w.r.t. S_{req}.*

If a coherent specification S_{tech} contains a symptomatic failure resolution k-fault then S_{tech} is either wrong or insufficient. Moreover the following observation is immediate:

Proposition 2.2 *Given a symptomatic failure resolution k-fault, with symptomatic failure on s and with change γ and an implementation P of S_{tech} for which $C[P]$ fails to comply with S_{req} on a particular input s then it must be the case that the computation of $C[P]$ on s makes use of one or more computation(s) of P (on various inputs say $t_1, \ldots, t_n$) which are not in conformance with $\gamma(S_{tech})$.*

Thus, the upgrade of S_{tech} to $\gamma(S_{tech})$ may be understood as a modification of the specification which blocks, and thereby removes as a cause of, the failing computation (of $C[P]$) on s, and ensures that a failure on the same input s will not occur for any implementation of $\gamma(S_{tech})$.

The notion of a symptomatic failure resolution fault allows a different wording of the definition of Laski faults for specifications.

Definition 2.3 *(Laski k-fault.) Given the setting of Definition 2.2 an additional (iv)-th condition is required besides (i), (ii), and (iii):*

(iv) *(sufficiency obtained) each implementation Q of $\gamma(S_{tech})$ it is the case that $C[P]$ $\underline{\text{sat}}$ S_{req}.*

Although the notion of a symptomatic failure resolution fault may seem convincing there is a serious problem with this notion: successive failure removals may undo the guarantees obtained before. This difficulty is not present in the case of Laski faults because upon having resolved a Laski fault no other faults remain by definition. In the latter observation also lies a weakness of the notion of a Laski fault: it is implausible to assume that an artefact (in this case a specification) involves a single fault only. In the case of programs MFJ faults were introduced to take the presence of a plurality of faults into account.

The idea of an MFJ fault involves two aspects: (a) symptom resolution: a change leading to a local improvement combined with (b) preservation of what works well already. We propose the following generalization of MFJ faults for programs to specifications.

Definition 2.4 *(MFJ k-fault.) Given the setting of Definition 2.2 an additional (iv)-th condition is required besides (i), (ii), and (iii):*

(iv) (preservation) for all inputs t for systems of the form of $C[X]$: if for each implementation Q of S_{tech} the computation of $C[Q]$ on t proceeds in conformance with S_{req}, then for each implementation Q' of $\gamma(S_{tech})$ the computation of $C[Q']$ proceeds in conformance with S_{req}.

A difficulty of both notions Laski fault and MFJ fault is the use of quantifiers over implementations of S_{spec} in the respective definitions. A similar difficulty arises with the definition of an MFJ program fault which involves quantification over all inputs.

2.5 Specification Faults Justified on the Basis of Regression Testing

For practical application such quantifiers pose a problem. In [5] a testing based alternative for the notion of an MFJ program fault is proposed which avoids such quantifiers. Describing a testing based alternative for the notion of an MJF fault in the case of specifications is more involved and allows a plurality of alternatives. Our proposal for a testing based notion of fault reads thus:

Definition 2.5 *(Regression test justification of change specification k-fault.) Given a requirements specification S_{req} concerning a generic program $C[X]$,*

with program parameter X, and an implementable but insufficient specification S_{tech} for the program parameter X, a change γ (with edit length k or less) is an RTJoC k-fault in S_{tech} relative to S_{req} if the conditions of a symptomatic failure resolution fault (see Definition 2.2) are met (with input s) and in addition the following items (iv),..,(x) with the listed properties are available:

(iv) a sequence $t_1, \ldots, t_n$ of test cases for programs X (i.e. that satisfy the static constraints of S_{tech}), such that for all cases the computation of P on t_i satisfies the specification S_{tech}, (these observations are understood as a justification of viewing P as a candidate implementation of S_{tech}),

(v) a sequence $s_1, \ldots, s_m$ (serving as a regression test suite), of test cases for systems of the form $C[X]$, such that (a) the computations of $C[P]$ on s_i $(1 \leq i \leq m)$ make use of subcomputations for P exclusively with inputs from $t_1, \ldots, t_n$, and (b) such that each of these computations (of $C[P]$) complies with S_{req},

(vi) a test case s (the symptomatic fault case) for systems of the form $C[X]$ such that the computation of $C[P]$ on s fails to comply with S_{req}, while each of the subcomputations of P of that computation has an input in $\{t_1, \ldots, t_n\}$.

(vii) a program P' serving as a candidate implementation of $\gamma(S_{tech})$, where P' has been developed from $\gamma(S_{tech})$ by a team of software engineers who were not involved in designing P from S_{tech} and who are unaware of S_{req} and of the context $C[-]$,

(viii) a set $\{r_1, \ldots, r_{n'}\}$ of test cases for programs X which extends $\{t_1, \ldots, t_n\}$ such that each of the computations of P' on r_i $(1 \leq i \leq n')$ complies with $\gamma(S_{tech})$,

(ix) so that each of the computations of $C[P']$ on $\{s_1, \ldots, s_m, s\}$ makes use of subcomputations of P' on inputs in $\{r_1, \ldots, r_{n'}\}$ only,

(x) and so that each of the computations of $C[P']$ on $s_1, \ldots, s_m$ is in conformance with S_{req} (the regression test is passed) and in addition the computation of $C[P']$ on s is in conformance with S_{req} (i.e. the failure of $C[P]$ on s, serving as the symptom of fault, has been repaired).

