Scientific Annals of Computer Science vol. 32 (1), 2022, pp. 5–86
doi: 10.7561/SACS.2022.1.5

A Survey of Testing for Instruction Sequence Theory

Jan A. Bergstra[1]

Abstract

Using the conceptual analysis of instruction sequence faults, failures, and defects as developed by the author in [10] and [12], a survey of testing is developed as an extension of a theory of instruction sequences. An attempt is made to develop a consistent terminology regarding instruction sequence testing while taking into account the literature on software testing at large.

Keywords: program algebra, instruction sequence, service, test, bug, specification, fault

1 Introduction

I will use instruction sequences based on the notations of PGA style program algebra (as outlined in [16]) as the starting point for collecting a coherent sample of concepts as well as technical definitions and additional information on testing. I will refer to the theory of instruction sequences that is being developed in said paradigm as *inSeqTh*. I will often abbreviate instruction sequence to *inSeq*. Referring to inSeqTh as "the theory of instruction sequences" is unwarranted, however, as inSeqTh is just one particular form of a theory of instruction sequences.

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

The objective of this paper is to extend inSeqTh with a chapter on testing. Given the vast literature on program testing this objective suggests an open ended endeavour without a well-demarcated outcome. An account of testing will be obtained, which might be developed into a more systematic and comprehensive form in due time. I will denote the resulting account of testing with testing4inSeqTh. The paper contains an attempt to provide an admittedly incomplete survey of testing which is cast in the terminology and the conceptual framework of inSeqTh. In addition several notions are either provided with more precise definitions than commonly done in papers on testing, or choices are made between different options that can be found in the literature on testing.

The paper contains no theorems and proofs, a novelty lies, hoever, in the collection (as well as design) of formal as well as informal definitions pertaining to the area of testing. I hope to contribute to the "philosophical logic for computer science", i.e. logic for computer science with the style of philosophical logic.

1.1 Motivating "testing4inSeqTh"

The introduction and use of the label testing4inSeqTh is motivated on two disparate grounds:

(i) (facilitating eclectic configuration:) development of testing4inSeqTh involves a significant number of design decisions each of which might be altered, and upon taking different choices substantially different accounts of testing for inSeqTh can be imagined, say testingB4inSeqTh, and testingC4inSeqTh. The label testing4inSeqTh supposedly does not suggest that "the" (unique) theory of testing for inSeqTh is being configured, and

(ii) (maintaining compatibility with existing literature on testing:) the literature on testing is quite inconsistent in terms of the meanings that different authors assign to various terms and therefore making choices is unavoidable when setting up testing4inSeqTh. Using the header testing4inSeqTh, instead of say testing4programs (which would express a more universal ambition), allows me to make choices without literally contradicting parts of the literature on software testing with each design decision that is being taken.

Given an extensive literature on different forms of software testing three aspects require systematic attention for in relation to a development of testing4inSeqTh:

(i) selection of topics: which methods, techniques and principles to discuss and include,

(ii) how to arrive at a consistent and coherent whole, and

(iii) to what extent to allow the peculiarities of inSeqTh to determine the resulting account of testing.

In summary the objectives of this project are these: A (collection of themes), B (working towards a consistent terminology) and C (if possible: harvesting, i.e. drawing conclusions the relevance of which may extend beyond testing4inSeqTh). Regarding A, B, and C the paper involves making choices from a plurality of options. The main choices which are proposed for adoption in testing4inSeqTh are as follows:

A (collection): to create (under the label of testing4inSeqTh) an informative and simplified outline of software testing at large and casting this outline in a setting of instruction sequences. Unavoidably, many themes were left out, for instance testability analysis (following [83]) and non-testability (following [87]),

B (streamlining): to present an account of "testing for PGA style instruction sequences" in a coherent and consistent manner, by making design choices which might be problematic when writing a survey on program testing.

  (1) to view testing as being included in verification rather than the other way around,

  (2) not to include formal verification in testing, while formal verification is included in verification,

  (3) not to include code walk-through and related static methods in testing,

  (4) to use test case (w.r.t. a given instruction sequence) for the inputs of a test; to use test sample for the pair of inputs and outputs of a test run complemented with a flag signalling how the test run was ended; to use verdicted test sample (or verdicted test case, or completed test case) for the combination of

a test sample with a verdict about it; to use test battery for a set of test cases meant for being applied to the same program; to use (verdicted) test suite for a collection of (verdicted) test cases that have been applied to the same inSeq; to view various coverage metrics as $[0, 1]$ valued meta-verdicts on test suites,

(5) to take fail, pass, none and error as the four possible verdicts (following TTCN-3),

(6) to take an act of testing as a physical process (computation) starting with a test case (and an ISuT i.e. instruction sequence under test) and ending with a test sample, (issuing the verdict is not included in said act of testing, however),

(7) to assume that acts of testing are performed within a test configuration (i.e. a testing context for the ISuT) under the guidance (i.e. control) of a test agent and to make the test agent responsible for providing a termination flag which is to be included in the test sample and which is chosen from these four options: termination-ok, termination-nok, interrupted, aborted.

(8) not to provide "test" or "testing" with a technical definition. Within testing4inSeqTh "test" is intentionally ambiguous, and it may refer to test case, test sample, completed test sample, but also, as is the case in the social sciences to a particular method and protocol for testing, i.e. investigating a certain class of subjects by applying various tests (i.e. means of investigation) to these subjects,

(9) to consider a fault as static phenomenon which (by definition) is a cause of a failure (a dynamic phenomenon); not to assume that all failures are caused by faults,

(10) to speak of a learnt instruction sequence rather than a model or an ML program in the context of machine learning,

C (harvesting): to improve the understanding of various concepts in the area of testing by making use of the simplification which comes about from a restriction to program algebra style instruction sequences. For instance:

(1) an attempt to clarify the notions of a bug, a bug report, and a dormant bug,

(2) viewing testing as a notion which features partisan ambiguity, and in fact intentional ambiguity (these refinements of ambiguity are discussed in detail in Paragraph 2.1 below),

(3) working with informal but rigorous definitions of: test case, test sample, verdicted test sample, test suite (both verdicted and non-verdicted),

(4) viewing testing as being included in verification, though not in formal verification.

1.2 Taking into Account the Diverse Literature on Testing

There is a multitude of survey papers and books on software testing. What these works seem to have in common, however, is that authors pay little attention to work by other authors which they disagree with or do not approve of. I consider each definition of testing, and of related notion, that is presented in a book or a survey paper on software testing to some extent problematic if no attention is paid to alternative definitions that have been proposed in earlier work. By not paying attention to other perspectives a degree of topical unity and coherence is suggested which is in fact largely absent in the area of program testing.

For configuring testing4inSeqTh the entire literature on software testing matters, at least in principle, including various manifest disagreements (such as about the relative status of verification and testing). When adopting a certain point of view or notation (that means to adopt for inclusion in testing4inSeqTh), it is not implied that I consider authors maintaining different views to be misguided.

1.3 The Point of Departure, and How to Proceed

An outline of inSeqTh can be found in the following papers: [16], [24], [20], and [13].

A chapter on faults, failures, bugs and flaws for inSeqTh (referred to as FFBF4inSeqTh) has been outlined in [15], [10] and [12]. These papers primarily describe the following notions: (ALR) failure, ALR fault, Laski fault, MFJ fault, RTJoC fault, and dormant failure. FFBF4inSeqTh is a theme of ongoing research.

For the development of testing4inSeqTh I will adopt the conventions and definitions on faults and failures as discussed/proposed in FFBF4inSeqTh. In particular this choice implies the notion that a fault is a fragment of an instruction sequence which causes a failure where causation must be justified by means of knowing in which way the replacement by way of a known change of said fragment remedies the occurrence of said failure.

A chapter on deterministic multi-threading for inSeqTh (that may be referred to as DMT4inSeqTh) has been developed in [17], [18] and [19].

1.4 A Conclusion Turned into an Assumption

The remarkable diversity in the literature cannot be accommodated by simply choosing the best available option in each case. For instance the distinction between black-box testing and white-box testing appear very often, but in the well-known monograph [2] one finds that this very distinction is considered obsolete, and in [80] the black-box/white-box terminology is not used. In agile testing according to [85], however there is a role for black-box testing as well as for white-box testing. Concerning the question whether testing is a quest for failures or a quest for confidence, various authors disagree while [7] notices that this kind of judgement, as made by a programmer or by a programming team, is an indication concerning a level of testing maturity, where viewing testing as quest for confidence indicates the higher level of the two, though not the highest level available. For an account from industrial practice regarding the succession of trends about testing I refer to [6]. I have developed the following working hypothesis as a conclusion, which retrospectively has functioned as structuring principle for this work.

Working hypothesis on inSeq testing. The working hypothesis I wish to advance has four parts: WH1, WH2, WH3, and WH4:

WH1: Program testing occupies a niche between four related themes, and testin4inSeqTh will reflect the influence of the four sides involved:

(i) *Formal verification.* In formal the context of verification where the coexistence of a specification (to be implemented) and a system (prospective implementation) under verification (SuV) is taken for granted, and testing invariably seems to be a second best option, especially when portrayed exclusively as being about the relation between a specification and a system (prospective implementation) under test (SuT). The classical theory of testing (as in e.g. [49]) fits in this compartment but plays no significant role.

(ii) *Debugging.* Debugging has not become a theme in theoretical computer science, though its methods and techniques (e.g. slicing) have. In the area of debugging the contrast between specification and implementation is less prominent than in formal verification. Debugging is very much a theme in the psychology or programming. Testing, however, seems to be remote from programmer experience and programming psychology, two core themes concerning debugging.

(iii) *Software process modelling and organisation.* Viewed from the perspective of software process modelling and various forms of team activity, program testing misses the spirit of engineering on a grand scale, and it primarily represents an unfortunate cost factor which must be contained.

(iv) *Automaton theory.* Finally seen from automaton theory, program testing is remarkably informal and non-mathematical, in spite of various attempts to provide program testing with a dedicated mathematical foundation. Automaton theory, however, allows to provide a comprehensive theoretical background for testing various kinds of systems, a capability which seems not always to reach practitioners in an adequate and timely manner.

WH2: In each of the four areas just mentioned in WH1 testing has a secondary status only. Turning testing into a topic with a core that serves the interaction with each of these neighbouring themes is still a meaningful challenge.

WH3: Only by taking the experimental character of testing as the basis of its theoretical development, a core account of testing meaningful to each of these four areas can emerge (a view implicitly present in [67]).

WH4: in due time, the emergence of quantum computing will turn attention of testing theory to proper experimentation. Quantum programs will be tested at a large scale and performance assessment will become central to testing of such programs.

In [34] the argument is made the other way around: quantum programs will defeat testing and render formal verification more important because of the intrinsic difficulty and counterintuitive mechanics of quantum computing. In [66] it is argued that even for classical probabilistic programs, i.e. deterministic inSeq's with various options for tossing a coin, testing is quite problematic because a (single) test run cannot disclose a failure, and for that reason cannot suggest the presence of a fault. The latter argument, however, disappears upon understanding testing broadly as the application of experimental methods for analysing program quality. Nevertheless a significant literature on testing quantum programs is emerging, mainly since 2020, e.g. [55, 71].)

I intend to set up testing4inSeqTh in such a way that it is not overly biased in either of these four directions. Unavoidably a discrepancy with established literature results. If I would replace inSeq testing by software testing, or by program testing, for instance the choices made for testing4inSeqTh (now read as a theory of program testing) would be in disagreement with [80] on at least the following terms and phrases: mistake, verification (versus testing), test, test case, while the would be partial agreement on fault, bug, and error. The thought experiment of synthesising testing4inSeqTh eliminates the risk of prematurely contradicting existing literature on program testing such as [80].

1.5 Other Chapters of Instruction Sequence Theory

Before taking testing in focus it is informative to imagine which other themes might lead to chapters for a theory of instruction sequences. Some chapters have, in fact been developed in ample detail already; I will only list the themes, proper references can be found elsewhere.

- CC&MM4inSeqTh on control codes and machine modelling. Control code is low level in the sense that it is directly linked to a processing mechanism for a hardware model. The CC&MM chapter which has been developed for inSeqTh thus far is based on Maurer machines.
- GCS4inSeqTh on garbage collection and shedding: Molecular programming, data linkages, garbage collection, shedding.
- RP4inSeqTh on reflexive programming: variations of the halting problem, including variants of Cohen impossibility for virus detection,

Some options for further chapters for extending inSeqTh can be mentioned:

(i) Software metrics. Here a useful starting point is the question to find a metrics for various instruction sequence notations which satisfy the 9 criteria on metrics as collected and proposed in [88],

(ii) definitions of the notion of program and algorithm,

(iii) code compactness: minimisation of the number of instructions needed for various tasks, and

(iv) quantum computing from a perspective of inSeqTh.

Certain themes, however important, are out of the reach of an approach based on PGA instruction sequences. For instance topics in the area of programmer productivity can hardly be studied from the perspective of program notations which are not used in practice or which, like the PGA inSeq notations, have not been designed with practical use in mind.

2 Terminology for Ambiguous Concepts

For software testing just as for many other topics the ambiguity of terminology, as it occurs in the literature, may stand in the way of achieving a unified exposition. For instance some authors include software verification in software testing, and other authors do not, while yet other authors include software testing in software verification. I take this state of affairs as an indication that software testing is an ambiguous concept. However, more can be said about this particular instance of ambiguity: (i) I prefer not to include formal software verification in software testing and to consider testing as being included in verification, but (ii) different choices can work and there are no compelling grounds for said preference.

Not all instances of ambiguity must be resolved by way of rigorous definitions, however, as ambiguity may be intentional. In this section I will introduce some terminology about concepts in relation to (un)ambiguity, (in)formality, and rigour. This terminology will be used below. The proposed terminology for ambiguity is unspecific for testing and might be included as well in other efforts of theory design. Below, in Paragraph 9.11, I will illustrate the various forms of ambiguity with an example from elementary arithmetic.

2.1 Refinements of Notions of Ambiguity

I will assume that a concept C comes with a term (also referred to as C) which serves as a name for it. I will speak of a concept/term, where term may also be phrase. I mention some 50 concept/terms that occur in the area of testing where I restrict focus to deterministic systems exclusively:

> test, testing, test case, test suite, regression test, mutation test, metamorphic test, random test, structural-test, black-box testing, white-box testing, grey-box testing, observation, verdict, oracle, automated oracle, fault, bug, failure, error, mistake, bug report, log file, fault localisation, fault repair, dormant fault, dormant failure, dormant bug, bug finding, verification, formal verification, proof checking, validation, quality control, specification, requirements document, requirements capture, compliance, confidence, trust, risk, confidence level, release, maintenance, patch, security vulnerability, security fault, consistency with specification, implementation, instruction, branch, line, exercising, execution, running.

Each of these concept/terms may feature ambiguity (it seems to me that most do) and making "best choices" in each case is unattractive because of the risk to end up with an account that deviates from (and thereby disagrees with) virtually every existing account of testing regarding the interpretation of one or more concept/terms.

Definition 2.1 *A concept C is subject to* accidental lexical ambiguity *(or accidental ambiguity for short) if the term (phrase) C refers to another concept in a quite different context.*

For instance: black-box, the box one does not, can not, or will not look into, versus Black's box (sometimes also referred to as black-box) which one can look into even after a crash; in both cases the color black is not involved; or as another example: bat as an animal and bat as used in baseball. Accidental lexical ambiguity seems not to be a significant issue in the area of program testing. The concept/term effective ("computable" versus "having become materialised") serves as an example of accidental lexical ambiguity that features in the testing literature.

Definition 2.2 *A concept/term C is subject to* systematic ambiguity *if the term (phrase) C refers to a plurality of related concepts*

For instance: number may stand for natural number, integral number (integer), rational number, algebraic number, real number, complex number, non-standard number, surreal number etc. I hold that as a concept/term "number" is subject to systematic ambiguity.

Definition 2.3 *A concept/term is C* relatively unambiguous *if it is defined in a way which narrows down its possible meanings quite significantly, although further ramification may be possible, mainly by making one or more parameters to the definition more explicit.*

I consider "formal verification" to be relatively unambiguous (though one might disagree as to whether formal verification requires in excess of a formal approach to program semantics also a formal approach to the construction and checking of the parts of a proof that are exclusively dealing with the underlying mathematics).

Definition 2.4 *A concept/term C is subject to* partisan ambiguity *if different (groups of) human agents assign different though related meanings to it.*

Partisan ambiguity concerning C may be present in cases where agents are not even aware of alternative (though related) interpretations for C. Occurrence of partisan ambiguity is plausible only in cases which feature systematic ambiguity. While accidental ambiguity is easily resolved by clarification of the context at hand, dealing with partisan ambiguity may be more challenging. For an author A who approaches a topic involving C and who notices the presence of partisan ambiguity concerning C in the literature, an obvious option is to choose between different options as provided in the literature. I will refer to making such a choice as partisan disambiguation. A first advantage of partisan disambiguation is that there will be ample examples of use of the chosen option which A can follow, and as second advantage is that there is an audience used to the chosen understanding of C. A disadvantage of partisan disambiguation lies in the risk of entering into a disagreement with authors who prefer other interpretations of C i.e. who performed partisan disambiguation as well but who made a choice for a different interpretation of C.

Definition 2.5 *When choosing for a concept C which is subject to partisan ambiguity one of the known meanings for it, an author applies* partisan disambiguation *to it.*

I prefer to write about testing while not engaging in partisan disambiguation to terms relating to program testing. Precisely this is achieved by writing about testing in the admittedly artificial context of instruction sequence theory.

Definition 2.6 *A concept/term C is subject to* disputed ambiguity *if the concept is subject to partisan ambiguity and if moreover different (groups of) users of C explicitly disagree about the proper meaning of the concept*

Definition 2.7 *A concept/term C is understood by one or more agents as being* intentionally ambiguous *if:*

(i) C features systematic ambiguity,

(ii) its use (by said agents) combines different meanings for each of which also more specific terminology is available (to these agents), and

(iii) said agents acknowledge the plurality of meanings of C involved.

It follows from this definition that intentional ambiguity of concept/term may be a judgement which is not universally shared. For instance if an author A applies partisan disambiguation to a concept/term C while another author B declares to acknowledge (and use) intentional ambiguity of the same concept C then A and B are in disagreement on that matter.

2.2 Examples from Programming and Program Testing

Below I will assume the ALR concepts for failure, fault, error, and mistake as known (see [10, 12] for more detail, which is based on [3, 4], [61], and [69]). I do not, however assume that the ALR definitions of the respective concepts would be widely accepted in the world of program testing. In these cases by adopting said interpretation I engage in partisan disambiguation. By providing an account of testing4inSeqTh partisan disambiguation will be avoided (in this paper anyhow).

1. Software testing:

   (i) not subject to accidental ambiguity,

   (ii) subject to systematic ambiguity according to some but not according to others,

   (iii) subject to partisan ambiguity (some authors include one or more of: code walk through, formal verification, model checking in software testing, while other authors view testing as a part of verification, and yet other authors take verification for a part of testing, and finally testing and verification may also be considered disjoint and complementary in a useful manner),

(iv) not subject to disputed ambiguity (authors are hardly ever explicit about the difference of their interpretation of software testing with the interpretation favoured by other authors, though there are explicit differences regarding the objectives of software testing).

2. Failure:

   (i) not subject to accidental ambiguity,

   (ii) relatively unambiguous (fairly general agreement that a failure is a dynamic phenomenon which takes place if a run proceeds in such a way as to be not in conformance with the specification and/or requirements at hand, which is the ALR interpretation of failure.)

3. Fault:

   (i) subject to accidental ambiguity (also a notion in geophysics),

   (ii) subject to partisan ambiguity (the ALR definition of fault is more narrow than other interpretations of fault which may include failure, bug, and defect),

   (iii) ALR fault provides a relatively unambiguous definition of fault (Laski fault specialises ALR fault by being explicit about the justification of causality at hand, and so do the notions of MFJ-fault and RTJoC-fault as discussed in [10]),

   (iv) I see no indication of disputed ambiguity of the notion of fault in informatics.

4. Bug:

   (i) is accidentally ambiguous (the first computer bug as found by Grace Hopper is claimed to have been the remains of an insect at the same time),

   (ii) bug is subject to partisan ambiguity (some authors assume that bugs occur in programs other authors include failures i.e. dynamic violations of the specification in the notion of bug, while other authors take all bug reports for bugs, and yet others do so sometimes but not always),

   (iii) bug is not subject to disputed ambiguity,

   (iv) "bug" seems to be intentionally ambiguous for many authors on the subject.

5. Error:
   (i) not accidentally ambiguous,
   (ii) subject to partisan ambiguity (some authors include ALR failures, other authors do not),
   (iii) does not feature disputed ambiguity,
   (iv) the ALR concept of error is too vague to be relatively unambiguous, although it was probably meant to be.

6. Dormant failure:
   (i) not accidentally ambiguous,
   (ii) relatively unambiguous.

7. Test case:
   (i) not accidentally ambiguous,
   (ii) subject to partisan ambiguity (one may identify test case with test input, or with test input plus an abstraction of the corresponding run, or one may (additionally) include the resulting verdict in the test case),
   (iii) no sign of disputed ambiguity,
   (iv) intentional ambiguity is plausible.

2.3 Residual Informality

Sometimes a notion has been informally defined to such a degree that an attempt to qualify the amount of ambiguity is implausible.

Definition 2.8 *A concept/term C is* residually informal *if its various definitions are informal, and no strong incentives seem to be in place to arrive at a rigorous or formal definition.*

I consider the following concept/term's to be essentially informal: computer, software, software engineering, software quality, problem, defect, testing, requirements capture, team, teamwork, machine, hardware, execution (of a program on a machine), interpretation (of a program on a machine), design, module, modularity, comprehensibility, structured programming, agile programming, causality, practice, research, development (as in R & D).

Definition 2.9 *A concept is* rigorously defined *if it has a definition on the basis of which it need not be qualified as essentially informal.*

I consider failure (understood as ALR failure) and ALR fault to be key examples of rigorously defined concept/term's. The same holds for verdict, and to a lesser extent for oracle. A concept/term may be considered problematic under some conditions. That is a subjective notion for which the following definition may be helpful.

2.4 Problematic Concept/Term's

Beyond intentional ambiguity, the ambiguity of a concept/term may in some case be considered problematic by some users of the concept/term at hand.

Definition 2.10 *A concept/term (given a definition for it) is problematic if it meets one or more of the following criteria:*

(i) the concept/term has a rigorous definition, which, however is not taken as authoritative by the creators of the definition, and this situation has not been changed in subsequent literature,

(ii) the concept/term features an unexpected level of ambiguity (unnecessary intentional ambiguity, unnecessary partisan ambiguity, unnecessary disputed ambiguity),

(iii) the concept/term lacks a rigorous definition while it might profitably be given a rigorous definition.

I consider the concept/term fault to be problematic. The wide-spread use of fault in the testing literature taken in combination with the equally wide-spread lack of clarity about what constitutes a fault renders much of the testing literature unrigorous to an unnecessary extent.

2.5 Examples of Concept/Term Qualifications

- Testability: in [43] over 30 definitions of testability are listed. An ambiguity arises from the fact that (i) some definitions take testability for a property/virtue of requirements and (ii) some definitions take testability for a property of programs or software components, (iii) yet other definitions take both the program and its requirements into account. Nevertheless each of the definitions is informal which is witnessed by the occurrence of the words "ease", "effectiveness", "degree to which", and "facilitate". I conclude that for some authors testability is essentially informal, while for other authors testability admits a rigorous definition.

- Untestability is an informal notion. The seemingly obvious conclusion that formal verification (including model checking) is the only remaining option available for gaining confidence in an untestable program is never drawn as far as I know.

- Dormant fault: as argued in [12] dormant failure is an unproblematic concept with a rigorous definition, whereas dormant fault seems not to have a convincing definition; dormant fault is problematic in view of case (iii) of the definition.

- At first sight the definition of a dormant bug in [35] is unproblematic as long as one refers to bugs that have been reported already, but then there are never any yet unknown dormant bugs, which deviates from what the authors have in mind. I conclude that [35] does not comply with the definition of a dormant bug given in the same paper. Dormant bug is a problematic concept/term for that reason.

- The notion of a fault (as specific as an ALR fault) features residual informality which can be avoided by means of being specific about change justification, a view adopted in [10], on the basis of [61] and [69]).

3 Subjective Aspects of Programming and Testing: A Rudimentary Agent Model

Some accounts of testing introduce a historic development of testing with subsequent phases, for instance [46] suggests the following phases:

- up to 1976: debugging oriented (becoming aware of the ubiquity and impact of bugs),

- 1957–1978: demonstration oriented (checkout oriented),

- 1979–1982: destruction oriented (fault detection as the primary objective of testing),
- 1983–1987: evaluation oriented (quality assurance oriented),
- from 1988: prevention oriented (systematically working towards the absence of faults).

I will now focus on the notion of a checkout (the programmer's assertion that the inSeq works) which plays a central role in these matters. However naive the idea of a checkout may be, the intuition of that notion is so obvious that it deserves a place in testing4inSeqTh. For this purpose I will introduce the programmer as an agent and use the language of promise theory (see [11, 14]).

Testing theory is often discussed in the context of a software process model, such as e.g. the famous waterfall model. Instead I will work with a rudimentary agent model: a programmer, say A, produces a program for a user, say B (representing a user community), and some agents in scope of the process, (e.g. a common manager $M_{a,b}$ of A and B, and/or some certification authority C, or a representative member of the user community other than B, or a lawyer known to A and B, to mention some possibilities).

> **InSeq checkout promise:** (programmer) A promises to (customer/user) B (with scope U) that inSeq X provides a satisfactory solution for the problem captured in requirements S_{req}.

Acceptance is primarily a matter for the customer/user of an inSeq. In [70] it is made explicit that acceptance comes with a customer side perspective and it is also emphasised that the notions involved are quite informal. Acceptance is complementary to checkout.

> **InSeq acceptance promise:** (customer/user) B promises to (programmer) A (with scope U) that inSeq X provides a satisfactory solution for the problem captured in requirements specification S_{req}.

The inSeq checkout promise does not require a specific methodological underpinning. I will use delivery if some model for software quality assurance is available and is being followed.

InSeq delivery promise: (programmer) A promises to (customer/user) B (with scope S) that inSeq X provides a satisfactory solution for the problem captured in requirements specification S_{req} and that its delivery in the given state is in accordance with specified guidelines G to that extent.

Prior to an exchange of X the programmer and the customer/user are likely to exchange a requirements specification S_{req} and a technical specification S_{tech}.

Requirements proposal promise: (customer/user) B promises to (programmer) A (with scope U) that their (i.e. B's) needs have been adequately captured in requirements specification S_{req}.

Requirements acceptance promise: (programmer) A promises to (customer/user) B (with scope U) that their needs as captured in requirements specification S_{req} constitute an adequate starting point for designing a corresponding technical specification S_{tech}.

Technical specification checkout promise: (programmer) A promises to (customer/user) B (with scope U) that their needs as captured in requirements specification S_{req} are adequately dealt with in technical specification S_{tech}.

Technical specification acceptance promise: (customer/user) B promises to (programmer) B (with scope U) that their needs as captured in requirements specification S_{req} are adequately dealt with in technical specification S_{tech} as proposed by A.

More likely than not the requirements specification and the technical specification will evolve during an inSeq development process. It is implicit in Promise Theory (following [14]) that no obligations emerge from promises and that upon having made a promise the promiser may still change their position.

3.1 Testing, Validation, Verification, Informal Verification, Formal Verification

I will use verification as the most general category of inSeq quality assurance. Under informal verification I include informal approaches as well as formal approaches. Informal approaches include for instance code walkthrough and informal correctness proof.

Now various terms can be linked to the promises the making of which involves the corresponding activities:

- inSeq checkout promise: verification (no particular emphasis on any particular style of verification),
- inSeq acceptance promise: testing against S_{req}, [optional: testing against S_{spec}],
- inSeq delivery promise: testing against S_{req}, [optional: formal verification against S_{spec}],
- Requirements proposal promise: informal verification (in this phase often called validation) [optional: rapid prototyping/testing],
- Requirements acceptance promise: informal verification [optional: prototyping/testing],
- Technical specification checkout promise: informal verification [optional: formal verification, automated prototyping/testing],
- Technical specification acceptance promise: informal verification [optional: automated prototyping/testing],

Here it is assumed that prototyping turns a requirements specification into a working system (the prototype), which then can be investigated via testing. If a technical specification is available it may be possible to generate a prototype automatically (that is at low cost).

3.2 From Claiming a Simultaneous Evolution of Testing to Claiming Concurrent Evolutions of Testing

The perspective that world-wide testing (as an activity taking place during and after software development process) moves through a series of phases as outlined above in this Section has been amended in [7] as follows: instead of perceiving a collective evolution of testing one focuses on an evolution of testing that takes place many times in various organisations that produce computer programs. Such organisations then, according to [7] evolve through a succession of testing maturity levels.

For testing4inSeqTh I prefer not to think in terms of the history or histories of testing process evolution and have a simple agent based model in mind where agents may freely choose from actions (promises) of each maturity level.

4 Security Testing

Avoiding security problems plays a central role in testing. If anywhere it is in the area of security that testing stands out as a necessity which will not easily become obsolete.

Security testing is often presented as a coherent subarea of software and system testing. It has its own terminology including these phrases: vulnerability scanning, security scanning, intrusion detection, penetration testing, SQL injection, ethical hacking, DevSecOps (DevOps integrating security testing and protection, while like in DevOps replacing the waterfall style development by a more gradual transition to usage), left shifting and right shifting. In [40] the following informative classification of security testing is presented and used:

(1) model-based security testing is grounded on requirements and design models created during the analysis and design phase,

(2) code-based testing and static analysis on source and byte code created during development,

(3) penetration testing and dynamic analysis on running systems, either in a test or production environment, as well as

(4) security regression testing performed during maintenance.

Explanations of security testing of programs and/or systems are conventionally not stated in terms of the intended functionality of said programs and/or systems.

4.1 Security Testing in the Absence of a Positive Notion of Security

Traditional program testing comprises a collection of methods and techniques/technologies that support the development of correct programs, where correctness is a concept that can be defined in theory and that has an intuitive appeal. In contrast security testing is not a toolkit helpful for obtaining secure programs. There is no notion of a secure program in theory and security is a systemic notion to such an extent that understanding security at a component level and thinking of secure systems as secure compositions of secure components is too simple. Security of a system is about being well-protected against known threats and about being easily adaptable to incorporate protection when new security threats have been identified. Just as a person cannot (at a given moment) be protected simultaneously against all conceivable viruses, while a healthy person is protected against the viruses the circulate in their context of life, a programmed computing system cannot at any stage in its life-cycle be protected against all conceivable attacks. The concept of a secure system rests on a survey of known and unknown but readily imaginable threats and attacks against all of which protection is either in place or achievable with relative ease.

4.2 Security testing4inSeqTh

Work on testing4inSeqTh may at best follow developments in security oriented testing with significant delay. I will describe some initial steps along that path where I will follow a line of historical development in the following account of this theme, while commenting on relations (or possible relations) with instruction sequences. In [60] the following is written on the question "what is a security flaw".

> This question is akin to"what is a bug?". In fact, an inadvertently introduced security flaw in a program is a bug. Generally, a security flaw is a part of a program that can cause the system to violate its security requirements.

Besides security flaw the phrase security bug is frequently used. It seems to have the same meaning. According to [60] a security flaw in a program is a bug and more specifically it is an ALR fault.

I am not sure that it is plausible to expect a security failure to be caused by a Laski-fault, an MJF-fault or an RTJoC-fault. A security flaw is plausibly the cause of a security failure, but attribution of the failure to a fault in such a manner that an appropriate change on a single location makes the security flaw disappear may be asking too much. Instead a more significant program redesign may be needed which goes beyond the limitations of Laski-faults, MJF-faults and RTJoC-faults as instantiations of ALR faults.

4.3 Working With Simple Security Models

Security models range from early and simple, e.g. Lampson's notion of confinement in [59] to recent and complex, as e.g. in [25]. With instruction sequences as a point of departure it is plausible to start with early security models and to see to what extent such models may be understood as requirements on architectures which are expressed in terms of instruction sequences and to see to what extent notions of testing apply in such cases.

4.4 Lampson Confinement

It seems that translating the notion of confinement as informally proposed by Lampson in [59] into the notations of program algebra is not straightforward. In particular the guidelines on how to guarantee proper confinement are not easily translated. Confinement, however, may be understood with different objectives in mind, for instance to guarantee confidentiality or to guarantee integrity. Both these refinements of confinement have led to very well-known security models.

4.5 Bell LaPadula Security Model for Confidentiality

In connection with the Bell-LaPadula security model (BLP model, i.e. the model as proposed in [8]), it is plausible to assume that levels of confidentiality have been assigned to programs (threads) as well as to services (or rather to foci). Now the two key principles "no read up" (i.e. a thread must not read from an object with higher level of confidentiality than the thread has itself) and "no write down" (i.e. a thread must not write to an object with lower level of confidentiality than the thread has itself) may be translated as follows:

(i) if a thread P has been assigned level k and a focus f has been assigned level l with $k < l$ then there may be only a single method m for which a call of the form $f.m$ occurs in P. The idea is that when different methods of the service (say H) in focus f can be called it becomes possible for P to communicate inputs to H. A call $f.m$ may always be used by P to receive information from H by way of the reply it returns.

(ii) if a thread P has been assigned level k and a focus f has been assigned level l with $k > l$ then every call of the form $f.m$ which occurs in P must occur in a subprocess of the form $f.m \circ Q$. Recall that $f.m \circ Q =$

$Q \trianglelefteq f.m \trianglerighteq Q$. The idea is that by choosing which method to call P may transmit information to H via focus f while the reply value as produced by H will be ignored by P.

Now if the BLP model is imposed on a possibly multi-threaded configuration by providing an assignment of levels of confidentiality to threads and to foci, then it may be assessed whether or not said configuration is in compliance with the BLP model. Doing so, however, can easily be done my means of a syntactic check of the programs X_i for the given threads P_i: for each pair X_i and $f.H$ with k the confidentiality level of $P_i = |X_i|$ and l the confidentiality level of f:

(i) (case $k < l$) amounts to finding all calls $f.\mu$ that occur in X_i and confirming that at most a single method, say m is used.

(ii) (case $k > l$) amounts to checking that each method call of the form $f.\mu$ occurs in a void instruction (that is not in a test $+f.\mu$ or in a test $-f.\mu$).

At first sight BLP assessment of compliance involves a syntactic check only and testing plays no role. Now this is not fully precise as the program X_I may contain violations of these requirements (in case (i) say occurrences of $f.m1$ and $f.m2$) and in case (ii) e.g. a test instruction $+f.m$) whiles these violations are in fact unreachable so that the BLP is not violated after all during a run of the system. Nevertheless, testing is hardly justified in this matter.