This regrettably elaborate definition merits various explanatory comments.

1. It is an implicit assumption that programs P and P' have a reasonable length. Given a specification S_{tech} and a set of test cases $T = T_{tetstForX}$ it will be doable to design P_T in such a manner that it complies with all test cases. However, the size of P_T will be proportional to the number of test cases in T and that is implausible. Thus P and P' are supposed to be "plausible" implementations of S_{tech} and $\gamma(S_{tech})$ which work for all relevant inputs and not just for inputs for P resp. P' which occur when computing $C[P]$ resp. $C[P']$ on the test set $\{s_1, \ldots, s_m\}$ and the primary symptom case s where $C[P]$ fails.

2. It is by $C[P']$ passing the regression test in addition to it working in conformance with S_{req} that justification is claimed for referring to γ as a fault in S_{tech} with a repair into $\gamma(S_{tech})$.

3. With the notations of Definition 2.5 we may replace S_{tech} by $\gamma(S_{tech})$ and replace P by P' write $s_{m+1} \equiv s$, thus obtaining $n+1$ test cases $s_1, \ldots, s_{n+1}$ for the generic program $C[X]$ and take $r_1, \ldots, r_{n'}$ for the sequence as in item (now for P') (ii), thus obtaining a setting from which another specification fault can be looked for and possibly be repaired by way of a change with k or less symbol changes. In this manner Definition 2.5 allows for the definition of a chain of successive faults with repairing changes.

4. Remarkably, defining a specification fault in the context where testing is the only way of obtaining information about program behaviour, and only such information may play a role in the definition, turns out to be significantly more involved than providing such a definition in case one is able to speak in term of programs semantics in a mathematical fashion as e.g. in Definition 2.3.

5. An advantage of Definition 2.5 over say Definition 2.4 comes from the observation that, assuming the availability of oracles for S_{tech}, $\gamma(S_{tech})$ and S_{req}, all conditions and assertions that occur in the definition admit straightforward checks.

6. It is implicitly assumed that testing is done on the basis of test cases that constitute a single input (or input stream) for a program. This rules out metamorphic testing where two or more related inputs are considered and a certain relation between the corresponding outputs is checked. In principle Definition 2.5 can be adapted in such a manner

as to admit metamorphic testing as well. We will not discuss the details of the latter.

3 Theory of Instruction Sequences

I will use inSeq as an abbreviation of instruction sequence, with the connotation of the instruction sequences occurring in the theory of Instruction sequences (also referred to as inSeq theory) as put forward in [8] and subsequent work. The series of examples in the next Section is phrased in terms of inSeq theory. The above descriptions of notions of specification fault involve both existential and universal quantifiers over programs. For these descriptions to be watertight in al cases it ought to be stated which classes of programs are meant when performing such quantification. Each of the inSeq notations from inSeq theory can be used for that purpose, and so can many other program notations.

3.1 Data and Control

The examples below make use of so-called single bit services (also called Boolean services) which were introduced in full detail in [10]. A brief introduction from first principles to the use of Booelan services in inSeq theory is provided in [11]. Simple examples of use of single bit services can be found in [4]. The application of an inSeq to inputs as contained in a service family is given by the apply operator from [9],

InSeq theory involves the following key elements:

(i) PGA style program algebra, with various inSeq notations including PGLC, as defined in [8], and instructions for structured programming (conditionals and a while loop). In the example the program P is assumed to be written in PGLC.

(ii) Threads, thread algebra, thread extraction, strategic interleaving.

(iii) Services, service families; instruction sequence processing operators which determine the interaction between inSeq's and services or service families.

(iv) Required interfaces of inSeq's and threads; provided interface for services and for service families.

3.2 Disclaimer

All developments of this paper are to be understood relative to inSeq theory, rather than to be claimed for programming in general. For instance the distinction made between technical specification and requirements specification as made above is preferably understood as being specific for the setting of inSeq theory.

In this manner a seemingly unnecessary limitation of the scope of conceptual development is introduced. For doing so we have the following reasons/arguments:

(a) by having the various concepts defined relative to inSeq theory contradictions with other literature (not based on inSeq theory) about (literally) the same notations are (trivially) avoided,

(b) making too broad claims, and untenable claims for that reason, is avoided,

(c) contrasts between related but different notions can be made sharper by providing additional emphasis or detail to some aspects of the definition of a concept while relaxing demands w.r.t. other aspects (both in comparison with the software engineering literature at large),

(d) if in later work a need arises to modify the definition of a notion it is easy to grasp how that might impact on preceding work, which then is limited to work that has been explicitly based on the theory of inSeq's setting,

(e) whenever a reader considers it useful to adopt certain definitions in a different setting, e.g. involving other program notations and programming styles and methods, then importing one or more of the definitions while claiming adapted generality is always an option.

(f) An additional justification of the limitation of the scope of conceptual development in the paper to a fairly narrow setting lies in the idea that the difficulty of concept development lies not so much in providing the definition of a single concept but rather in developing a useful framework of mutually related and dependent notions. By looking at a framework of related notions complications and distinctions may arise which remain hidden otherwise. An example of such a distinction in the context of the paper at hand is that we will be able to introduce

various types of technical specification faults, while at the same time we will not find any basis for discussing requirements specification faults, and we must keep open the option that a meaningful notation of a requirements specification fault cannot be found.