4.6 Biba Security Model for Integrity

The Biba security model (i.e. the model proposed in [27]) complements the BLP model with mechanisms/protocols for the protection of integrity. Instead of confidentiality levels the Biba model makes use of integrity levels (though called security levels in [27]). This model advances a setting allowing for a plurality of security policies. Leaving out invocation for reasons of simplicity one of these policies can be formulated thus:

(i) (no write down after read up) a thread must not write into an (an object with) a certain integrity level after having read from a (an object with) a higher integrity level, (or otherwise the integrity level of that object is temporarily decreased to the lowest level of integrity of the objects from which the thread has been reading in the computation before performing the write),

(ii) (no write up) a thread must not write into (an object with) higher integrity level than the thread has itself (or otherwise the integrity level of that object is temporarily decreased to the level of the writing thread).

Using strategic interleaving with locking and blocking (see [17]) these constraints lead to assertions for which tests can readily be designed.

4.7 Non-interference

Goguen & Meseguer proposed in [47] to view non-interference as a mechanism for formulating criteria for security mechanisms that are amenable to testing and formalisation. In [84] and related work one may find a proposal on how to model non-interference in the setting of instruction sequence theory.

4.8 Recent Models

The classic models involving confidentiality levels, security levels and non-interference are very simple in comparison with what is needed to understand system security, say some 40-50 years later (as e.g. pursued in [25]). However, when preparing for testing, even without the ambition to advance towards formal verification, a significant modelling effort is needed. The latter modelling effort is needed just as well in advance of formal verification. I draw the following conclusions from this observation:

(i) At least in the context of security flaws, program testing constitutes a plausible activity which may well precede efforts towards formal verification, because both activities depend on modelling the system, while formal verification imposes higher demands on the model.

(ii) In the context of security flaws, when a system has been modelled and tests are being designed, the corresponding oracle problem may well be tractable because security flaws are likely to allow quite manifest problems to happen (such as e.g. a system crash).

(iii) Given a program/system P/S and its intended security model, establishing confidence in compliance of the system with its intended security model and establishing the security of P/S are quite different matters.

(iv) Formal verification of security of a program P is likely to take the following form: some pattern p_{attack} of attack is proposed, and a security model $m^{counter}_{p_{attack}}$ is proposed which supposedly guarantees that attacks following pattern p_{attack} will not be successful. Establishing the guarantee requires first of all a formal analysis of the model, which may be done in advance and be applied in a variety of cases. Subsequently it must be formally verified that the program P complies with model $m^{counter}_{p_{attack}}$. At least initially, so it seems, a testing approach can avoid the development and use of security models like $m^{counter}_{p_{attack}}$, which renders security testing attractive in comparison to formal verification.

5 Tensions and Choices for testing4inSeqTh

A tension occurs if the focus on instruction sequence theory is difficult to align with principles and objectives of software testing.

5.1 Program Notation Diversity

Work on program testing often makes use of some specific program notation, either for providing examples and explanation or as the basis of data extracted from various repositories containing data of production processes for programs in the notation at hand. At the same time most qualitative insights on program testing are independent of any specific program notation.

Design choice: I will make use of instruction sequence notation as proposed in [16] and as surveyed in [23] and for used for instance in [68]. These notations provide cornerstones of the PGA family of program algebras from [16].

Implication(s): An implication of this choice is that examples and counterexamples have to be provided in these notations. More importantly the chapter on testing for instruction sequences will not produce or import and use statistical results on testing in practice. Moreover by adopting instruction sequences as a point of departure a focus on deterministic systems comes about. Concurrency can take a deterministic form via strategic interleaving. However, if nondeterministic systems are to be tested a different theoretical background is required. A variety of proposals to that extent have been published to date. If timing is relevant, then discrete timing (see e.g. [5]) can be used in combination with multi-threading as captured by way of strategic interleaving.

5.2 Model to Reality Gap (for Software Testing)

Under the assumption that testing is a real world activity it is real programs running on real computers which yield observable outcomes that are subject to verdicts made by real agents. But a theoretical account will unavoidably substitute models and formal entities for "real" components, thereby introducing a model to reality gap.

Theories of testing may fully ignore that gap or may go into considerable detail about it. In [82] the general introduction to testing makes a difference between modelled components and their real counterpart. So [82] makes an explicit attempt to bridge the to model to reality gap for software/systems testing.

Design choice: I will work under the assumption that bridging the model to reality gap is left entirely to the reader.

Implication(s): Given a piece of theoretical work on testing the simplest way to bridge the model to reality gap will usually consist of constructing (hypothetical as well as practical) realities for which the claim that these are properly modelled by the theory at hand makes sense. Doing so is unlikely to result in practical conclusions, however, but it may help to understand what is needed for bridging the model to reality gap in other circumstances.

5.3 Industry to (Academic) Research Gap

In [41] one may find an extensive statement concerning the self-perceived needs of industrial software testers and the current (academic/fundamental) research on software testing. The authors claim the existence of a significant gap between these these two practices.

Design choice: Working towards testing4inSeqTh is not informed by any known needs of industry.

Now the literature on software testing, both in practice and in much of the research on testing, is by and large devoid of any theoretical account of what software testing is about, as if this were an obvious matter. That situation complicates the task for development of testing theories. The situation is in sharp contrast with quantum computing where academics as well as workers from industry seem to be in full agreement that a significant amount of theory constitutes a crucially important shared basis.

Implication(s): applications of the work on testing4inSeqTh are not easily imagined or obtained. Nevertheless, from an industrial perspective, it may be informative to experiment with the design of a comprehensive account of program testing in the presence of ample freedom to reconsider key definitions in order to work towards a systematic terminology.

5.4 Theory to Theory Gap

The interconnection between theoretical work on software testing and other work in theoretical computer science is not so clear. For instance in [64] one finds a recent proposal extending Kleene algebra with parallel composition. The proposal is subsequently illustrated with the design and analysis of tests for realistic example programs. The resulting extension of Kleene algebra is quite close to the process algebra ACP, though with parallel composition using action sharing in the style of TCSP. However, neither ACP nor (T)CSP nor CCS, a common ancestor of both, are mentioned in this paper.

I speak of a theory to theory gap to indicate that the selection of a theoretical framework for dealing with a certain issue in testing is quite often than not rather arbitrary.

Design choice: I will write about testing from my own background in theoretical computer science (TCS). My background comprises: λ-calculus, (conditional/prioritised) term rewriting, (formal) verification of sequential programs, computability theory in finite types, (abstract datatypes) and algebraic specifications thereof, process algebra (ACP style), program algebra (PGA style), proposition algebra (logic of sequential connectives with potential side effects).

Implication(s): while when writing the paper I made an attempt to cover the notion of testing such as to take a significant portion of the testing literature into account, a corresponding claim cannot be made regarding the use that is made of results from the theory of computing. For instance approaches to testing based on the following theoretical frameworks are not included or taken into account: type theory, category theory, functional programming, logic programming, temporal logic, and Bayesian inference.

5.5 Testing Scope Spectrum

Some hold that program walkthrough, formal verification, model checking, and various forms of static type checking are included in testing. Others claim that such is not the case and that testing is confined to experiments with running a program (or minor variations of it) on inputs during a phase of the software engineering life-cycle which precedes delivery and use. These observations correspond with the idea that testing is ambiguous, and in particular that the notion of testing involves partisan ambiguity.

I will assume that knowledge about an instruction sequence and its behaviour is likely to have three sources: (i) general theoretical assumptions including immediate consequences thereof, (ii) experimentation (e.g. testing, as done by oneself and by others), and (iii) detailed theoretical analysis.

Consider the situation that a safety critical instruction sequence X is embedded in a context $C[X]$. Would anyone trust the proper functioning of $C[X]$ purely on the basis of knowledge of the forms (i) and (iii) above? It seems reasonable to expect that some testing will always take place, if only to verify that the module X indeed fits in context $C[-]$. Assumptions concerning the context $C[-]$ are essential for acquiring confidence that $C[X]$ works well and is not malfunctioning because of some trivial (or non-trivial) problem which would come to light with almost every test. These considerations lead to the following assumption:

Assumption 5.1 *(Necessity of testing assumption.) Whatever theoretical framework one chooses, in safety critical circumstances, some form of testing will always be preferable (and if conditions allow, required) in advance of accepting a new software module (instruction sequence).*

In Assumption 5.1 may not hold in each and every case but it holds convincingly in case software is put into effect on bare hardware. In support of Assumption 5.1 I quote [65]:

> Despite advances in formal methods and verification techniques, a system still needs to be tested before it is used. Testing remains the truly effective means to assure the quality of a software system of non-trivial complexity [...], as well as one of the most intricate and least understood areas in software engineering [...].

From a practical perspective there may not be much doubt about Assumption 5.1, but its status from a theoretical perspective is questionable nevertheless. Some further comments on this matter are made in Paragraph 9.12 below. The following question is implicit in [67] where it is noticed that up to 2010 no definite information about this questions was found in the literature.

Problem 5.1 *Is it possible to "prove" the necessity of testing assumption on theoretical grounds? (Or is this a wrongheaded objective?)*

> *Design choice*: (instruction sequence) testing is (theory about) an experimental process involving actual or symbolic (simulated) runs of an instruction sequence (or minor validations thereof) on various inputs.
>
> *Implication(s)*: testing includes all knowledge acquisition about an instruction sequence which is not purely "static" (types (i) and (iii) above). Testing, however marginal, is considered essential for any effort of instruction sequence construction. But so are forms of static analysis, be it from first principles, or more sophisticated. Thus testing plays a key role for bridging the model to reality gap.

5.6 Testing Objectives Spectrum

The objectives of software testing are not easy to determine: at one end of the spectrum one finds the localisation of faults in a program as the core testing objective, at the other end of the spectrum one finds improvement or certification of a software engineering process as mentioned objectives of testing. In between are experiments which are meant to substantiate the confidence in a specific software product.

> *Design choice* A: black-box (instruction sequence) testing is about finding failures, and at the same time about increasing confidence if no or not many failures are found.
>
> *Design choice* B: not finding failures during testing creates confidence; however, not finding faults in the presence of known failures is likely to decrease confidence.
>
> *Design choice* C: instruction sequence testing primarily aims (in the context of testing4inSeqTh) at increasing the confidence that an instruction sequence will work well. Indeed only in the presence of an alternative path

towards the acquisition of confidence in a program one may discharge the relevant testing activity from an orientation towards confidence creation and direct the focus primarily or even exclusively towards the localisation of faults or the detection of failures.

Design choice D:

D1: instruction sequence testing is an ordinary component of the design and construction of instruction sequences. Testing is not a matter of assessment in hindsight but takes place in parallel with programming. Programming comes with formulating hypotheses about the program under construction and testing plays an important role in the assessment of such hypotheses. A relevant hypothesis need not be framed in terms of correctness or functionality. See e.g. [91].

D2: viewed as a part of professional programming, testing is becoming increasingly specialised into a plurality of directions. To mention some: symbolic testing, metamorphic testing, concolic testing, risk based testing, fuzzing (formerly known as random testing, see [29]), mutational fuzzing.

D3: the idea that testing is about guarantees that the user will be served with appreciated (and well-specified) functionality becomes less prominent. Such quality assurances are increasingly amenable to formal verification, while protection against hostile exploitation of vulnerabilities (the theme of security testing) has little in common with contemplation of user requirements.

D4: Confidence in certain programs may have grown too high during times of positive experiences with it, and systematic ongoing fuzzing may be meaningful in order to eliminate risks in the form of yet undetected (i.e. dormant) failures caused by yet unknown (and for that reason unlocalised) faults which might otherwise go undetected for far too long. Testing/fuzzing is as much relevant for creating confidence in specific software as for undermining confidence in that same software. It is plausible that a program is tested much more frequently than that it is used, and it is plausible that testing proceeds as long as a program is being used by any user until the program is officially withdrawn and declared obsolete.

Implication(s): as an implication from adopting designs choices A, B, C and D, I mention that each of these “axioms of instruction sequence testing" contributes aspects which, due to its purely theoretical nature, a theory of instruction sequence testing cannot demonstrate (or refute) from first principles. In other words, I take these "design decisions" as having been made very plausible by the practice of software testing, and as having been made explicit in various forms by corresponding research. At the same time these "axioms" have the status of additional assumptions from the perspective of instruction sequence testing. The very practice which could demonstrate (or refute) the validity of these "axioms" is missing, and that will not change.

5.7 Oracle Problem

The oracle problem for software testing is a difficult matter. There seems to be no obvious approach to this issue. How to judge whether or not the run of a program produces a satisfactory outcome?

However, it is helpful to consider cases where an oracle problem has simple solutions. A convincing and effective way out in some practical cases is found via metamorphic testing, see e.g. [77]. Metamorphic testing, as proposed in [36] is a black-box approach which aims at the detection of failures.

For instance consider an instruction sequence X which is supposed to multiply (as binary numbers of length 10.000 bits) two naturals n and m in order to obtain a third one (denoted $P(n,m)$). An instruction sequence for computing X is proposed for instance in [22]. Then a metamorphic test is to randomly choose inputs n and m and to perform the very straightforward test that $X(n,m) = X(m,n)$. If this identity is not valid a failure has been detected: either on input (n,m) or on input (m,m), or perhaps even on both inputs, X is not producing the right result. In fact all equations true in the semiring of natural numbers may be used for creating metamorphic tests.

Metamorphic black-box testing may detect failures of (the behaviour of) X (without assuming that a non-trivial oracle problem has been solved) and subsequently spectrum based testing (see e.g. [92]) may be used to locate candidate faults in X.

Mutation based testing (i.e. introducing faults or candidate faults, in an instruction sequence) can be used to investigate the effectiveness of black-box metamorphic testing for detecting failures, and of subsequent spectrum based fault localisation for finding candidate locations for faults.

Design choice: instruction sequence testing can be meaningful in cases where the oracle problem is trivially solved. A restriction to such cases creates room for the application of a range of methods, to mention: (i) black-box testing against a known oracle (e.g. a trusted program for multiplication in the example mentioned above), or in a setting of metamorphic testing, (ii) spectrum based (candidate) fault localisation, (iii) mutation testing, (iv) fuzzing, (v) formal program testing, (vi) symbolic testing, (vii) concolic testing, and finally (viii) coverage directed forms of testing (which do not require a non-trivial oracle) may uncover inaccessible parts of an instruction sequence.

Implication(s): instruction sequence theory is unlikely to provide instances where a human observer can assist in solving the relevant oracle problem.

5.8 History and State of the Art

In [42] one may find valuable information regarding the history of software testing. Notably in [42] the notion of a "bug" is used throughout the paper as the essence of "what one is looking for" without any conceptualisation of what a bug is. Following the analysis of [61] and lateron [69] "bug" is a non-trivial notion, I will return to bugs below.

Design choice: there is no such thing as the history of instruction sequence testing. Historical remarks must be drawn from the history of program testing.

Implication(s): whether or not awareness of the history of program testing is relevant for the activity of program testing I don't know. In any case such influence is not made explicit in the literature on testing, so that for the time being I see no merit in providing instruction sequence testing (understood as a model of program testing), with an artificial history of the field (serving as a model for that history of program testing) in order to investigate that sort of influence.

6 Bug: An Intentionally Ambiguous Concept

In work on testing it is often written that finding and removing bugs is a major objective. Bugs are the subject of many papers, many of which do not mention testing. Awareness of the presence bugs comes from the experience of users as much as from testing and from other forms of verification. I will adopt the following conclusions regarding the concept/term of a bug.

1. Bug first of all features as the expected content of bug reports (included in project log files) which play a role during a software engineering project. In that context a bug is a "perceived problem". The first mention of the bug in a log file opens the bug. Subsequently it can be assigned, reassigned, fixed, closed, reopened. These matters are connected with version management with the ISuC (inSeq under construction) at hand.

   The distinction between a pre release bug and a post-release bug is non-obvious. The simplest definition of these notions takes a bug which was first reported pre-release as a pre-release bug and a bug which was first mentioned after release as a post release bug. These notions are vague: a pre-release bug may be unfounded and a post-release bug may have been present in advance of the final acceptance tests.

2. All (instruction sequence) faults are bugs, but not all bugs are faults.

3. Bug is more inclusive than flaw because flaw involves a connotation of failure (as a dynamic phenomenon), which bug does not.

4. Bug refers to any problem which comes about from the design, production, and or maintenance of computer programs. Bug is more specific than "problem", however, even in the context of software engineering. For instance "problem" may also include having bought too expensive software or hardware, an event which would not be labeled as a bug. However, importing an inadequate module into a programming project might lead to a bug, and might be qualified as such.

5. Papers with a focus on bugs do not define this notion. Such papers also do not involve any notion of specification which is used as a yardstick for the assessment of "buggyness". Bug is often used as the most liberal class of software defect.

6. Root cause bug is mentioned in various papers, but I am unable to provide a definition for that notion. Neither can I provide a definition for a surprise bug (see [79]). The same holds for the idea of a blocking bug (a bug which supposedly hides the visibility of one or more other bugs).

7. Attempt to define a blocking bug: a blocking bug is a fault, say F in an inSeq which, upon having been repaired by a change in the most plausible manner, leads to an inSeq in which a new fault arises which was not present until said change was applied. The new fault is said to have been blocked by fault F, which is labelled a blocking bug for that reason.

   The above defnitional attempt is problematic because it suggests that the notion of a fault has a useful companion where the notion of a change is uncoupled from the actual improvement it achieves (i.e. the justification of change, which must take the appearance of the supposedly blocked bug into account) and where the notion of change is based instead upon the notion of a plausible change. There is no theoretical basis for the soundness of that suggestion, however.

8. Breakage bug (an intentionally introduced "bug" in order to avoid access to obsolete or otherwise problematic functionality in a system); a breakage bug is classified as a functional bug in [79]. This notion is unclear, it seems that because of maintenance the functionality of a program would be changed by applying a modification which, w.r.t. the original specification constitutes a bug.

9. The notion of a performance bug is used (see e.g. [75]) but it is hard to define. It is left open if the bug symptom consists of slow processing w.r.t. given requirements on timing, or merely that slow processing takes place where, upon a fix, faster processing would be possible. A performance fault can be defined as a fault which causes a performance failure. Now the notion of a performance failure combines that the performance is less than required with the existence of a change which brings performance back into the intended range.

10. The concept of bug is intentionally ambiguous. It is useful for testing4inSeqTh to avail of such a notion of bug.

11. In an American History item [58] it is stated as a commentary that nowadays bugs are an instance of computer malware. I will not adopt this meaning of bug for testing4inSeqTh.

12. Given the inclusiveness of the notion of bug, the idea of a bugfree program (serving as a component in a larger context) is only meaningful upon adopting a restricted interpretation of "bug" for instance by reading bug as fault.

13. In [78] is is explained that "bug" as used for a minor fault or defect in a machine or a plan dates back to Thomas Edison at least while Grace Hopper might be correctly linked to the first use of that term in connection with computer hardware: spotting and naming the first computer bug (in fact computer hardware bug). The term "bug" was used in engineering already for decades, however. That the bug (as found by Hopper) was actually a moth which had to be removed is a mere coincidence and plays no role as an origin of the term bug in computing.

14. Bugs are sometimes understood as changes (changes making a program depart from its ideal form, i.e. not changes that did occur during the development of the program at hand).

15. Mistakes are not bugs: mistake (if the word is used at all) refers to a programmer action which causes a fragment of a software component to "be buggy", i.e. to constitute a fault. A fault is often named bug, and repairing said fault by means of a change constitutes a fix of the bug.

16. A dormant bug is defined in [35], as a bug that was introduced in a certain version, say version n, and persists in a later version, being first reported in version $n + 2$ or later. The consequences of this definition are slightly counterintuitive. For instance:

    > We find that dormant bugs are fixed faster than non-dormant bugs: dormant bugs have a median fix time of 5 days and non-dormant bugs have a median fix time of 8 days. We also find that dormant bugs have a statistically significant higher reopen rate than that of non-dormant bugs, even though both types of bugs are rarely reopened (90% are never reopened).

    Another quote suggests that the authors do not require of a dormant bug that it has been reported already.

    > Our experiment results are based only on the fixed bugs, so there may be more dormant bugs in the systems. This problem exists in all studies that study bug reports.

17. The following quote is taken from [51]

> > The most common reason bug reports are reassigned is because people want to find the root cause of the problem before they are willing to attempt a fix. Bug reports usually only indicate superficial symptoms, but a high-quality fix should address the root cause and not merely patch the reported symptoms. The root cause is often in a completely different component than symptoms indicate, though.
> >
> > A survey respondent elaborates on this reason for why bugs are reassigned multiple times before being resolved:
> >
> > > "Bugs many times are exposed in the UI [user interface], but are not caused by the team writing the UI code. These bugs can pass down several layers of components before landing on a lower level component owner. As the UI team gets more familiar with the component layers they can more directly assign bugs to the offending component, but that takes time and knowledge."
> >
> > We can quantify the above phenomenon by correlating reassignments with changes in the "Component path" field of bug reports, which indicates in which component people currently believe a bug originates.
>
> Here "bug" seems to refer to failure, though "the problem" may be not quite the same as "the bug". Nevertheless bugs are being caused (as are failures) rather than being causes (as are faults). Moreover a bug originates in a component rather than that it is part of the component. But the idea that a bug trickles down suggests a combination of a failure and a diagnostic hypothesis.

The anecdote linking Grace Hoper to bugs suggests a technical definition as follows:

Definition 6.1 *A* Hopper bug *is a bug in hardware or software which, for any engineer skilled in the art, would be immediately recognisable as a minor defect once pointed at, to the extent that is it known to them (i) what it means that the bug would not be present, and (ii) how to remove said bug, i.e. to perform a change upon which the bug is not anymore present.*

It is not necessarily assumed that the presence of a Hopper bug in an artefact impedes its functionality. In other words a Hopper bug need not constitute a fault (which, by definition, causes some failure). A software Hopper bug is hard to imagine otherwise than as a syntax error, i.e. a fragment in a text which locates non-conformance with given syntactic requirements. Some security vulnerabilities may take the form of a Hopper bug.

6.1 Bugs and Bug Life-cycles

The notion of a bug is independent of a specific software process. The following may be assumed:

Assumption 6.1 *An ISoI (instruction sequence of interest) moves through a series of versions, say $(X_{v_1}, D_1), \ldots, (X_{v_n}, D_n)$. These versions come along with documentation (D_i for version v_i). All documentation is stored for later use. First of all D_{i+1} includes commits concerning changes which explain what was changed and why when moving from version v_i to version v_{i+1}. Comments D_{i+1} may also contain explanations regarding the life-cycle of various bugs. Such explanations are termed bug reports.*

Assumption 6.2 *(At least) the following events may occur in connection with a bug (i.e. during a bug life-cycle): opened (given a name and a unique description), assigned to an engineer for initial investigation, assigned to an engineer to be fixed, reassigned to another engineer, fixed per proposal, upon having been fixed per proposal assigned to an engineer for review and assessment of the proposal, fixed by application of a change to the instruction sequence, closed, and reopened.*

Definition 6.2 *A bug has been dormant at a version v_n if (i) it is opened at some version v_{k+2} with $k + 2 \leq n$, (ii) it could have been opened at version v_k or before (because the underlying problem is present with that version already).*

A bug which has been dormant is also called initially dormant. Dormancy is first of all a judgement which an external observer may make for the purpose of case independent research or for case dependent review and/or assessment.

However, a software engineer may open a bug in version v_{k+2} and then claim dormancy, i.e. the claim that the same bug could already have been opened in v_k or before. Mistaken bugs may be labeled as dormant. Fake bugs may only be labeled as dormant by a hostile software engineer.

An initially dormant bug is called dormant in the terminology of [35]. I consider the terminology of [35] to be unfortunate in this matter and I prefer to use "initially dormant" for that reason.

7 Testing Instruction Sequences: Terminology and Definitions

The idea of a testing theory for instruction sequences suggests that by specialising to a particular format, in this case instruction sequences (in PGA notation), one may obtain additional insights in testing, which, at least from a theoretical viewpoint may be relevant.

However, as stated above, obtaining additional insight for special cases is not my intention with the limitation to instruction sequences. The limitation to instruction sequences is meant to avoid making too large claims. Testing theory for instruction sequences (leading to testing4inSeqTh) constitutes a thought experiment aiming at developing a coherent framework with respect to testing which "works" in the case of instruction sequences.

If a design decision is made or proposed regarding testing4inSeqTh that state of affairs is not meant to imply that (according to the author) a similar design decision ought to be taken for software engineering at large. On the contrary, for software engineering at large additional aspects may need to be taken into account. By developing testing4inSeqTh no design decision risks being in flat contradiction with previous work, for the simple reason that, to the best of my knowledge no effort to design an account of testing for a theory of instruction sequences has been written and published before.

Assumption 7.1 *Failure, fault, error, and causality.*

(i) For testing4inSeqTh it is assumed that fault is understood as ALR fault (the cause of a failure, a dynamic phenomenon taking place during or at the end of a run, which in turn is the visible symptom of an error(state) that was reached during said run).

(ii) Laski-fault, MFJ-fault and RTJoC-fault are special cases of ALR fault, each more specific regarding the interpretation of causality at hand, as defined in [10].

(iii) Various qualifications of these notions (in particular: dormant failure) are as outlined in [12].

Assumption 7.2 *Testing4InSeqTh has an exclusive focus on deterministic systems, which includes multi-threading with strategic interleaving.*

Definition 7.1 *Testing is an experimental process where an instruction sequence under test (ISuT) or a system of those, possibly embedded in some context $C[-]$, and possibly obtained as a mutant of an ISoI (instruction sequence of interest) is put into effect with one or more inputs, so that the resulting computations can be observed and conclusions about these can be captured via a verdict. The objective of testing is to use the collected verdicts in order to acquire information about ISoI via information about ISuT*

Notably Definition 7.1 is far from obvious. To begin with many authors include code inspection and other static analysis methods in testing and include all or most activities aiming at the determination of confidence in a program under the umbrella of testing. Other authors view testing as a theme within verification. Notably in [52] testing is exclusively used for activities in search of failures and faults, while verification is exclusively used for activities meant to create confidence in a program.

Proposition 7.1 *(i) Viewed from the perspective of testing4inSeqTh, testing features partisan ambiguity (different groups assign different meaning and scope to it).*

(ii) (Relative to testing4inSeqTh) instruction sequence quality assurance serves as an umbrella for a wide range of activities and objectives including: testing, validation, verification, formal verification, and model checking.

(iii) testing is an intentionally ambiguous notion which refers to an ever increasing family of specialised approaches to it all in the context of Definition 7.1, to mention: black-box testing, white-box testing, functional testing, non-functional testing, structural testing, metamorphic testing, fuzzing (random testing), symbolic testing, concolic testing, fault directed testing.

Now it is important that Proposition 7.1 must not be read as a claim that for software testing in general this is the state of affairs. Proposition 7.1 as well as the definitions below make use of the liberty to depart from conventional terminology in the design of terminology for testing4inSeqTh.

Definition 7.2 *(Single act of testing X.) An act of testing (of X) consists of making an instruction sequence X performing a run within a test configuration context $TCC_\alpha[-]$ and at the same time collecting data regarding the run which that may be used as inputs for generating a verdict about the run. Here α contains various parameters which are instantiated for each test run.*

Thus (an act of) testing amounts to creating a run of $TCC_\alpha[X]$. $TCC_\alpha[-]$ will include data but does not include an operator A who operates the test environment and who is working concurrently with it. Importantly A may abort the run in order to obtain an incomplete computation, which may nevertheless be informative and may admit a verdict.

7.1 Test Case and Test Sample: First Definitions

I will first provide some definitions in style of original work on testing theory viz. [48] and [49]. In a deterministic setting without any interaction, one may for simplicity assume that a program X computes a function $[\![X]\!]$.

Then the first stage of a test t is a potential input for X, say i_t. Upon having performed the test run, and assuming termination of X on i_t an input-output pair (i_i, o_t) results with $o_t = [\![X]\!](i_t)$. Assuming an effective oracle O_X a verdict $v_t = O_X(i_i, o_t)$ with value *pass* or *fail* may be computed (or in the absence of an effective oracle a human software engineer may produce v_t. Subsequently the verdict (i.e. *pass* or *fail*) may be tagged to the input output pair resulting in a triple (i_i, o_t, v_t). TTCN-3 (see e.g [50]) also allows the verdict *none* (for use in case no verdict was obtained) and the verdict *error* (for use e.g. in case i_t was manifestly outside the range of relevant inputs for X).

I will refer to i_t as a test case, to (i_i, o_t) as a test sample (which is non-verdicted by default), to v_t as a verdict and to (i_i, o_t, v_t) as a verdicted test case. A verdicted test case is alternatively referred to as a verdicted test sample. For a test sample (i_i, o_t), i_t is the corresponding test case.

A test suite is a collection of test cases (meant to be used in connection with the same program). An effectuated test suite is a collection of test samples (usually obtained by running an ISuT on the different tets cases of a given test suite). A verdicted test suite is a collection of verdicted test samples (usually obtained by issuing verdicts on the test samples of an effectuated test suite). If an effectuated test suite V_{ets} has been obtained by running X on the test cases in a test suite, V_{ets} is called an effectuated test suite for X. If a verdicted test suite V_{vts} has been obtained by issuing verdicts relative to an oracle O on test samples form an effectuated test suite for X, then V_{vts} is a verdicted test suite for X relative to O.

Suppose V is a verdicted test suite for X. If X has been modified to X' by way of a local change which (hopefully) repairs a fault then a regression test on the verdicted test suite V consists of the set V' of verdicted tests obtained by running X' on underlying test case for each verdicted test case in V with verdict pass. A regression test is successful if all verdicted test cases in V' have verdict pass.

If a run involves interaction with the test configuration a trace of the interaction is plausibly included in the test case, so that such information can be used for the determination of a verdict.

Unfortunately the definitions as just outlined do not comply with TTCN-3 (see e.g. [50]). For instance in TTCN-3 a test case includes a verdict. Extracting from descriptions of TTCN-3 a very simple terminology which is helpful for theoretical work from first principles is complicated by the fact that TTCN-3 constitutes a realistic programming environment.

In practice test samples may come with more data than mere input-output pairs.

Definition 7.3 *(Test sample for X.) A test sample for inSeq X comprises a projection (and for that reason an abstraction) of a run of $TTC_\alpha[X]$ understood in terms of a preferred semantics of X. The projection may extract from α specific data including identification of the test case input for the run, and output (if any were produced); moreover said projection may involve the contents and ordering of one or more intermediate communications that were exchanged between X and the test configuration context $TCC_\alpha[-]$ during the run of $TCC_\alpha[X]$ as well as possibly other data that were collected during the run of $TCC_\alpha[X]$.*

7.2 A Programmer Driven Software Process

The terminology has been set up in such a manner that for instance a programmer (say agent B) may work as follows:

(step 1) create a prospective test suite V_{ts} consisting of test cases only, (not spending much time on how to arrive at verdicts, or on how to automate issuing verdicts via a suitable test oracle),

(step 2) B writes an instruction sequence X, (step 3) B puts into effect (i.e. runs) all tests cases in V_{ts} and thereby creates an effectuated test suite; when doing so B expects to acquire confidence in the quality of X (as an approximation of the final product which B has somehow in mind) and at the same time B expects to detect bugs ready for being resolved,

(step 4) B tries to issue verdicts for each of the tests samples in the effectuated test suite so obtained; if issuing verdicts for one or more of the test cases turns out to be problematic, then B writes (parts of) an informal specification of the intended behaviour of X which hopefully will suffice (for B) to issue verdicts for the entire effectuated test suite,

(step 5) B determines if the verdicted test suite gives rise to the suspicion of the presence of faults in X; if so B attempts to localise candidate faults (thereby performing an act of debugging) and to determine candidate changes which might substantiate the qualification of candidate faults as faults and the application of which will remove the various faults.

Preferably this work can be done in such a manner that effectuating those test cases again (now with modified X) the for which verdict *pass* had been issued, now lead (when run again on the underlying test cases) to test samples which deserve the verdict *pass* once more, thereby achieving a successful regression test on the given test suite.

(step 6) Determine whether or not further steps are needed such as:

(i) ask other programmers for comments and advice,

(ii) let other experts issue the verdicts,

(iii) write a more detailed specification so that an automated test oracle can be developed which then allows for more extensive testing, including automated random testing.

(step 7) (optional) Provide formal specification(s) and design one or more informal correctness proofs,

(step 8) (optional) Formalise proofs and perform proof checking.

(step 9) Declare X ready for use.

Remark. It is tempting to reject the listed software process for being naive. Preferably the work should start with writing a fairly rigorous specification and preferably the issuing of verdicts must be understood in advance of doing the programming and running the test cases. In the example above I allow for the situation that in the eyes of the programmer the program itself constitutes the best available (though perhaps in need of improvement) expression of its own specification, quite irrespective of whether or not maintaining such a perspective on the programming task is considered to be state of the art, to be encouraged, or to be discouraged.

7.3 Black-box Testing Versus White-box Testing (I)

An informal account of black-box testing versus white-box testing, together with an outline of a combined strategy, can be found in [81]. For testing4inSeqTh the first challenge is to obtain a definition of black-box testing. In Section 7.5 below a proposal is given to that end.

For testing4inSeqTh I will not adopt the widespread viewpoint that black-box testing must be based on the availability of a specification of the behaviour of a program. Following Definition 7.2 the testing process can go ahead without a specification of behaviour. It is the issuing of verdicts which requires a behavioural specification of the ISuT. More specifically issuing verdicts requires a test oracle.

The contrast between black-box testing and white-box testing is geared towards imperative programming where a computation may be imagined as a trip through the program text which for that purpose is somehow understood as a series of instructions.

White-box testing takes different forms, as described in the following items:

1. **White-box design of test castes**: in preparation of "testing X" one may develop a test suite V_{ts} meant specifically for use in connection with X. The design of V_{ts} can be done with or an awareness of the text (structure, architecture, design) of X. In that case we speak of white-box test suite construction.

2. **White-box meta-verdicting**: the text of X is used to produce meta-verdicts on the basis of an effectuated test suite, say V_{ets}. Here the underlying test suite V_{ts} may have been designed in a white-box, manner, or in a black-box manner (no knowledge of the text of X s used, or in a grey-box manner, use of partial knowledge of X.

This meta-verdict will express statistical, probabilistic, or possibilistic information about the various instructions, branches (test instructions paired with a specific Boolean reply), and paths through X which have occurred during the various runs the abstractions of which have been collected in V_{ets}.

The meta-verdict may but need not be used for a quality judgement regarding the underlying test suite V_{ts}. Typically meta-verdicting may lead to the suspicion that a part of X is unreachable, or to the suggestion that V_{ts} must be extended in order to increase coverage.

3. **White-box aware effectuation of test cases.** A run of X on a test case may be performed with information about the structure of X at hand. This may matter for instance if the run gets live-locked into an unending loop, upon which the run can be aborted. It may also matter if a run-time error causes a premature abort of the test run and the ability to track the path of the computation through X allows to determine the kind of error that was encountered and to report that fact in the test sample.

4. **White-box relevant test sampling**: in order to make full use of knowledge of the text of X once it is disclosed (which may be after effectuation of a test) in order to reconstruct the computational path of the test run it will suffice to deliver (and include in the resulting test sample) the sequence of successive replies to test instructions (here test refers to conditions evaluated during a run) tak were encountered during the test run.