4 Example: Specifying a Component for a Program for Adding 5-bit Numbers

We look for an inSeq $\mathtt{X}$ with a parameter α that acts as a part of the focus for various basic actions which satisfies the following requirements specification $\mathtt{S_{req}}$:

$\mathtt{S_{req}}$ requires first of all that for $\mathtt{k} \in \mathtt{N}$, $[\mathtt{k}/\alpha]\mathtt{X}$ (the result of substituting $\mathtt{k}$ for α in $\mathtt{X}$) exclusively applies methods for single bit registers (Boolean registers) with the following foci: $\mathtt{aux}, \mathtt{in1{:}k}, \mathtt{in2{:}k}, \mathtt{out{:}k}$. In other words the foci for methods occurring in $\mathtt{X}$ are in $\{\mathtt{aux}, \mathtt{in1{:}}\alpha, \mathtt{in2{:}}\alpha, \mathtt{out{:}}\alpha\}$.

In terms of functionality $\mathtt{S_{req}}$ requires that the inSeq $\mathtt{Y_5[X]}$ (as given below) performs addition of 5 bit binary numbers a and b where a is stored in single bit registers (SBR's, each with a method interface providing 16 methods, as defined in [11]) with focus $\mathtt{in1{:}1}, \ldots, \mathtt{in1{:}5}$ and b is stored in $\mathtt{in2{:}1}, \ldots, \mathtt{in2{:}5}$. Outputs are to be delivered in SBR's $\mathtt{out{:}1}, \ldots, \mathtt{out{:}5}$. The initial value of register $\mathtt{aux}$, meant to contain a carry bit, is arbitrary.

```
Y5[X] = aux.0/0;
•5
'k=1,step=1
   (+in1:k.i/i{;[k/α]X;}{;
     +in2:k.i/i{;
         +aux.i/i{;out:k.0/0;}{;out:k.0/1;};
         }{;
         +aux.i/0{;out:k.0/1;}{;out:k.0/0;};
         };
     }
   );!
```

This description of a generic inSeq requires some explanation. We notice that when $\mathtt{X}$ has n instructions then $\mathtt{Y_5[X]}$ has $(n+16)\cdot 5+2$ instructions. The carry bit placed in the single bit register under focus $\mathtt{aux}$ is first initialised to $\mathtt{0}$, then subsequently for each $\mathtt{k} \in [\mathtt{1}, \mathtt{5}]$ for the pair of inputs $\mathtt{in1{:}k}, \mathtt{in2{:}k}$ a sequence of $n + 16$ instructions is included which works thus: if reading $\mathtt{in1{:}k}$ results in value $\mathtt{1}$ then $[\alpha/\mathtt{k}]\mathtt{X}$ is performed, and otherwise

```
..{;
      +aux.i/i{;out:k.0/0;}{;out:k.0/1;};
      }{;
      +aux.i/0{;out:k.0/1;}{;out:k.0/0;};
      }
```

is performed. The latter instruction sequence achieves the following: having checked that `in1:k` contains `0` the contents of the carry bit and of `in2:k` are added (in binary arithmetic) and the least significant bit is placed in `out:k` while the most significant resulting bit is placed in the SBR with focus `aux`.

4.1 Further Clarification of S_{req}

The description of $\mathtt{S_{req}}$ is to some extent informal. By adding some clarifying remarks more precision is obtained. These remarks may be thought of as replies given by the project supervisor to the designer of $\mathtt{S_{tech}}$.

1. (Initialisation of output registers.) No assumptions on initial values of output SBR's are made, and that $\mathtt{Y_5[X]}$ must work in all cases.

2. (Details of addition w.r.t. most significant bits.) If the sum exceeds 2^5 so that 5 output bits don't suffice it is required requiring that just before termination `aux` is set to value 1).

3. (Specification of single bit registers.) Implicitly *true* is identified with 1 and *false* is identified with 0, that may be made explicit.

4. (Choice of inSeq notation.) A plausible inSeq notation is PGLC (termination does not require ! and takes) place once the position after the last instruction is reached). Moreover it is plausible to state explicitly that α can occur as a parameter only in the following foci: $\mathtt{in1}{:}\alpha, \mathtt{in}{:}2\alpha, \mathtt{aux}{:}\alpha$, and in no other manner,

With these clarifications we hold that the relation $\mathtt{Y_5[X]}$ $\underline{\mathtt{sat}}$ $\mathtt{S_{req}}$ is rigorously defined as a property of `X`.