   A test sample provides full path disclosure in hindsight if (given the text of X) the computation can be reconstructed step by step from the information that was collected in the test sample. For testing4inSeqTh I will assume that by providing the sequence of successive replies to method calls as made by X during a run, it is possible to reconstruct a computation path for X which clarifies which instructions were performed in what order and which branches were taken during a run. If such information is sampled full path disclosure in hindsight is guaranteed.

   White-box relevant test sampling has following advantages (upon X becoming known, before or after the test a re run and sampled).

   (a) it is possible to determine whether an instruction, a branch, a block or a path was performed during a test run, and in particular that

(b) no modifications of an instruction sequence need to be made in advance of the test run in order to guarantee (a),

(c) so that white-box meta-verdicting is possible on the basis of performing black-box tests (i.e. on obtaining a verdicted test suite from a test suite which was designed without knowledge of (the text of) X.

7.4 Black-box Testing Versus White-box Testing (II)

Black-box testing and white-box testing are often compared by way of either contrast or complementarity. I will now survey some of the arguments concerning these matters.

- An argument about black-box testing versus white-box testing reads thus: white-box testing allows an assessment of the quality of a test suite, whereas for black-box testing no such quality assessment exists.

  This argument is not obvious, because a black-box test suite may be considered "better" if it resembles better the expected usage profile (operational profile) of the ISuT. In the absence of information about a user profile the latter criterion is of no use. In [53] the distinction between reliability and dependability is emphasised. Reliability is about the probability of correct operation relative to a user profile, whereas dependability is independent of any specific usage profile. The independence is achieved by making dependability of a program local, that is a function of its inputs, thereby replicating the idea of correctness probability of [28].

- White-box testing is at best complementary to black-box testing when it comes to generating confidence (a notion akin to probable correctness as discussed in [54]) in the ISuT, simply because it is only the behaviour that really matters. The argument comes in different forms, for instance in Table 1 of [74] black box testing is mentioned as the preferred option for acceptance testing.

  Once a distinction is made between the confidence that the program works well (in terms of providing a solution to the problems which the program is supposed to handle), and the confidence that bugs have been found and removed to satisfaction, then the argument is convincing (upon assuming that primarily black-box testing creates the first type of confidence). Moreover, as an observation regarding the current practice of testing, the complementary role of white-box testing may be a valid observation. Remarkably both "program confidence" and "software confidence" are phrases with a minimal presence in the software engineering literature. "Software confidence level" is used, I saw no occurrence of "program confidence level". The literature on program testing provides no terminology for the positive information in terms of say confidence which program testing may create. Instead testing is often considered a verification method and the concept/term verification has a positive connotation.

Otherwise (if confidence is not split in two parts) the argument (that white-box testing would be complementary to black-box testing) is weak, as it misses the point that white-box testing may be necessary to obtain a sufficient level of the second type of confidence (i.e that the number of dormant faults is sufficiently low).

For testing4inSeqTh I adopt the notion of functional (instruction sequence) confidence (i.e. the confidence that the instruction sequence can produce helpful solutions as required), as well as the notion of fault freeness confidence (the confidence that faults are absent or infrequent). The competent programmer hypothesis of [38] may then be rephrased as follows: in the presence of functional confidence it is reasonable to expect that a failure is caused by a fault.

- Black-box testing does not require availability of the source code of X and can just as well be performed with an executable (a.o. mentioned in [74], and mentioned as the key advantage of black-box testing in [26]).

- Black-box test (case) design may be based on additional data that are unrelated to specifications: (i) a user profile, if available, may provide indication of what tests to perform, and (ii) an adversary profile may provide indications of the forms of misuse of the program which may exploit (and uncover) security problems; an adversary profile may support large scale automated random testing (fuzzing).

- Black-box testing is alternatively referred to as specification based testing because only a specification is taken as an input to the testing process, an also it is referred to as oracle based testing, following [76]. White-box testing when aiming at line coverage and branch coverage can do without an oracle.

- White-box testing, on the other hand, may be based on a risk assessment which takes the structure of X into account.

7.5 Test Runs of PGLA Instruction Sequences

I will now revisit the basic definition of inSeq testing, taking more detail into account than was done above in Paragraph 7.1. Consider the instruction sequence $X \equiv +f.m; \#3; +g.m2; \#3; h.m2; !; f.m; !$. The corresponding thread $|X|$ is as follows $P = |X| = f.m \circ ((f.m \circ S) \trianglelefteq g.m2 \trianglerighteq (f.m \circ S \trianglelefteq h.m2 \trianglerighteq D))$. When testing X in a black-box manner the test environment $CCT_\alpha[-]$ (as in Definition 7.2.) which I will denote with $C_{test}[-]$ is loaded with X thus obtaining $C_{test}[X]$. Then the test operator, say agent A, starts a run of $C_{test}[X]$: at first the test environment notifies to A the occurrence of a call $f.m$ which takes place without significant delay. The test environment can only notify the call but cannot process it properly, and therefore expects A to provide a reply which $C_{test}[X]$ can hand over back to X so that it can proceed (with the run that was started). If *true* is provided as a reply by A the the test run proceeds with the next episode as $C_{test}[\#3; +g.m2; \#3; h.m2; !; f.m; !]$, while if *false* is provided by A as a reply to the call $f.m$ the run proceeds as $C_{test}[+g.m2; \#3; h.m2; !; f.m; !]$.

After this brief informal introduction of the mechanics of a test environment I will describe in detail how a test environment can be understood for such instruction sequences, first for black-box testing, and thereafter for grey-box testing and for white-box testing. Grey-box testing lies in between black-box testing and white-box testing in that only partial information regarding the design and the structure of the ISuT is available to the tester. I will assume that all data are Booleans (single bits), under the assumption that many phenomena of relevance for testing can already be observed and studied in the presence of such trivial data only. For the service carrying a single bit I refer to [21, 23, 9]. A detailed account of testing ought to begin with the simplest case: a single instruction sequence is to be tested, no services are involved, no polyadic feature (multiple instruction sequences processed sequentially) no multi-threading, no distribution etc.

Arguably the simplest case is black-box testing of a single pass instruction sequence: no program counter needs to be maintained and no historical information needs to be stored and exchanged. Not maintaining a program counter matters because if a bounded program counter would be provided that state of affairs by itself uncovers information regarding an upper bound of the number of instructions of X, a figure named LLOC(X) for logical lines of code in [9]. As it turns out, making LLOC(X) available to S constitutes a significant step towards grey-box testing already.

I will first specify in detail a mechanism for testing a single pass instruction sequence X. Here X involves no backward jumps, but it may be finite or infinite, in which case it is repeating the same finite instruction sequence indefinitely. In spite of the simple program notation, PGLA of [16], the notation available in the PGA family, the amount of detail involved in a precise definition of testing is considerable.

By assumption U is a possibly infinite set of all conceivable method calls in the program notation at hand. Subsets of U serve as interfaces for threads and service families. A thread has a required interface, as it expects its method calls to be processed by an environment, while a service family has a provided interface as it provides the option for processing method calls. For more information on interfaces see [9].

A component $C^U_{test}[-]$ is assumed to be available which allows the following repertoire of actions and observations, given a thread P with required interface $I_{required}(P) \subseteq U$:

- I will assume the presence of an agent A who acts as the operator of a test. A can perform the following actions in connection with $C^U_{test}[-]$ and X.

  (i) load X (as an ISuT) in $C^U_{test}[-]$, thus obtaining $C^U_{test}[X]$,

  (ii) start the test run,

  (iii) abort the run when, according to A, it is taking too much time,

  (iv) read off the status $terminated{-}ok, terminated{-}nok$, $waiting{-}for{-}reply$, $aborted$, upon termination and,

  (v) supply a Boolean reply (by pushing either button r_{true} or r_{false}) if $C^U_{test}[X]$ has terminated in a $waiting{-}for{-}reply$ condition, and then restart the test from the state where it has been left starting to wait for a reply (to be provided by the test environment).

- The testing process may only use information regarding the external behaviour of X, i.e. about $|X|$. This restriction excludes information about:

(i) an upper bound for LLOC(X),

(ii) whether a method call is void, or is included in a positive test or in a negative test,

(iii) information concerning $I_{required}(X)$ which is accessible via static inspection of S only. It is merely known that $I_{required}(X) \subseteq U$.

- When started $C^U_{test}[X]$ computes and the computation may either go on forever, or may end in one of the following three states which can be read off from $C^U_{test}[-]$

  (i) $s_{terminated-ok}$ indicating that the computation has properly terminated,

  (ii) $s_{terminated-nok}$ indicating that the computation has improperly terminated,

  (iii) $s_{waiting-for-reply}$ indicating that the computation has made a method call (i.e. not involving a test) say at the n-th instruction of X (n not known to A), and is then waiting for the test operator A to provide a reply for that call in order to continue with the next episode of the test run.

  During the run the visible sign (visibility for A is meant) of the status of the test is "*running*".

- If a computation takes too long (according to A), A may abort the computation without any form of proper or improper termination having been achieved. Subsequently the instruction sequence is unloaded, and the same or another instruction sequence can be loaded in order to proceed with testing (using the same context, i.e. testing equipment).

- In case $C^U_{test}[X]$ is waiting for a reply after having observed the method call say $f.m$, either the run is aborted (ending up in state *aborted*) because A is unable to provide a reply, or because A considers the test run taking too many steps, or alternatively one of two buttons may be applied by: r_{true}, and r_{false}. Upon that having been done, the test run proceeds with the next episode, which works in detail as follows: let $X \equiv u; Y$ where $u \in \{f.m, +f.m, -f.m\}$ and where Y is a possibly empty instruction sequence. The instruction u is half-way of being processed. Now 6 cases are distinguished:

(i) if Y is empty then the run improperly terminates (i.e. with behaviour D) which is made visible to A via the sign *terminated−nok*, thereby ending both the test episode and the over-all test run.

Otherwise it may be assumed that Y is nonempty, and that $Y \equiv v; Y'$ with v an instruction, and where, if Y' is non-empty $Y' \equiv w; Y''$ with w an instruction.

(ii) if $u \equiv f.m$ (i.e. a void method call), then (irrespective of the reply that A has provided), the test run proceeds with the next episode from state $C^U_{test}[Y]$.

(iii) in case $C^U_{test}[X]$ is waiting for a reply after having called a positive test ($u \equiv +f.m$), and the A has provided reply *true* (with button r_{true}), the test continues with the next episode from state $C^U_{test}[Y]$.

(iv) in case $C^U_{test}[X]$ is waiting for a reply after having called a positive test ($u \equiv +f.m$), and A has provided reply *false* (with button r_{false}) then: if Y' is empty the test ends in state *terminated−nok*, while if is non-empty the test resumes with the next episode from state $C^U_{test}[Y']$.

(v) in case $C^U_{test}[X]$ is waiting for a reply after having called a negative test ($u \equiv -f.m$), and the A has provided reply *false*, the test continues with the next episode from state $C^U_{test}[Y]$.

(vi) in case $C^U_{test}[X]$ is waiting for a reply after having called a negative test ($u \equiv -f.m$), and A has provided reply *true*, then: if Y' is empty the test ends in state *terminated−nok*, while if is non-empty the test resumes with the next episode from state $C^U_{test}[Y']$.

As it turns out it is implausible to assume that the operator will work under the assumption that (their interaction with) the test environment is 100% deterministic. The very fact that A sometimes needs (or will wish) to perform a time-out introduces some form of non-determinism. Some assumptions are needed to clarify what happens in detail when termination or divergence takes place.

1. $C^U_{test}[!; X]$ will end quickly in state $s_{terminated-ok}$.
2. $C^U_{test}[\#0; X]$ will end quickly in state $s_{terminated-nok}$.
3. After a jump out of range the run will end quickly in state $s_{terminated-nok}$.

4. Jump instructions are performed quickly at a predictable pace, and even (writing $X = Y; Z^\omega$) performing as many consecutive jumps as $X; Z$ has instructions (there can be no more consecutive jumps without ending up in a deadlock) takes so little time that there will be no incentive for A to abort the ongoing run.

5. An infinite sequence of jumps, which occurs for instance when running an first episode for $(\#2; +g.m1)^\omega$, will take very long so that A must eventually abort the run, (the resulting livelock may not be recognised by the test environment assuming that testing is done in a black-box manner). In this case, with the help of A, a state $s_{aborted}$ is reached after some time, thereby ending the test run.

Single pass instruction sequences can be transformed to instruction sequences which determine the same thread in such a manner that all method calls are positive tests. If one assumes that void method calls and negative test method calls do not occur the above definition becomes simpler.

7.6 Test Case Versus Test Sample

The notion of a test case must be adapted to the fact that inputs to the computation now include the sequence of boolean replies. A difficulty is that in advance of the run it is not clear how many replies on various method calls will need to be provided.

Definition 7.4 *A test case (in the case of X in PGLA) consists of a finite or infinite eventually repeating sequence σ of Boolean values. The sequence σ contains the successive replies that A is supposed to provide during a test. If σ is too short the test run will end in terminated−nok.*

For the special case of single pass instruction sequences the notion of a test case can now be made more specific.

Definition 7.5 *A test sample (for X in PGLA) consists of a sequence of one more successive episodes, where except for the first and the last episode, each episode is characterised by the pair of a method call (as issued by X during its run), and the corresponding reply (as provided by the test operator A). The operator A provides a Boolean reply and then restarts the run for the next episode which consists of zero or more jumps followed by a method call or one of both options for termination. A test sample is l-bounded if at most l episodes occur. The final episode encodes that status in which the test run has ended.*

According to these definitions the ISuT behaves like a deterministic reactive system. Such systems are amply studied in automaton based testing theory.

Remarks. Several remarks are in order:

(i) Testing (i.e. running an inSeq on a test case) already makes sense without an oracle being available.

(ii) The test environment is not a service as defined in e.g. [20]. The test environment may have an infinite interface (if U is infinite) and there is some essential interaction with an operator (named A above).

(iii) It is a matter of taste whether or not the test environment is considered non-deterministic.

(iv) Providing to A or to the test environment a finite interface J such that $I_{required}(X) \subseteq J \subseteq U$ already brings one into the realm of grey-box testing. Indeed it is not possible by merely running a finite number of tests to rule out the option that some method call may eventually occur.

This can be seen as follows: suppose $J = \{f.u, f.w\}$, and assume that the property $\psi(-)$ of X which is to be confirmed by way of testing is that no method call $f.w$ will ever take place. Let $X = (+f.u; !; \#1)^{\omega}$ and $X_n \equiv (+f.u; !; \#1)^n; f.w; !$. Clearly $\psi(X)$ holds but for arbitrary n, $\psi(X_n)$ is not valid.

Suppose that at most k test runs are admitted each of which are l bounded (under these conditions I will speak of (k, l)-bounded black-box testing), then such tests are unable to distinguish X from X_{l+1}. Apparently using (k, l)-bounded black-box testing it cannot reliably be excluded that a failure in the form of a call of $f.m$ will occur.

7.7 Completeness of Black-box Testing for Single Pass Instruction Sequences

In some sense black-box testing is complete, at least in principle. The following Proposition expresses that fact. I will omit the elementary proof which is based on (i) the observation that no path between consecutive method calls involves more than n steps, (ii) all states of $|X|$ are reached with runs with at most n episodes.

Proposition 7.2 *(Conditional completeness of black-box testing.) If an upper bound of the number of states of* $|X|$ *is known in advance, say* n *then* $(2^{2\cdot n+1}, 2\cdot n+1)$ *bounded testing suffices to determine the thread* $|X|$ *in all detail (which includes the exact number of states of* $|X|$*).*

As an immediate consequence of this observation it follows that:

Proposition 7.3 *Presenting the testing agent* A *with an upper bound* n *of LLOC(X) already moves its testing capability outside black-box testing and into the realm of grey-box testing.*

In practice, however, testing will be (k,l)-bounded for some k and l known in advance of the first practical use of X and it is a common assumption in testing that k and l will be small in comparison to 2^n, with n any known upper bound for the number of states of $|X|$ so that an application of Proposition 7.2 is not understood as being a part of testing theory. I will make that assumption for testing4inSeqTh as well.

Related literature. If one conceives of testing as leading to positive conclusions, the information theoretic approach of [90] is quite related, to the above definitions. A principled approach to black-box testing can be found in [37]. Proposition 7.2 is nothing new in mathematical terms. Proposition 7.2 provides a rephrasing of facts about automaton theory known since [73], see also [72] for more information in a setting of black-box testing and machine learning. The phenomenon of test cases in a black-box testing which for some reason do not run to completion (as taken into account by A's option to abort an episode of a test run) is investigated in detail in [30].

8 Auxiliary Data, Input and Output

The definitions about instruction sequence testing give thus far are somehow complicated by the fact that at the lowest level instruction sequences denote threads which feature interactive behaviour rather than functions on data which can be appreciated without taking interaction into account.

8.1 Taking Auxiliary Services into Account

Let $H(b)$ be a single bit service (as detailed in [9]), with memory state $b \in \{0,1\}$. An instruction sequence X may produce a behaviour by using a service family say $f_1.H(0) \oplus \ldots \oplus f_n.H(0)$ as auxiliary services. Now $P \equiv |X|/(f_1.H(0) \oplus \ldots \oplus f_n.H(0))$ denotes the thread produced by X where all calls $f.m$ with a focus $f \in \{f_1, \ldots, f_n\}$ are to be processed by $f_1.H(0) \oplus \ldots \oplus f_n.H(0)$, A method call $f.m$ with $f \in \{f_1, \ldots, f_n\}$ is not visible outside the configuration (thread) $|X|/(f_1.H(0) \oplus \ldots \oplus f_n.H(0))$ and therefore it is not a required method P while in case $f \notin \{f_1, \ldots, f_n\}$ a method call $f.m$ is in the interface of P, so that during a test the operator A must provide a reply.