4.2 $\mathtt{S^0_{tech}}$ a Plausible Technical Specification

Carrying on with the example a plausible technical specification $\mathtt{S^0_{tech}}$ of `X` is thus:

$\mathtt{S}^0_{\mathtt{tech}} \equiv \mathtt{S}^0_{\mathtt{interface}} \& \mathtt{S}^0_{\mathtt{functional}}$ with
$\mathtt{S}^0_{\mathtt{interface}} \equiv (\mathtt{I}_{\mathtt{required}}(\mathtt{X}) = \mathtt{aux}.\mathtt{I}_{\mathtt{method}}(\mathtt{SBR})$
$\cup\{\mathtt{in1}{:}\alpha.\mathtt{i}/\mathtt{i}, \mathtt{in2}{:}\alpha.\mathtt{i}/\mathtt{i}, \mathtt{out}{:}\alpha.0/0, \mathtt{out}{:}\alpha.0/1\}$, and
$\mathtt{S}^0_{\mathtt{functional}} \equiv (\mathtt{post}(\mathtt{out}{:}\alpha) = (\mathtt{pre}(\mathtt{in2}{:}\alpha) + \mathtt{pre}(\mathtt{aux}) + 1) \bmod 2 \wedge$
$((\mathtt{pre}(\mathtt{aux})) = 1 \wedge \mathtt{pre}(\mathtt{in2}{:}\alpha) = 1) \to \mathtt{post}(\mathtt{aux}) = 1) \wedge$
$((\mathtt{pre}(\mathtt{aux})) = 1 \wedge \mathtt{pre}(\mathtt{in2}{:}\alpha) = 0) \to \mathtt{post}(\mathtt{aux}) = 1) \wedge$
$((\mathtt{pre}(\mathtt{aux})) = 0 \wedge \mathtt{pre}(\mathtt{in2}{:}\alpha) = 1) \to \mathtt{post}(\mathtt{aux}) = 1) \wedge$
$((\mathtt{pre}(\mathtt{aux})) = 0 \wedge \mathtt{pre}(\mathtt{in2}{:}\alpha) = 0) \to \mathtt{post}(\mathtt{aux}) = 0) \wedge$
$\mathtt{post}(\mathtt{in1}{:}\alpha) = \mathtt{pre}(\mathtt{in1}{:}\alpha) \wedge \mathtt{post}(\mathtt{in2}{:}\alpha) = \mathtt{pre}(\mathtt{in2}{:}\alpha))$.

Here $\mathtt{pre}(\mathtt{f})$ equals the content of the service (SBR) in focus $\mathtt{f}$ just before performing $\mathtt{X}$ and $\mathtt{post}(\mathtt{f})$ is the content of the service (SBR) in focus $\mathtt{f}$ just after performing $\mathtt{X}$.

Proposition 4.1 $\mathtt{S}^0_{\mathtt{tech}}$ *is implementable and allows the following implementation in the inSeq notation PGLC extended with instructions for the conditional construct, as outlined in [8]:*

$\mathtt{P} =$
$+\mathtt{in2}{:}\alpha.\mathtt{i}/\mathtt{i}\{;$
$+\mathtt{aux}.\mathtt{i}/1\{;\mathtt{out}{:}\alpha.0/1;\}\{;\mathtt{out}{:}\alpha.0/0;\};$
$\}\{;$
$+\mathtt{aux}.\mathtt{i}/\mathtt{i}\{;\mathtt{out}{:}\alpha.0/0;\}\{;\mathtt{out}{:}\alpha.0/1;\};$
$\}$

Proposition 4.2 *With* $\mathtt{P}$ *as in Proposition 4.1 it is the case that* $\mathtt{Y}_5[\mathtt{P}]$ $\underline{\mathtt{sat}}$ $\mathtt{S}_{\mathtt{req}}$.

Proposition 4.3 $\mathtt{S}_{\mathtt{spec}}$ *is sufficient w.r.t.* $\mathtt{S}_{\mathtt{req}}$ *and* $\mathtt{Y}_5[-]$.

4.3 Weakening $\mathtt{S}^0_{\mathtt{tech}}$ (e.g. to $\mathtt{S}^1_{\mathtt{tech}}$ Below) May Introduce a Laski Fault

In $\mathtt{S}^1_{\mathtt{tech}}$ below it is possible that the carry is set to 1 in case it ought to be set to (or kept at) 0.

$\mathtt{S}^1_{\mathtt{tech}} \equiv \mathtt{S}^0_{\mathtt{interface}} \& \mathtt{S}^1_{\mathtt{functional}}$ with
$\mathtt{S}^1_{\mathtt{functional}} \equiv (\mathtt{post}(\mathtt{out}{:}\alpha) = (\mathtt{pre}(\mathtt{in2}{:}\alpha) + \mathtt{pre}(\mathtt{aux}) + 1) \bmod 2 \wedge$
$((\mathtt{pre}(\mathtt{aux})) = 1 \wedge \mathtt{pre}(\mathtt{in2}{:}\alpha) = 1) \to \mathtt{post}(\mathtt{aux}) = 1) \wedge$
$((\mathtt{pre}(\mathtt{aux})) = 1 \wedge \mathtt{pre}(\mathtt{in2}{:}\alpha) = 0) \to \mathtt{post}(\mathtt{aux}) = 1) \wedge$
$((\mathtt{pre}(\mathtt{aux})) = 0 \wedge \mathtt{pre}(\mathtt{in2}{:}\alpha) = 1) \to \mathtt{post}(\mathtt{aux}) = 1) \wedge$
$((\mathtt{pre}(\mathtt{aux})) = 0 \wedge \mathtt{pre}(\mathtt{in2}{:}\alpha) = 0) \to \mathtt{post}(\mathtt{aux}) = \mathtt{post}(\mathtt{aux})) \wedge$
$\mathtt{post}(\mathtt{in1}{:}\alpha) = \mathtt{pre}(\mathtt{in1}{:}\alpha) \wedge \mathtt{post}(\mathtt{in2}{:}\alpha) = \mathtt{pre}(\mathtt{in2}{:}\alpha))$.

It is easy to imagine an implementation $\mathtt{Q}$ of $\mathtt{S}^1_{\mathtt{tech}}$ such that $\neg(\mathtt{Y}_5[\mathtt{Q}]\ \underline{\mathtt{sat}}\ \mathtt{S}_{\mathtt{req}})$, for instance if the carry aux is always set to 1 by Q. It follows that

Proposition 4.4 *(i)* $\mathtt{S}^1_{\mathtt{tech}}$ *is implementable. (Any implementation of* $\mathtt{S}^0_{\mathtt{tech}}$ *also implements* $\mathtt{S}^1_{\mathtt{tech}}$*),*

(ii) $\mathtt{S}^1_{\mathtt{tech}}$ *is not wrong (same argument),*

(iii) $\mathtt{S}^1_{\mathtt{tech}}$ *is insufficient. (One may imagine an implementation* P *of* $\mathtt{S}^0_{\mathtt{tech}}$ *where the carry is always set to* 1*) in that case the addition of* $(1,0,0,0,0)$ *and* $(0,0,0,0,0)$ *produces:* $(1,1,0,0,0,)$ *which constitutes a failure for* $\mathtt{Y}_5[\mathtt{P}]$*),*

(iv) $\mathtt{S}^1_{\mathtt{tech}}$ *has a* $9-$*Laski fault. (Indeed by changing* $\mathtt{post}(\mathtt{aux}) = \mathtt{post}(\mathtt{aux})$*) to* $post(aux) = 0$ *one may transform* $\mathtt{S}^0_{\mathtt{functional}}$ *into* $\mathtt{S}^1_{\mathtt{functional}}$ *a sufficient specification is obtained.)*

(v) $\mathtt{S}^1_{\mathtt{tech}}$ *has a* $2-$*Laski fault (assuming that multiplication is allowed in the specification notation). (Indeed by changing* $\mathtt{post}(\mathtt{aux}) = \mathtt{post}(\mathtt{aux})$*) to* $\mathtt{post}(\mathtt{aux}) = 0 \cdot \mathtt{post}(\mathtt{aux})$ *one may transform* $\mathtt{S}^0_{\mathtt{functional}}$ *into a specification which logically equivalent to* $\mathtt{S}^1_{\mathtt{functional}}$ *and therefore is a sufficient specification. It also follows that the presence and size of specification faults depends on the specification notation.)*

4.4 Modifying $\mathtt{S}^0_{\mathtt{tech}}$ to $\mathtt{S}^{1b}_{\mathtt{tech}}$ Renders it Wrong

In $\mathtt{S}^{1b}_{\mathtt{tech}}$ below the carry is set to 1 in case it ought to be set to (or kept at) 0.

$$\begin{aligned}
&\mathtt{S}^{1b}_{\mathtt{tech}} \equiv \mathtt{S}^0_{\mathtt{interface}} \& \ \mathtt{S}^{1b}_{\mathtt{functional}} \text{ with}\\
&\mathtt{S}^1_{\mathtt{functional}} \equiv (\mathtt{post}(\mathtt{out}{:}\alpha) = (\mathtt{pre}(\mathtt{in2}{:}\alpha) + \mathtt{pre}(\mathtt{aux}) + 1) \bmod 2 \wedge\\
&\qquad ((\mathtt{pre}(\mathtt{aux})) = 1 \wedge \mathtt{pre}(\mathtt{in2}{:}\alpha) = 1) \rightarrow \mathtt{post}(\mathtt{aux}) = 1) \wedge\\
&\qquad ((\mathtt{pre}(\mathtt{aux})) = 1 \wedge \mathtt{pre}(\mathtt{in2}{:}\alpha) = 0) \rightarrow \mathtt{post}(\mathtt{aux}) = 1) \wedge\\
&\qquad ((\mathtt{pre}(\mathtt{aux})) = 0 \wedge \mathtt{pre}(\mathtt{in2}{:}\alpha) = 1) \rightarrow \mathtt{post}(\mathtt{aux}) = 1) \wedge\\
&\qquad ((\mathtt{pre}(\mathtt{aux})) = 0 \wedge \mathtt{pre}(\mathtt{in2}{:}\alpha) = 0) \rightarrow \mathtt{post}(\mathtt{aux}) = 1) \wedge\\
&\qquad \mathtt{post}(\mathtt{in1}{:}\alpha) = \mathtt{pre}(\mathtt{in1}{:}\alpha) \wedge \mathtt{post}(\mathtt{in2}{:}\alpha) = \mathtt{pre}(\mathtt{in2}{:}\alpha)).
\end{aligned}$$

Proposition 4.5 *(i)* $\mathtt{S}^{1b}_{\mathtt{tech}}$ *is implementable,*

(ii) $\mathtt{S}^{1b}_{\mathtt{tech}}$ *is wrong (adding* $(1,0,0,0,0)$ *and* $(0,0,0,0,0)$ *will fail),*

(iii) $\mathtt{S}^{1b}_{\mathtt{tech}}$ *has a* $1-$*Laski fault. (Indeed by changing a single* 1 *to* 0 *the sufficient specification* $\mathtt{S}^0_{\mathtt{tech}}$ *is regained.)*

4.5 Another Wrong Specification

In $\mathtt{S}^{1c}_{tech}$ the arithmetic of addition is mistaken ($+1$ is missing) with the consequence that each implementation of $\mathtt{S}^{1c}_{tech}$ fails to satisfy $(\mathtt{Y}_5[\mathtt{Q}]\ \underline{\mathtt{sat}}\ \mathtt{S}_{req})$.