Black-box testing of P will make use of a testing environment as before while only asking the operator for a reply to method calls $f.m$ for $f \notin \{f_1, \ldots, f_n\}$. Testing is now taking place in a setup of the form

$$C^U_{test}[|X|/(f_1.H(0) \oplus \ldots \oplus f_n.H(0))]$$

The given definitions of test case and test sample can be used in this case as well. There are several differences however: (i) a run can get into a loop while interacting with the auxiliary services only. To decide whether or not that will happen is NP-hard (see [13]); it is the task of A to determine how long to wait before a run is aborted. (ii) Because the system is deterministic once X is known as a text, a test case suffices for computational path reconstruction.

In these circumstances a straightforward analogue of Proposition 7.2 stated in terms of the size of the state space of $|X|$ fails and the size of the state space of $f_1.H(0) \oplus \ldots f_n.H(0)$ (i.e. 2^n) enters the picture as well.

8.2 Dealing with Inputs and Outputs

If inputs and outputs are present then these are given as the states of various services. In the simplest case n input bits are transformed into as many output bits, represented by the respective states of services upon termination of the computation. Now it is plausible to assume that $J \subseteq I_{provided}(f_1.H(0) \oplus \ldots \oplus f_n.H(0))$, i.e. that each method call is of the form $f.m$ with $f \in \{f_1, \ldots, f_n\}$ and $m \in I_{method}(H(0)) = I_{method}(H(1))$. An expression for what is computed during a run now reads as follows:

$$C^U_{test}[|X| \bullet (g_1.H(b_1) \oplus \ldots \oplus g_m.H(b_m))]$$

In such computations the operator will not be asked to provide replies for any method calls. If a run takes too long the operator may have no other choice than to abort the run.

Definition 8.1 *A test case for X consists of a sequence $(b_1, \ldots, b_m)$ of initial values for the respective copies of service $H(-)$.*

Definition 8.2 *(Test sample with input and output.) A test sample for X consists of:*

(i) *a sequence $(b_1, \ldots, b_m)$ of initial values for the respective services,*

(ii) *a flag $s_{terminated-nok}$ in case the run terminates improperly,*

(iii) *a flag aborted or in case it was aborted by A,*

(iv) *a flag $s_{terminated-ok}$, if valid termination took place, together with final values $(b'_1, \ldots, b'_m)$ of the Boolean services which now contain outputs rather than inputs.*

Test cases according to Definition 8.2 disclose behavioural information only and allow path reconstruction so that white box verdicting is possible for an effectuated test suite consisting of such tests.

8.3 Combining Inputs, Outputs and Auxiliaries

Inputs, outputs and auxiliaries can be combined, leading to a configuration of the following form.

$$C^U_{test}[(|X|/(f_1.H(0) \oplus \ldots \oplus f_n.H(0))) \bullet (g_1.H(b_1) \oplus \ldots \oplus g_m.H(b_m))]$$

Now it is assumed that $J \subseteq I_{provided}(f_1.H(0) \oplus \ldots \oplus f_n.H(0)) \cup I_{provided}(g_1.H(B_1) \oplus \ldots \oplus g_m.H(b_m))$. Writing $H_{aux} = f_1.H(0) \oplus \ldots \oplus f_n.H(0))$ and $H_{in/out}(\vec{b}) = g_1.H(b_1) \oplus \ldots \oplus g_m.H(b_m))$ one finds the configuration:

$$C^U_{test}[(|X|/(H_{aux})) \bullet H_{in/out}(\vec{b})]$$

Apparently the notion of a test run becomes increasingly more complicated with increasing complexity of the test configuration.

8.4 Taking Multi-threading into Account

The situation becomes more complicated if X is one of a series of threads which is combined by way of a strategic interleaving operator.

$$C^U_{test}[||_c(\langle |X_1|/H_{aux,1}\rangle ^\frown \ldots ^\frown \langle |X_k|/H_{aux,k}\rangle ^\frown \langle |X|/H_{aux}\rangle) \bullet H_{in/out}(\vec{b})]$$

Here for simplicity it is assumed that the service families H_{aux}, and $H_{aux,1}$, $\ldots, H_{aux,k}$ pairwise provide disjoints method interfaces.

There may be various synchronisation primitives involved (I refer to [17] for options for that) in strategic interleaving operator, which may also be more sophisticated than mere cyclic interleaving.

Defining a test sample is becoming a challenge. The simplest idea is as follows:

Definition 8.3 *(Test sample with input, output, local auxiliaries, and deterministic multi-threading.) A test case for X consists of*

(i) a sequence $(b_1, \ldots, b_m)$ of initial values for the respective services $H_{in/out}(\vec{b})$,

(ii) a sequence $r_1, \ldots, r_k)$ of Boolean replies that were returned to all method calls in the order of occurrence by each of the threads until either proper or improper termination took place, or the run was aborted by A,

(iii) a flag $s_{terminated-nok}$ in case the run terminates improperly,

(iv) a flag aborted in case the run was aborted by A, or in case it was aborted by A,

(v) otherwise a flag $s_{terminated-ok}$ together with final values $(b'_1, \ldots, b'_m)$ of the services $H_{in/out}(-)$ which now contain outputs rather than inputs.

The notion of a test sample according to Definition 8.3 is probably best characterised as grey-box rather than black box. That all replies to method calls made to the auxiliary services are logged in a test sample can hardly be understood as "black-box compliant" : more information about how the computation of X works is collected than the absolute minimum. However, it is the case that path reconstruction can be done with this information. I conclude that for an inSeq that determines a thread in a deterministic concurrent system it is not obvious what it means (or requires) for a testing to be qualified as black-box.

8.5 Revisiting General Program Testing Theory

General program testing theory testing is briefly discussed in the concluding Section. The work of Gourlay, in particular [49], may serve as a point of departure. Although it is stated in [49] that program and specification stand side by side and that no implicit assumption is made that specification precedes programming or the design of a test battery (i.e. test case suite), the separation of concerns which is achieved in the formalisation given by Gourlay seems to introduce a dependency which I would prefer to avoid. The approach of [49] rests on the notion of a testing system which takes the form of a quintuple

$$(U_{program}, U_{test}, U_{specification}, corr, ok)$$

Here $U_{program}$ is the collection (using U for universe) of programs under consideration, U_{test} is the collection of tests under consideration, and $U_{specification}$ is the collection of specifications under consideration. Initially there is no need to be more specific about these three collections, unless examples are discussed. Further $corr \subseteq U_{program} \times U_{test}$ (elsewhere often written as X $\underline{\texttt{sat}}$ s) is a relation which determines whether or not program X satisfies specification s, and $ok \subseteq U_{test}, U_{program}, U_{specification}$ is a relation (with $ok(t, X, s)$ written as X ok_t s). The relation ok represents a plurality of mechanisms, (i) the issuing of verdicts after a test run on an individual test case has been competed, (ii) the use of an oracle with or without a specification as input to generate a verdict, universal quantification over the verdicts made for the test cases in a test battery (pass if all test individual cases were passing). Now by packaging in a single relation aspects of programs as well as of verdicts the impression arises that verdicts are somehow related to programs, which in my view is not the case.

Assuming that an oracle produces verdicts, where the oracle may involve, i.e. make use of, computing as well as of human judgement, there is no need to view the oracle (or the "current verdicting process") as a stable and static mathematical entity. The oracle produces additional information once a program (inSeq) has been run to completion (a non-trivial notion unfortunately) and has produced one or more successive outputs, while or after having received one or more successive inputs. The additional information is about how the current program X behaves on certain inputs. Now this idea is consistent with the suggestion that the current program at the same time embodies the best known written description of its own specification. New information coming about from the test, however, may provide an incentive for the programmer to revise their views on the matter and may for that reason create an incentive to modify the program.

It follows from this idea that neither a concept of a specification S as a mathematical entity, nor a satisfaction relation X $\underline{\texttt{sat}}$ S need to be present at any stage of the process of program construction (for X), and that some form of ad hoc and incremental specification by need may suffice. The verdict serves allows to introduce other information, from outside the program and from outside the programming activity, into the program construction process and such information does not depend on the program. The verdict may depend on a specification if that is available and has been made accessible via an oracle, but that need not be so.

8.6 Incorporating General Testing Theory in testing4inSeqTh

The general theory of testing can be instantiated for inSeqTh thereby including more detail in order to arrive an a more comprehensive picture. A testing system (for testing4inSeqTh) consists of the following elements:

1. U_{inSeq}: a collection of inSeq's which is considered as potential outcomes of a programming process,
2. optional: $U_{specification}$, a collection of specifications for inSeq's (which may contain no more than the empty specification ok which is always satisfied),
3. a relation $\_ \ \underline{\mathsf{sat}} \ \_ \subseteq U_{inSeq} \times U_{specification}$,
4. a collection T^{in}_{case} of possible inputs for individual test runs (which may include a succession of messages sent to X (while running) by the test environment),
5. a collection T^{out}_{sample} of possible outputs for individual test runs, (which may include a succession of messages sent by X (while running) to the environment),
6. T_A the test agent, who is in control of the test environment,
7. A set T_{flags} consisting of $\{termination{-}ok, termination{-}nok, interrupted, aborted\}$; alternatively one may write $interrupted{-}ok$, $interrupted{-}nok$ for the pair $interrupted, aborted$. Interrupts and aborts are made by T_A where an interrupt takes place if sufficient information has been accumulated for delivering a verdict, and an abort is enacted if the runs seems to have live-locked or to have deadlocked (though without the certainty needed to raise the flag $termination{-}nok$.

8. $T_{sample} = T^{in}_{case} \times T^{out}_{case} \times T_{flags}$, which represents the collection of conceivable test cases,

9. a function $\mathsf{perform}\colon U_{inSeq} \times T_{case} \to T_{sample}$; this function formalises the act of testing in terms of what it produces of relevance for testing (with more sophistication time, energy, and other resources may be include as inputs as well),

10. a collection $T_{verdict} = \{pass, fail, none, error\}$ which constitutes possible verdicts,

11. $T^{verdicted}_{sample} = T_{sample} \times T_{verdict}$, the collection of possible verdicted test cases,

12. a (total) function $\psi_{oracle}\colon T_{case} \times T_{specification} \to T_{verdict}$; in case no verdict is made the result is *none*, (the intuition is that ψ_{oracle}, may be just as much, or even more, a moving target as X, if X is in the role of an ISuC, i.e. an instruction sequence under construction),

13. $T_{battery} = 2^{T^{in}_{case}}$, (test battery, often referred to as test or as test suite),

14. $T^{case}_{suite} = 2^{T_{case}}$,

15. $T^{verdicted}_{suite} = 2^{T^{verdicted}_{sample}}$,

16. a pointwise extension of the oracle function to test suites $\psi_{oracle}\colon T_{suite} \to T^{verdicted}_{suite}$,

17. requirement: for all $X \in U_{inSeq}$ and $t_{in} \in T^{in}_{case}$ and for each $s \in U_{specs}$:

$$X \ \underline{\mathsf{sat}}\ s \to \psi_{oracle}(\mathsf{perform}(X, t_{in})) = pass$$

Having the terminology as listed above available the classical theory of testing can be redesigned in more detail. Whether or making that redesign will be a fruitful endeavour remains to be seen. Using the terminology just outlined, several remarks can be made:

(i) Suppose programmer B is developing instruction sequence X with a possibly incomplete and possibly invalid specification $t_{specification}$ at hand, and at some stage B wishes to perform a test on the basis of test case t^{in}_{case} then this can be done, assuming that X has advanced to the point of being runnable, and while B plays the role of the test agent T_A. Then the test sample $t_{sample} = (t^{in}_{case}, perform(X, t^{in}_{case}))$ will result say with termination flag $termination{-}ok$. Next a verdict can be made resulting in $\psi_{oracle}(t_{sample}, t_{specification})$ with verdict $fail$.

Now B has two choices: either to modify X, say to X', so that on the input for test case t_{case} another test case is generated for which the verdict is *pass* is obtained, or alternatively to modify ψ_{oracle}, say to ψ'_{oracle}, now with the intention to obtain so as to obtain *pass* for the same test case i.e. for $\psi'_{oracle}(t_{sample}, t_{specification})$. The testing4inSeqTh account of testing involves no preference on said choice.

(ii) In [9] the notion of a fault is contemplated in detail, and following [61] for a fault to serve as an explanation of a failure a change needs to be known which resolves the failure. More specifically a justification is needed that a change serves as (a possible) remedy for a problem (the occurrence of a failure). Three forms of justification are discussed: leading to specialised notions of fault: (1) Laski fault (following [61]) where the change turns the program in to a provably correct program, (2) a MFJ-fault (following [69]) where the change as applied to X leads to a program X'which fails on fewer inputs (or more generally, which behaves better in whatever metric for conformity of program behaviour with the specification one wishes to use), (3) an RTJoC fault (Regression test based justification of change), where it is required that then changed program still passes the test suite (consisting of successful tests only) which has thus far been obtained for X. Now the phrase "test suite" was used in [9] but in the light of the terminology just outlined a more precise wording can be found.

Now assume that the available test suite consist of (successfully) verdicted test samples only. In this case all verdicted test samples are stripped from outputs, flags and verdicts, so that a test battery remains. Then a test with X' is performed for every test case in the new test battery, and a new test suite (collection of test samples) results. These are all inspected and augmented with a verdict such that a new suite of verdicted test samples results. How the requirement on regression testing reads that all verdicts are positive (i.e. *pass*).

Thus regression testing is a matter of performing test runs on a test battery which is obtained by taking the test cases out of the verdicted test samples which constitute the successful part of the testing history of the project for designing X thus far.

(iii) One might claim that both remarks (i) and (ii) move the subject from testing to debugging, where testing would be done with a specification at hand, and with a preference for adapting the program to the specification rather than conversely, while debugging may not even take any specification into account. The vast literature on program debugging seems to share with the literature on testing, as well as with the equally large literature on bugs, that no definition of a bug is provided, or is imported via a reference to earlier work. Interestingly the literature on debugging takes for granted that a bug is localised in the text of a program somehow, and that debugging has two dimensions: finding the bug, and repair of the bug. In the literature on bugs, however, bug is more general and bug reports are plausibly about failures rather than about fragments of programs. Although debugging seems to lie at the very practical end of the subject of program quality, it also opens a door to a theme which seems to be out of reach of program testing: debugging may shed light on the fundamental phenomenon of unrealisability of a specification, see e.g. [57].

9 Concluding Remarks

Many approaches to and perspectives on program testing have been left untouched in the paper thus far. I will briefly discuss some of these matters in the concluding section. Finally I will provide an example of the notion of partisan ambiguity from elementary arithmetic.

9.1 Conditions on Test Batteries.

Theoretical work on testing started with [48]. A brief summary in slightly adapted notation reads as follows. One considers a program X which determines a function $[\![X]\!]$ from D to D'. A test case is an element of D. It is assumed that verdict function $OK\colon D \times D' \to \{pass, fail\}$ is known. A test battery T is a subset of D. Test battery T is successful for X if for all $d \in T$ it is the case that $OK(d, [\![X]\!](d))$. A notion of X-completeness C is introduced as a predicate on test batteries, i.e. as subsets of 2^D. I have added "X-" to the notation in order to highlight that the notion is specific for X. Now $X \subseteq 2^D$ is X-complete of it enjoys the following property: for each test battery $T \in C$, if T is successful for X then X is failure free (i.e. for all $d \in D$, $OK(d, [\![X]\!](d))$, or stated differently: D, the so-called exhaustive test battery, is valid for X).

Now notions of X-validity and X-reliability can be defined for C as follows: C is X-valid if whenever X is not failure free there must be a test battery $T \in C$ (that is an X-complete test battery) which is not successful for X. C is X-reliable (as a completeness criterion for X) if either all test batteries contained in C are valid for X or alternatively no test battery in C is successful for X, i.e. it does not matter which T battery in C is used for X. Then the following result is shown:

Theorem 9.1 *(Goodenough & Gerhart, 1975). Given an X-valid and X-reliable completeness criterion C, if $T \in C$ and T is valid for X, then X is failure free.*

These notions are geared towards the option that testing provides complete certainty, that is as much certainty as exhaustive testing would bring, if doable. The practical problems with application of these notions are clearly explained. The paper then continues with an exposition of the so-called condition table method for testing, which can be seen as a precursor of later equivalence class based methods, where instead of exhaustive testing for each equivalence class of D testing a single test case in it suffices. Later Weyuker & Ostrand [89] discover that X-validity and X-reliability are not independent notions and that instead for each program X and each completeness criterion C, C is X-valid or C is X-reliable. This observation constitutes a limitation of the notion of X-completeness which motivates the introduction of another qualification for a criterion viz. the notion of a criterion being (X, S)-revealing for a subset $S \subseteq D$. The criterion $C \subseteq 2^S (\subseteq 2^D)$ is (X, S)-revealing if and only if it enjoys the following property: if X fails on S (i.e. for some $d \in S$, $\neg OK(d, [\![X]\!](d))$) then for each test battery $T \in C$, X is not successful on T. It is then shown by example that the new meta-criterion covers more cases than X-reliability.

Interestingly [48] comes close to arguing that testing cannot always be replaced by formal proofs, though such is not stated as a theoretical fact. I will now look into that matter more closely. I will assume that single pass instruction sequence X and data $d \in D$ are mathematical entities. The notion that program X can be put into effect on hardware M with input $d \in D$ amounts to the idea that a hardware configuration $M[\langle X \rangle, \langle d \rangle]$, with physical components $\langle X \rangle$ and $\langle d \rangle$ supposedly representing X and d produces after some processing time next state of itself which is adequately described as $M[\langle \star \rangle, \langle d' \rangle]$ for d' in D' where $\star$ represents whatever data (including X) as the program may have been dropped by being put into effect (which is plausible for a single pass instruction sequence). As an assertion in physics rather than in the theory of computing consider the following hypothetical proposition:

Proposition 9.1 *(Assertion in physics.) Running $M[\langle X\rangle, \langle d\rangle]$ leads to $M[\langle\star\rangle, \langle d'\rangle]$.*

Proposition 9.1 would require for its validation repeated tests runs with the same inputs. Validity of Proposition 9.1 would at best known with a probability close to 1 but yet different from 1. Even if formal verification of X w.r.t. some formal model fm_M of $M[-,-]$ has been successfully completed, that state of affairs will not guarantee Proposition 9.1 with a certainty of 100%. In fact experiments intending to validate or falsify Proposition 9.1 (or similar assertions) may be performed in order to validate the formal model of $M[-,-]$, and the very assertion that fm_M provides a satisfactory model of $M[-,-]$ will involve probabilities and statistics. These considerations may not matter much for computation on a discrete domain, but if timing is essential, reading of devices is needed and handling of actuators takes place, and if requirements are phrased in terms of expectations of the behaviour of "real hardware" then the relevance of "physical assertions" like Proposition 9.1 increases.

In order to remove a possible role of physical experimentation theoretical work on testing it would be helpful to make the assumption that testing is done on the basis of simulation of computations on a formal machine model, and that verification is done (i.e. proof checking) w.r.t. the same formal machine model, and that both are done on the same or comparable hardware.

9.2 Instruction Coverage and Branch Coverage

Coverage is a magic word in program testing. In [56] a survey of 101 forms of coverage is documented. The author indicates that the plurality of forms of coverage renders it unavoidable that in a software project choices are made. Not all forms of coverage are achieved or aimed at, and there is no easy way to prioritise. The relevance of achieving various forms of coverage is non obvious: for instance in [86] the question is posed to what extent branch coverage shows a high correlation with fault detection. Mixed conclusions are drawn from inspecting a significant body of testing outcomes.

Definition 9.1 *Let $X \equiv Y; Z^\omega$. During a run of X an instruction u in Z is performed if at least one of its unfolded versions is performed. A testing* *process ψ_{test} delivers p% instruction coverage if for p% of the instructions of X (i.e. the instructions of Y combined with the instructions of YZ) are performed during at least one of the runs (as generated by testing process ψ_{test}) which end with verdict pass.*

Definition 9.2 *Let inSeq X contain k test instructions. A suite of test samples for inSeq X of size n (the number of test cases), delivers p% branch coverage if p% of the $2 \cdot k$ pairs of method call and corresponding Boolean reply occur during at least one of the runs of ψ_{test}.*

Given X it may be very difficult to design a test process which achieves 100% instruction coverage as well as 100 % branch coverage. It is reasonable to ask the programmer of X to provide such test cases, however, because by doing so some evidence is provided that none of the instructions and none of the tests in X are redundant. Such information should be available to the designer at low cost, and a customer of software who intends to grasp how it works should be entitled to such information.

9.3 Formal Program Testing

Formal program testing was proposed in [33]. The idea is simple: try to prove a program correct and one ends up with a number of verification conditions for which a formal proof is needed. That may be a problem (which it was in 1981), and the idea is that besides computer tools which simplify these conditions testing may be used to obtain confidence in said verification conditions.

Formal program testing approaches formal verification with a mathematical mindset on what constitutes a proof and at the same time a testing mindset on how to deal with so many details. Formal program testing can be adapted to instruction sequences without any problem. As such the idea is quite attractive because it allows a stepwise method where the precondition is made weaker along with the postcondition and the proof is improved at each step, until perhaps a failure prevents further weakening of the precondition. If for a program P a test case a (i.e. the input) creates a run which produces test sample (a, b) and a verdict $v(in = a, out = b) = pass$ is issued, then $\{in = a\}P\{out = b\}$ is a valid partial correctness assertion, for which one may look for a proof, and the verification conditions thus found may be made plausible by means of tests.

9.4 Further Theories of Testing

Gourlay [49] provides a comprehensive analysis for various proposals for a meta-criterion i.e. a qualification of a criterion such as it being X-reliable or (X, S)-revealing. Almost 40 years later this paper is still state of the art on the subject of general (program) testing theory.

Bougé [31] surveys a history (in 1985) of program testing theory and the author concludes that the theory of program testing is about black-box behaviour and focuses on various criteria and interrelations of test batteries. With reference to original work on criteria for test sets of [48] Bougé writes:

> It must in fact be split into a transcendental part (proving ideality of the criterion and adequacy of the test) and a calculable part (running the test cases). The most important feature of the proposal is, to our mind, the notion of a criterion and the properties which are defined about it. Notice that passing any test satisfying an ideal criterion demonstrates the correctness.

The idea that testing works towards confidence is reflected in this quote:

> Any testing theory should focus rather on success of tests than on failure, because only the former is actually informative.

The idea that testing activity comes with a transcendental part is quite useful for bridging the gap with formal verification. However, when scanning the literature on program testing said transcendental part seems not to play a role of any significance until now.

9.5 The Relevance of Testing

Lipton [63] writes:

> Almost all real computations use testing to insure their correctness.

and

> Further testing is usually viewed as a method of discovering errors, not as a method of proving the absence of errors. Indeed the main appeal of formal verification is that it has the potential to yield "proofs" that software is error free. Clearly testing would be more valuable if it led to provable statements about programs.

Here I read error free as “leading to failure free behaviour”. Further testing is positioned as a second best option for acquiring confidence, given the inherent difficulties of (formal) verification, while at the same time in practice testing constitutes a search for failures.

Remark: The apparent contradiction that testing “is” looking for failures and faults, while at the same time testing is considered a part of verification, that is working towards positive conclusions, can be overcome if one admits informal negation by failure as a reasoning pattern for software quality. In the medical profession reasoning on the basis of negation by failure is standard practice: if a certain effort to uncover a class of potential problems yields nothing, i.e. yields a negative outcome, then one proceeds as if said problems are absent. For programs a similar attitude is possible: rather than to categorically state the absence of certain problems, one proceeds under the hypothesis that such problems have a low probability, and as if these are absent, comparable to boarding a plane. Even a proof does not provide 100% certainty (as the proof might be wrong, or a proof checking might fail by producing a false positive, or the underlying specification may be invalid etc.), so I it seems that there is no unbridgeable categorical gap between testing and formal verification.

9.6 Restricting the Scope of Program Testing

Gaudel [44] writes:

> First, let us make more precise the background of the works reported here. The aim of a testing activity is to detect some faults in a program. The use of the verb detect is important here, since locating and correcting the faults are generally regarded as out of the scope of testing: they require different data and techniques.

The notion of a fault is left undefined in this work.

Remark: Gaudel appears to refer to a failure (rather than a fault, i.e. an ALR fault) in the sense of the program producing an invalid result on certain inputs. Presumably for for Gaudel spectrum based fault localisation, which locates faults that supposedly cause one or more failures of which the existence is known, does not classify as an activity in testing.

In any case, restricting the scope of testing is a path chosen by many authors in order to make the topic accessible for fundamental research.

9.7 Risk Based Testing

Aktaş et. al. [1] discuss risk based testing (RBT). The approach of RBT has a significant tradition already and is aiming at the acquisition of confidence based on software specific risk analysis which them guide the creation of test cases. In [39] a difference is made between the use of testing for risk assessment and the use of risk assessment for testing.

For a theoretical approach starting with instruction sequences discussing RBT is not plausible because for theoretical examples starting out with a risk analysis would be rather artificial. Risk analysis is remote from theoretical considerations on program structure and behaviour. The simpler issue of developing a theoretical framework concerning user profiles seems not to have drawn any attention thus far, while risk analysis would need to take such work as a point of departure. Nevertheless RBT provides in principle a perspective for obtaining quantified positive information from testing, and on the long run developing methods for RBT seems to be a necessity.

Risk assessment may be considered an overhead on testing. However, the same overhead is present, in principle, for formal verification. Indeed RBT stands besides RBFV (risk based formal verification), on equal footing. RBFV would be a theme within RBV (risk based verification). As it turns out *risk based verification* is a widely used phrase in high tech engineering without a connotation of software verification. I did not find work on "risk based software verification" or more specifically on "risk based formal verification". That state of affairs is counterintuitive, however. Indeed given a specification for a system $S \equiv S[X]$ containing a software component X is given as a conjunction of properties $P_1(S), \ldots, P_n(S)$ each of which is amenable to testing as well as to verification. Then one would expect risk analysis to determine for which of these properties the additional effort of obtaining a formal verification on top of a successful and informative test suite would be warranted.

Nevertheless, a focus on instruction sequences is unlikely to shed any further light on RBT.

9.8 Fuzzing

Fuzzing (see e.g. [62]) is another word for systematic and completely automated (and massive) random testing. Fuzzing has become mainstream in testing and has been credited with the detection of tens of thousands of faults in critical software. Fuzzing requires an automated solution to the oracle problem, otherwise it can't be performed at sufficient scale. Typically a new system may be fed with all sorts of inputs (including inputs which lies outside the range which the designers had in mind) to see if a crash takes place. In that manner security problems may be detected which otherwise would be revealed only after the program has been used in practice.

Fuzzing may cover more ground, i.e. look into more cases, than the use of a program in practice will do during its entire lifetime. Fuzzing may uncover problems which otherwise might only be detected after many years. Fuzzing need not be restricted to any known user profile. On the contrary fuzzing may be used to spot vulnerabilities which can be exploited by adversary "users" who don't comply with any known or expected user profile.

9.9 Program Testing in the Context of Machine Learning

Machine learning gives rise to programs the origin of which comes from training data which are used during the learning process. In comparison to classical design, the training data, which embody intended behaviour, determines the logic, rather than the other way around. Lots of blogs (e.g. [45]) have been devoted to the role of testing for programming on the basis of machine learning.

In [93] a distinction is made between machine learning-based testing, which is about the use of machine learning as a tool for testing in the conventional sense, and machine learning testing which uses testing to obtain information about machine learning programs. Further a distinction is made between (i) a learning program (an ML program in the terminology of [93]), software which performs learning and thereby creates a learning system, which may include support for validation and testing, and (ii) the "model" i.e. the learnt program which comes about as a result of machine learning that may be used in practice.

In [32] the separation between model (learnt program), and machine learning program (ML program) is less clear, however. In the terminology of [32] ML programs include learnt programs. I prefer not to follow the latter conventions and instead to view learnt programs as novel outputs of ML programs which are instances of conventional software, i.e. implementations of machine learning algorithms which admit specification in advance.

9.10 On Terminology for Testing Machine Learning Instruction Sequences

I will adopt the following assumptions, which are deliberately cast in terms of inSeq's rather than in terms of programs in order to avoid manifest discrepancies with existing literature (such as [32]). This leads to the following, perhaps over-simplified, picture.

- A model as created from an ML inSeq in combination with training data, validation data, and (perhaps human) input to serve as an oracle for the classifiers to be learnt is referred to as a "learnt inSeq". An ML instruction sequence implements a process for the development of learnt inSeq's. The quality of an ML inSeq is independent from training sets, validation sets, human intervention by way of serving as oracles for the problem which is to be learned.

- An ML inSeq supports the production of learnt inSeq's on the basis of training data and validation data, where the quality of the result may be evaluated by means of so-called test data. Now the ML inSeq is a conventional program to which all and exclusively conventional notions of correctness, validity, verification and testing apply.

  In [93] it is mentioned that training data can contain bugs. That phenomenon would be immaterial for the quality of an ML inSeq (other than the ability of the instruction sequences to deal with flawed training sets), while it matters for created learnt inSeq's.

  ML inSeq testing is about testing ML inSeq's, just as matrix multiplication inSeq testing will be about testing inSeq's for matrix multiplication.

- There is no reason why ML inSeq'ss would not be understood as conventional programs.

- ML testing is about the quality of learnt inSeq's. Judging the quality of learnt inSeq's requires a solution of the oracle problem w.r.t. the intended learnt programs, a topic whiles lies outside the scope of software engineering for an ML program. Thus ML testing is not about the testing of ML inSeq's, but about the testing of ML inSeq outputs, i.e. learnt inSeq's.

9.11 Partisan Ambiguity in Arithmetic: A Case Study from Outside Informatics

The idea of partisan ambiguity can be illustrated with an example from outside computer programming. I hold that the concept/term fraction features partisan ambiguity:

(i) some will claim that a fraction is a composed entity consisting of a numerator, a function symbol for division, and a denominator,

(ii) some will hold that a fraction is a rational number,

(iii) some consider fraction a notion from mathematics, other authors do not,

(iv) some consider fraction to be intentionally ambiguous, by combining two or more of the mentioned options under a single umbrella, etc.

I hold that the partisan ambiguity of fraction stands in the way of an efficient use of that concept/term. As a remedy in [10] it is proposed to use fracterm for an expression with division as its leading function symbol. The value of a fracterm is a fracvalue, and (in elementary arithmetic) fracvalue is merely another word for rational number. A fracterm is a composite entity and for that reason it is not a number. Equivalence (i.e having the same value) of fracterms is denoted with =, and for fracterms t and r, $t = r$ signifies that t and r have the same value (i.e. fracvalue), not that t and r are the same fracterm. Equality of fracterms can plausibly be denoted with $t \equiv r$. Thus the assertion $t = r$ is ambiguous: asserting equivalence of fracterms and asserting the equality of two fracvalues, but both interpretations will lead to the same truth value. Simplification of a fracterm, if possible, yields a different but equivalent fracterm.

I consider fraction not to be intentionally ambiguous. Having made that choice, there is some merit in contemplating the possibility that natural language for arithmetic is, or might be, in need of an intentionally ambiguous umbrella for fracvalue and fracterm. I consider fraction not to be a candidate for that role, because I have no wish to resist the widespread views (i) and (ii) on fractions listed above. I will use *fracthing* for that purpose.

Fracvalue and fracterm are to be distinguished from fracsign, which is a (physical) sign which refers to a fracterm and thereby indirectly to a fracvalue. Fracthing is an umbrella which covers fracsign as well. On a blackboard one may count the number of fracsigns that have been written on it. If one asks for the number of fracthings on a blackboard it is plausible to coerce (i.e. reduce the scope of meaning) fracthing into fracsign first before counting. So there are as many fracthings on the blackboard as fracsigns. For fracsigns the question is clear and also for fracterms and fracvalues: if the blackboard shows the following four fracthings: 1/2, 1/2, 2/4 and 1/3 there are respectively, four fracsigns, three fracterms and two fracvalues. It is an unusual property of fracthings that these include the (physical) fracsigns and yet there are infinitely many fracthings.

The terminology thus obtained is consistent with three plausible but mutually incompatible identifications:

(i) fraction "is" fracterm,

(ii) fraction "is" fracvalue, and

(iii) fraction "is" fracthing.

Rather less plausible is option

(iv) fraction "is" fracsign. Upon adopting any particular convention from these four options a simple account of fractions results, but compatibility with the other three options for assigning a meaning to fraction would then be lost.

9.12 Conclusions

Designing testing4inSeqTh is more easily started than completed. The literature on faults, bugs, testing, fault localization, and debugging is quite extensive. The thought experiment of incorporating terminology, results, and objectives on the area of program testing in testing4inSeqTh has been very helpful for me in the sense that having the liberty to settle for definitions which do not precisely, and in some cases not even remotely, coincide with definitions found elsewhere in the literature on testing, is needed in order to be able to work towards a consistent and coherent picture.

My attempt to take into account a significant part of the issues discussed in the testing literature may not convince the reader. But doing so is a necessity if a realistic picture of testing is to be achieved. I have not spent as much attention to structural testing methods as might be expected given the relative size of that part of the literature on program testing. Testing is a part of technology and therefore the subject is being reinvented time and again, and in the ongoing adaptation to changing practical needs lies its strength. The following relatively recent themes in program testing I consider essential for an appreciation of testing:

(a) fuzzing for finding security flaws,

(b) metamorphic testing as an escape from the oracle problem, a technique which "by definition" finds failures without disclosing any clue on underlying faults, thereby providing an incentive for

(c) spectrum based fault localisation as an instance of white box testing, and

(d) mutation based testing.

It is remarkably difficult to argue convincingly that testing is there to stay, which I think is the case, and that whatever progress is made in the context of formal specification, automatic programming, and formal verification, there will always be a residual need for testing, a methodology which for that reason must be provided with a strong theoretical basis.

Moreover, I expect that with quantum computing becoming incorporated in the practice of informatics, the principled necessity of program testing will become more obvious. The unavoidability of testing will become a consequence of the fact that the unpredictability and computational intractability of the underlying physics cannot be ignored in the same way as it was done for (now) classical and traditional digital computing.

References

[1] Ahmet Ziya Aktaş, Eray Yağderelí, and Doğa Serdaroğlu. An introduction to software testing methodologies. *Gazi University Journal of Science Part A: Engineering and Innovation*, 8:1–15, 2021. URL: https://dergipark.org.tr/en/pub/gujsa/issue/60638/517975.

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

[3] 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.

[4] 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.

[5] Jos C. M. Baeten and Cornelis A. Middelburg. *Process Algebra with Timing*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2002. doi:10.1007/978-3-662-04995-2.

[6] Adam Barr. *The problem with software, why smart engineers write bad code*. MIT Press, 2018.