$\mathtt{S}^{1c}_{tech} \equiv \mathtt{S}^{0}_{interface} \& \mathtt{S}^{1c}_{functional}$ with

$$\begin{aligned}\mathtt{S}^{1c}_{functional} \equiv\ & (\mathtt{post}(\mathtt{out}{:}\alpha) = (\mathtt{pre}(\mathtt{in2}{:}\alpha) + \mathtt{pre}(\mathtt{aux})) \bmod 2 \wedge \\ & ((\mathtt{pre}(\mathtt{aux})) = 1 \vee \mathtt{pre}(\mathtt{in2}{:}\alpha) = 1) \rightarrow \mathtt{post}(\mathtt{aux}) = 1) \wedge \\ & ((\mathtt{pre}(\mathtt{aux})) = 0 \wedge \mathtt{pre}(\mathtt{in2}{:}\alpha) = 0) \rightarrow \mathtt{post}(\mathtt{aux}) = 1) \wedge \\ & ((\mathtt{pre}(\mathtt{aux})) = 0 \wedge \mathtt{pre}(\mathtt{in2}{:}\alpha) = 1) \rightarrow \mathtt{post}(\mathtt{aux}) = 1) \wedge \\ & ((\mathtt{pre}(\mathtt{aux})) = 0 \wedge \mathtt{pre}(\mathtt{in2}{:}\alpha) = 0) \rightarrow \mathtt{post}(\mathtt{aux}) = 0) \wedge \\ & \mathtt{post}(\mathtt{in1}{:}\alpha) = \mathtt{pre}(\mathtt{in1}{:}\alpha) \wedge \mathtt{post}(\mathtt{in2}{:}\alpha) = \mathtt{pre}(\mathtt{in2}{:}\alpha)).\end{aligned}$$

Proposition 4.6 $\mathtt{S}^{1c}_{tech}$ *involves a* $2-$*Laski fault. (By adding a summand* $+1$ *an implementable and sufficient specification (i.e.* $S^0)_{tech}$*) is obtained).*

4.6 Further Weakening $\mathtt{S}^{1}_{tech}$ May Introduce One or More MFJ Faults

In $\mathtt{S}^{2}_{tech}$ below it both possible that the carry is set to 1 in case it ought to be set to (or kept at) 0 and the other way around.

$\mathtt{S}^{2}_{tech} \equiv \mathtt{S}^{0}_{interface} \& \mathtt{S}^{2}_{functional}$ with

$$\begin{aligned}\mathtt{S}^{2}_{functional} \equiv\ & (\mathtt{post}(\mathtt{out}{:}\alpha) = (\mathtt{pre}(\mathtt{in2}{:}\alpha) + \mathtt{pre}(\mathtt{aux}) + 1) \bmod 2 \wedge \\ & ((\mathtt{pre}(\mathtt{aux})) = 1 \wedge \mathtt{pre}(\mathtt{in2}{:}\alpha) = 1) \rightarrow \mathtt{post}(\mathtt{aux}) = 1) \wedge \\ & ((\mathtt{pre}(\mathtt{aux})) = 1 \wedge \mathtt{pre}(\mathtt{in2}{:}\alpha) = 0) \rightarrow \mathtt{post}(\mathtt{aux}) = 1) \wedge \\ & ((\mathtt{pre}(\mathtt{aux})) = 0 \wedge \mathtt{pre}(\mathtt{in2}{:}\alpha) = 1) \rightarrow \mathtt{post}(\mathtt{aux}) = \mathtt{post}(\mathtt{aux})) \wedge \\ & ((\mathtt{pre}(\mathtt{aux})) = 0 \wedge \mathtt{pre}(\mathtt{in2}{:}\alpha) = 0) \rightarrow \mathtt{post}(\mathtt{aux}) = \mathtt{post}(\mathtt{aux})) \wedge \\ & \mathtt{post}(\mathtt{in1}{:}\alpha) = \mathtt{pre}(\mathtt{in1}{:}\alpha) \wedge \mathtt{post}(\mathtt{in2}{:}\alpha) = \mathtt{pre}(\mathtt{in2}{:}\alpha)).\end{aligned}$$

Proposition 4.7 *(i)* $\mathtt{S}^{2}_{tech}$ *is implementable. (Any implementation of* $\mathtt{S}^{1}_{tech}$ *also implements* $\mathtt{S}^{2}_{tech}$*),*

(ii) $\mathtt{S}^{2}_{tech}$ *is not wrong (same argument),*

(iii) $\mathtt{S}^{2}_{tech}$ *is insufficient. (Immediate because* $\mathtt{S1}_{tech}$ *is insufficient),*

(iv) $\mathtt{S}^{2}_{tech}$ *has a* $1-$*MFJ fault. (Indeed by changing a single* 1 *to* 0 *the specification* $\mathtt{S}^{1}_{tech}$ *can be regained.),*

Proof: Only (*iv*) requires attention. Consider an implementation $\mathtt{P}$ of the following specification