[7] Boris Beizer. *Software System Testing and Quality Assurance*. Van Nostrand Reinhold, 2nd edition, 1990.

[8] D. Elliott Bell and Leonard J. LaPadula. Secure computer systems: mathematical foundations. Technical Report 2547, MITRE CORP BEDFORD MA, 1973.

[9] 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/.

[10] 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.

[11] Jan A. Bergstra. Promise theory as a tool for informaticians. *Transmathematica*, 2020. doi:10.36285/tm.35.

[12] 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.

[13] Jan A. Bergstra and Inge 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.

[14] Jan A. Bergstra and Mark Burgess. *Promise Theory: Principles and Applications*. χt Axis Press, 2014. 2nd edition, 2019.

[15] Jan A. Bergstra and Mark Burgess. Candidate software process flaws for the Boeing 737 Max MCAS algorithm and risks for a proposed upgrade. *CoRR*, abs/2001.05690, 2020. arXiv:2001.05690.

[16] 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.

[17] Jan A. Bergstra and Cornelis A. Middelburg. Thread algebra for strategic interleaving. *Formal Aspects of Computing*, 19(4):445–474, 2007. doi:10.1007/s00165-007-0024-9.

[18] Jan A. Bergstra and Cornelis 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.

[19] Jan A. Bergstra and Cornelis A. Middelburg. Thread algebra for polythreading. *Formal Aspects of Computing*, 23(4):567–583, 2011. doi:10.1007/s00165-011-0178-3.

[20] 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.

[21] 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.

[22] Jan A. Bergstra and Cornelis A. Middelburg. Instruction sequences expressing multiplication algorithms. *Scientific Annals of Computer Science*, 28(1):39–66, 2018. doi:10.7561/SACS.2018.1.39.

[23] 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/.

[24] Jan A. Bergstra and Alban Ponse. Execution architectures for program algebra. *Journal of Applied Logic*, 5(1):170–192, 2007. doi:10.1016/j.jal.2005.10.013.

[25] Gustavo Betarte, Juan Diego Campo, Carlos Luna, and Agustín Romano. Formal analysis of android's permission-based security model. *Scientific Annals of Computer Science*, 26(1):27–68, 2016. doi:10.7561/SACS.2016.1.27.

[26] Harsh Bhasin, Esha Khanna, and Sudha Sudha. Black box testing based on requirement analysis and design specifications. *International Journal of Computer Applications*, 87(18):36–40, 2014. doi:10.5120/15311-4024.

[27] Ken J. Biba. Integrity considerations for secure computer systems. Technical Report 3154, MITRE CORP BEDFORD MA, 1977.

[28] Manuel Blum and Sampath Kannan. Designing programs that check their work. *Journal of the ACM*, 42(1):269–291, 1995. doi:10.1145/200836.200880.

[29] Marcel Böhme, Cristian Cadar, and Abhik Roychoudhury. Fuzzing: Challenges and reflections. *IEEE Software*, 38(3):79–86, 2021. doi:10.1109/MS.2020.3016773.

[30] Adilson Luiz Bonifácio and Arnaldo Vieira Moura. Test suite completeness and black box testing. *Software Testing, Verification and Reliability*, 27(1-2), 2017. doi:10.1002/stvr.1626.

[31] Luc Bougé. A contribution to the theory of program testing. *Theoretical Computer Science*, 37:151–181, 1985. doi:10.1016/0304-3975(85)90090-8.

[32] Houssem Ben Braiek and Foutse Khomh. On testing machine learning programs. *Journal of Systems and Software*, 164:110542, 2020. doi:10.1016/j.jss.2020.110542.

[33] Robert Cartwright. Formal program testing. In John White, Richard J. Lipton, and Patricia C. Goldberg, editors, *Conference Record of the Eighth Annual ACM Symposium on Principles of Programming Languages, POPL 1981*, pages 125–132. ACM Press, 1981. doi:10.1145/567532.567546.

[34] Christophe Chareton, Sébastien Bardin, Dongho Lee, Benoît Valiron, Renaud Vilmart, and Zhaowei Xu. Formal methods for quantum programs: A survey. *CoRR*, abs/2109.06493, 2021. arXiv:2109.06493.

[35] Tse-Hsun Chen, Meiyappan Nagappan, Emad Shihab, and Ahmed E. Hassan. An empirical study of dormant bugs. In Premkumar T. Devanbu, Sung Kim, and Martin Pinzger, editors, *11th Working Conference on Mining Software Repositories, MSR 2014*, pages 82–91. ACM, 2014. doi:10.1145/2597073.2597108.

[36] Tsong Yueh Chen, S. C. Cheung, and Siu-Ming Yiu. Metamorphic testing: A new approach for generating next test cases. Technical Report HKUST-CS98-01, Deptartment of Computer Science, The Hong Kong University of Science and Technology, 1988. arXiv:2002.12543.

[37] Mohammad Torabi Dashti and David A. Basin. A theory of black-box tests. *CoRR*, abs/2006.10387, 2020. arXiv:2006.10387.

[38] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978. doi:10.1109/C-M.1978.218136.

[39] Gencer Erdogan, Yan Li, Ragnhild Kobro Runde, Fredrik Seehusen, and Ketil Stølen. Approaches for the combined use of risk analysis and testing: a systematic literature review. *International Journal on Software Tools for Technology Transfer*, 16(5):627–642, 2014. doi:10.1007/s10009-014-0330-5.

[40] Michael Felderer, Matthias Büchler, Martin Johns, Achim D. Brucker, Ruth Breu, and Alexander Pretschner. Security testing: A survey. *Advances in Computers*, 101:1–51, 2016. doi:10.1016/bs.adcom.2015.11.003.

[41] Michael Felderer and Vahid Garousi. Together we are stronger: Evidence-based reflections on industry-academia collaboration in software testing. In Dietmar Winkler, Stefan Biffl, Daniel Méndez, and Johannes Bergsmann, editors, *12th International Conference on Software Quality: Quality Intelligence in Software and Systems Engineering, SWQD 2020*, volume 371 of *Lecture Notes in Business Information Processing*, pages 3–12. Springer, 2020. doi:10.1007/978-3-030-35510-4_1.

[42] Gordon Fraser and José Miguel Rojas. Software testing. In Sungdeok Cha, Richard N. Taylor, and Kyo Chul Kang, editors, *Handbook of Software Engineering*, pages 123–192. Springer, 2019. doi:10.1007/978-3-030-00262-6_4.

[43] Vahid Garousi, Michael Felderer, and Feyza Nur Kilicaslan. A survey on software testability. *Information and Software Technology*, 108:35–64, 2019. doi:10.1016/j.infsof.2018.12.003.

[44] Marie-Claude Gaudel. Testing can be formal, too. In Peter D. Mosses, Mogens Nielsen, and Michael I. Schwartzbach, editors, *TAPSOFT'95: Theory and Practice of Software Development, 6th International Joint Conference CAAP/FASE*, volume 915 of *Lecture Notes in Computer Science*, pages 82–96. Springer, 1995. doi:10.1007/3-540-59293-8_188.

[45] Yulia Gavrilova. Machine learning testing: a step to perfection. Serokell Site, November 11 2020 [Online]. URL: https://serokell.io/blog/machine-learning-testing.

[46] David Gelperin and Bill Hetzel. The growth of software testing. *Communications of the ACM*, 31(6):687–695, 1988. doi:10.1145/62959.62965.

[47] Joseph A. Goguen and José Meseguer. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy*, pages 11–20. IEEE Computer Society, 1982. doi:10.1109/SP.1982.10014.

[48] John B. Goodenough and Susan L. Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, 1(2):156–173, 1975. doi:10.1109/TSE.1975.6312836.

[49] John S. Gourlay. A mathematical framework for the investigation of testing. *IEEE Transactions on Software Engineering*, 9(6):686–709, 1983. doi:10.1109/TSE.1983.235433.

[50] Jens Grabowski, Anthony Wiles, Colin Willcock, and Dieter Hogrefe. On the design of the new testing language TTCN-3. In Hasan Ural, Robert L. Probert, and Gregor von Bochmann, editors, *Testing of Communicating Systems: Tools and Techniques, IFIP TC6/WG6.1 13^{th} International Conference on Testing Communicating Systems (TestCom 2000)*, volume 176 of *IFIP Conference Proceedings*, pages 161–176. Kluwer, 2000.

[51] Philip J. Guo, Thomas Zimmermann, Nachiappan Nagappan, and Brendan Murphy. "Not my bug!" and other reasons for software bug

report reassignments. In Pamela J. Hinds, John C. Tang, Jian Wang, Jakob E. Bardram, and Nicolas Ducheneaut, editors, *Proceedings of the 2011 ACM Conference on Computer Supported Cooperative Work, CSCW 2011*, pages 395–404. ACM, 2011. doi:10.1145/1958824.1958887.

[52] Brent Hailpern and Padmanabhan Santhanam. Software debugging, testing, and verification. *IBM Systems Journal*, 41(1):4–12, 2002. doi:10.1147/sj.411.0004.

[53] Dick Hamlet. Predicting dependability by testing. *ACM SIGSOFT Software Engineering Notes*, 21(3):84–91, 1996. doi:10.1145/226295.226305.

[54] Richard G. Hamlet. Foundations of software testing: Dependability theory. In David S. Wile, editor, *Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering, SIGSOFT 1994*, pages 128–139. ACM, 1994. doi:10.1145/193173.195400.

[55] Shahin Honarvar, Mohammad Reza Mousavi, and Rajagopal Nagarajan. Property-based testing of quantum programs in q#. In *ICSE '20: 42nd International Conference on Software Engineering*, pages 430–435. ACM, 2020. doi:10.1145/3387940.3391459.

[56] Cem Kaner. Software negligence and testing coverage. In *STAR 96: Fifth International Conference on Software Testing, Analysis, and Review*, pages 313–327, 1996.

[57] Robert Könighofer, Georg Hofferek, and Roderick Bloem. Debugging formal specifications using simple counterstrategies. In *Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2009*, pages 152–159. IEEE, 2009. doi:10.1109/FMCAD.2009.5351127.

[58] Christine M. Kreiser. Computer bug. *American History*, 45(6):19, 2011.

[59] Butler W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973. doi:10.1145/362375.362389.

[60] Carl E. Landwehr, Alan R. Bull, John P. McDermott, and William S. Choi. A taxonomy of computer program security flaws. *ACM Computing Surveys*, 26(3):211–254, 1994. doi:10.1145/185403.185412.

[61] 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.

[62] Jun Li, Bodong Zhao, and Chao Zhang. Fuzzing: a survey. *Cybersecurity*, 1(1):6, 2018. doi:10.1186/s42400-018-0002-y.

[63] Richard J. Lipton. New directions in testing. In Joan Feigenbaum and Michael Merritt, editors, *Distributed Computing And Cryptography, Proceedings of a DIMACS Workshop*, volume 2 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 191–202. DIMACS/AMS, 1989. doi:10.1090/dimacs/002/13.

[64] Pan Liu, Yihao Li, and Huaikou Miao. Transition algebra for software testing. *IEEE Transactions on Reliability*, 70(4):1438–1454, 2021. doi:10.1109/TR.2021.3116054.

[65] Lu Luo. Software testing techniques: Technology maturation and research strategies. Technical Report 17-939A, Institute for Software Research International, Carnegie Mellon University, 2001.

[66] Annabelle McIver and Carroll Morgan. Correctness by construction for probabilistic programs. In Tiziana Margaria and Bernhard Steffen, editors, *9th International Symposium on Leveraging Applications of Formal Methods: Verification Principles, ISoLA 2020*, volume 12476 of *Lecture Notes in Computer Science*, pages 216–239. Springer, 2020. doi:10.1007/978-3-030-61362-4_12.

[67] Cornelis A. Middelburg. Searching publications on software testing. *CoRR*, abs/1008.2647, 2010. arXiv:1008.2647.

[68] Cornelis A. Middelburg. Program algebra for random access machine programs. *CoRR*, abs/2007.09946, 2020. arXiv:2007.09946.

[69] 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.

[70] Roy Miller and Christopher Collins. Acceptance testing. In *Proceedings XPUniverse 2001*, page 238, 2001.

[71] Andriy V. Miranskyy, Lei Zhang, and Javad Doliskani. Is your quantum program bug-free? In Gregg Rothermel and Doo-Hwan Bae, editors, *ICSE-NIER 2020: 42nd International Conference on Software Engineering, New Ideas and Emerging Results*, pages 29–32. ACM, 2020. doi:10.1145/3377816.3381731.

[72] Joshua Moerman. *Nominal Techniques and Black Box Testing for Automata Learning*. PhD thesis, Radboud University Nijmegen, 2019. URL: http://hdl.handle.net/2066/204194.

[73] Edward F. Moore. *Gedanken-Experiments on Sequential Machines*, pages 129–154. Princeton University Press, 1956. doi:10.1515/9781400882618-006.

[74] Srinivas Nidhra and Jagruthi Dondeti. Black box and white box testing techniques-a literature review. *International Journal of Embedded Systems and Applications (IJESA)*, 2(2):29–50, 2012. doi:10.5121/ijesa.2012.2204.

[75] Adrian Nistor. *Understanding, detecting, and repairing performance bugs*. PhD thesis, University of Illinois at Urbana-Champaign, 2014. URL: https://mir.cs.illinois.edu/marinov/publications/Nistor14PhD.pdf.

[76] Tom Ridge, David Sheets, Thomas Tuerk, Andrea Giugliano, Anil Madhavapeddy, and Peter Sewell. Sibylfs: formal specification and oracle-based testing for POSIX and real-world file systems. In Ethan L. Miller and Steven Hand, editors, *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015*, pages 38–53. ACM, 2015. doi:10.1145/2815400.2815411.

[77] Sergio Segura, Dave Towey, Zhi Quan Zhou, and Tsong Yueh Chen. Metamorphic testing: Testing the untestable. *IEEE Software*, 37(3):46–53, 2020. doi:10.1109/MS.2018.2875968.

[78] Fred R. Shapiro. Etymology of the computer bug: History and folklore. *American Speech*, 62(4):376–378, 1987. URL: `http://www.jstor.org/stable/455415`.

[79] Emad Shihab, Audris Mockus, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. High-impact defects: a study of breakage and surprise defects. In Tibor Gyimóthy and Andreas Zeller, editors, *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13)*, pages 300–310. ACM, 2011. `doi:10.1145/2025113.2025155`.

[80] Yogesh Singh. *Software Testing*. Cambridge University Press, 2011. `doi:10.1017/CBO9781139196185`.

[81] Eric Steegmans, Pieter Bekaert, Frank Devos, Geert Delanote, Nele Smeets, Marko van Dooren, and Jeroen Boydens. Black & white testing: Bridging black box testing and white box testing. pages 1–12. Sterck, P, 2004. URL: `https://lirias.kuleuven.be/retrieve/6947`.

[82] Jan Tretmans. Testing concurrent systems: A formal approach. In Jos C. M. Baeten and Sjouke Mauw, editors, *CONCUR '99: 10th International Conference on Concurrency Theory*, volume 1664 of *Lecture Notes in Computer Science*, pages 46–65. Springer, 1999. `doi:10.1007/3-540-48320-9\_6`.

[83] Jeffrey M. Voas and Keith W. Miller. Software testability: The new verification. *IEEE Software*, 12(3):17–28, 1995. `doi:10.1109/52.382180`.

[84] Thuy Duong Vu. Thread algebra for noninterference. *RAIRO - Theoretical Informatics and Applications*, 43(2):249–268, 2009. `doi:10.1051/ita:2008026`.

[85] John Watkins. *Agile Testing: How to Succeed in an Extreme Testing Environment*. Cambridge University Press, 2009.

[86] Yi Wei, Bertrand Meyer, and Manuel Oriol. Is branch coverage a good measure of testing effectiveness? In Bertrand Meyer and Martin Nordio, editors, *International Summer Schools on Empirical Software Engineering and Verification, LASER 2008-2010, Revised Tutorial Lec-*

tures, volume 7007 of *Lecture Notes in Computer Science*, pages 194–212. Springer, 2010. doi:10.1007/978-3-642-25231-0_5.

[87] Elaine J. Weyuker. On testing non-testable programs. *The Computer Journal*, 25(4):465–470, 1982. doi:10.1093/comjnl/25.4.465.

[88] Elaine J. Weyuker. Evaluating software complexity measures. *IEEE Transactions on Software Engineering*, 14(9):1357–1365, 1988. doi:10.1109/32.6178.

[89] Elaine J. Weyuker and Thomas J. Ostrand. Theories of program testing and the application of revealing subdomains. *IEEE Transactions on Software Engineering*, 6(3):236–246, 1980. doi:10.1109/TSE.1980.234485.

[90] Linmin Yang, Zhe Dang, and Thomas R. Fischer. Information gain of black-box testing. *Formal Aspects of Computing*, 23(4):513–539, 2011. doi:10.1007/s00165-011-0175-6.

[91] Peng Yang, Zixi Liu, Jin Xu, Yong Huang, and Ya Pan. An empirical study on the ability relationships between programming and testing. *IEEE Access*, 8:161438–161448, 2020. doi:10.1109/ACCESS.2020.3018718.

[92] Abubakar Zakari, Sai Peck Lee, and Chun Yong Chong. Simultaneous localization of software faults based on complex network theory. *IEEE Access*, 6:23990–24002, 2018. doi:10.1109/ACCESS.2018.2829541.

[93] Jie M. Zhang, Mark Harman, Lei Ma, and Yang Liu. Machine learning testing: Survey, landscapes and horizons. *IEEE Transactions on Software Engineering*, 48(2):1–36, 2022. doi:10.1109/TSE.2019.2962027.