$\mathtt{S}^{2b}_{tech} \equiv \mathtt{S}^{0}_{interface} \& \mathtt{S}^{2b}_{functional}$ with

$$\begin{aligned}\mathtt{S}^{\mathtt{2b}}_{\mathtt{functional}} \equiv (&\mathtt{post}(\mathtt{out}{:}\alpha) = (\mathtt{pre}(\mathtt{in2}{:}\alpha) + \mathtt{pre}(\mathtt{aux}) + 1) \bmod 2 \wedge\\ &((\mathtt{pre}(\mathtt{aux})) = 1 \wedge \mathtt{pre}(\mathtt{in2}{:}\alpha) = 1) \rightarrow \mathtt{post}(\mathtt{aux}) = 1) \wedge\\ &((\mathtt{pre}(\mathtt{aux})) = 1 \wedge \mathtt{pre}(\mathtt{in2}{:}\alpha) = 0) \rightarrow \mathtt{post}(\mathtt{aux}) = 1) \wedge\\ &((\mathtt{pre}(\mathtt{aux})) = 0 \wedge \mathtt{pre}(\mathtt{in2}{:}\alpha) = 1) \rightarrow \mathtt{post}(\mathtt{aux}) = 0) \wedge\\ &((\mathtt{pre}(\mathtt{aux})) = 0 \wedge \mathtt{pre}(\mathtt{in2}{:}\alpha) = 0) \rightarrow \mathtt{post}(\mathtt{aux}) = 1) \wedge\\ &\mathtt{post}(\mathtt{in1}{:}\alpha) = \mathtt{pre}(\mathtt{in1}{:}\alpha) \wedge \mathtt{post}(\mathtt{in2}{:}\alpha) = \mathtt{pre}(\mathtt{in2}{:}\alpha)).\end{aligned}$$

Now $\mathtt{P}$ is an implementation of $\mathtt{S}^{\mathtt{2}}_{\mathtt{tech}}$ as well because $\mathtt{S}^{\mathtt{2b}}_{\mathtt{tech}}$ refines $\mathtt{S}^{\mathtt{2}}_{\mathtt{tech}}$. Consider input $\mathtt{s} = (\mathtt{a}, \mathtt{b})$ with $\mathtt{a} = (1, 0, 0, 0, 0)$ and $\mathtt{b} = (1, 0, 0, 0, 0)$ for $\mathtt{Y}_5[\mathtt{P}]$. On this input pair $\mathtt{s}$ the result of the computation of $\mathtt{Y}_5[\mathtt{P}]$ will be $(0, 0, 0, 0, 0)$ which is not in conformance with $\mathtt{S}_{\mathtt{req}}$. We find that $\mathtt{s}$ is a symptomatic failure case for $\mathtt{P}$. Moreover by changing (with an appropriate γ with edit distance 1) $\mathtt{post}(\mathtt{aux}) = 0$ to $\mathtt{post}(\mathtt{aux}) = 1$ that particular failure is resolved, though the resulting specification specification $\gamma(\mathtt{S}^{\mathtt{2}}_{\mathtt{spec}}) \equiv \mathtt{S}^{\mathtt{1b}}_{\mathtt{spec}}$, say with implementation $\mathtt{Q}$, is still wrong, according to Proposition 4.5. Now $\gamma(\mathtt{S}^{\mathtt{2}}_{\mathtt{spec}})$ has a 1-Laski fault which can be repaired by changing the last instance of $\mathtt{post}(\mathtt{aux}) = 1$ into $\mathtt{post}(\mathtt{aux}) = 0$ thereby regaining $\mathtt{S}^{\mathtt{0}}_{\mathtt{tech}}$. □

4.7 A Connection With Regression Testing

The specification $\mathtt{S}^{\mathtt{2}}_{\mathtt{tech}}$ and its implementation P with change $\gamma(\mathtt{S}^{\mathtt{2}}_{\mathtt{spec}})$ and symptomatic failure $\mathtt{s}$ and a regression test suite consisting of a single test $\mathtt{s}'$ for adding $\mathtt{a}' = (0, 0, 0, 0, 0)$ and $\mathtt{b} = (0, 0, 0, 0, 0)$ with result $(0, 0, 0, 0, 0)$ provides an example of a $1-$RTJoC fault for $\mathtt{S}^{\mathtt{2}}_{\mathtt{tech}}$.

5 Concluding Remarks

For the notion of a program specification we may make use of classical literature such as [3, 15, 14]. Although less common such specifications may involve performance characteristics as well. In more than mere functional properties are specified from a program we may speak of an extended functional (EF) specification.

Some authors, however, claim that a specification determines for a software component $\mathtt{X}$ "how it works", and might prefer to refer to an EF-specification as a requirements specification instead. Understanding of specification as being about the how rather than the what is mentioned as the second option in [19], and is also mentioned as an existing viewpoint in [15]. 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 the notion of a fault becomes quite detached from observable program behaviour.

Therefore in the paper it is assumed that a specification P is in fact a functional specification or more generally an EF-specification. Having looked into the details of program faults, cast in the setting of instruction sequences, it seems plausible to contemplate the notion of a specification fault. As it turns out, however, unlike with the case of program faults we were unable to find any existing work on specification faults, apart from the idea that if a specification has no implementation something must be wrong. In [7] the notion of an algorithm fault is defined, a definition which is indirectly based on the notion of a program fault.

For a specification to be considered faulty some reference to another description of constraints for a program is needed, and in fact the other description must be attributed a higher status. We have chosen to speak of technical specifications and to introduce requirements specifications as the label for the background against which a claim that a specification is faulty is to be justified. Moreover we have chosen to assign requirements specifications the role of use case descriptions which for that reason are too specific for a certain application to be included as a part of a program specification. Under these assumptions we find that the theoretical framework for defining and analysing program faults as pioneered by [17] and [18] can be generalised to the case of program specifications.

An obvious question, which we have not attempted to solve is to find out whether in the existing body of practical software specifications the different notions of fault as identified in the paper can be recognised. Another topic for future work is to investigate to what extent notions of fault for technical specifications can be maintained once requirements specifications are allowed to be informal rather than formal. A further question might be to investigate to what extent is makes sense to speak of faults in a requirements specification. The current paper gives no clue on how to conceptualise faults in a requirements specification. Intuitively however, one may easily imagine a failure taking place during the phase of requirements capture which then becomes inadvertedly codified in a requirements specification document.

References

[1] Algirdas Avizienis, Jean-Claude Laprie, and Brian Randell. Fundamental concepts of computer system dependability. In *IARP/IEEE-RAS Workshop on Robot Dependability: Technological Challenge of Dependable Robots in Human Environments*, 2001.

[2] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004. doi:10.1109/TDSC.2004.2.

[3] Robert Balzer and Neil M. Goldman. Principles of good software specification and their implications for specification languages. In *American Federation of Information Processing Societies: 1981 National Computer Conference*, volume 50 of *AFIPS Conference Proceedings*, pages 393–400. AFIPS Press, 1981. doi:10.1145/1500412.1500468.

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

[5] Jan A. Bergstra. Instruction sequence faults with formal change justification. *Scientific Annals of Computer Science*, 30(2):105–166, 2020. doi:10.7561/SACS.2020.2.105.

[6] Jan A. Bergstra. Qualifications of instruction sequence failures, faults and defects: Dormant, effective, detected, temporary, and permanent. *Scientific Annals of Computer Science*, 31(1):1–50, 2021. doi:10.7561/SACS.2021.1.1.

[7] Jan A. Bergstra. Defects and faults in algorithms, programs and instruction sequences. *Transmathematica*, 2022. doi:10.36285/tm.49.

[8] Jan A. Bergstra and M. E. Loots. Program algebra for sequential code. *Journal of Logic and Algebraic Programming*, 51(2):125–156, 2002. doi:10.1016/S1567-8326(02)00018-8.

[9] Jan A. Bergstra and Cornelis A. Middelburg. Instruction sequence processing operators. *Acta Informatica*, 49(3):139–172, 2012. doi:10.1007/s00236-012-0154-2.

[10] Jan A. Bergstra and Cornelis 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.

[11] Jan A. Bergstra and Cornelis A. Middelburg. A short introduction to program algebra with instructions for boolean registers. *Computer Science Journal of Moldova*, 26(3):199–232, 2018. URL: http://www.math.md/publications/csjm/issues/v26-n3/12735/.

[12] Alonzo Church. Application of recursive arithmetic to the problem of circuit synthesis. In *Summaries of the Summer Institute of Symbolic Logic*, pages 3–50. Cornell University, 1957.

[13] Wided Ghardallou, Ali Mili, and Nafi Diallo. Relative correctness: A bridge between proving and testing. In Mohamed Ghazel and Mohamed Jmaiel, editors, *10th Workshop on Verification and Evaluation of Computer and Communication System, VECoS 2016*, volume 1689 of *CEUR Workshop Proceedings*, pages 141–156. CEUR-WS.org, 2016.

[14] John V. Guttag, James J. Horning, and Jeannette M. Wing. Some notes on putting formal specifications to productive use. *Science of Computer Programming*, 2(1):53–68, 1982. doi:10.1016/0167-6423(82)90004-1.

[15] James J. Horning. Issues and observations. In Jørgen Staunstrup, editor, *Workshop on Program Specification*, volume 134 of *Lecture Notes in Computer Science*, pages 5–24. Springer, 1981. doi:10.1007/3-540-11490-4_2.

[16] Jean-Claude Laprie. Dependable computing and fault tolerance: concepts and terminology. In *Twenty-Fifth International Symposium on Fault-Tolerant Computing*, pages 2–11, 1995. doi:10.1109/FTCSH.1995.532603.

[17] Janusz W. Laski. Programming faults and errors: Towards a theory of software incorrectness. *Annals of Software Engineering*, 4:79–114, 1997. doi:10.1023/A:1018966827888.

[18] Ali Mili, Marcelo F. Frias, and Ali Jaoua. On faults and faulty programs. In Peter Höfner, Peter Jipsen, Wolfram Kahl, and Martin Eric Müller, editors, *14th International Conference on Relational*

and Algebraic Methods in Computer Science, volume 8428 of *Lecture Notes in Computer Science*, pages 191–207. Springer, 2014. doi:10.1007/978-3-319-06251-8_12.

[19] Jeannette M. Wing. Program specification. In Anthony Ralston, Edwin D. Reilly, and David Hemmendinger, editors, *Encyclopedia of Computer Science*, pages 1454–1458. Wiley, 4th edition, 2003.

© Scientific Annals of Computer Science 2022