The Oracle Problem in Software Testing: A Survey

Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo

Abstract—Testing involves examining the behaviour of a system in order to discover potential faults. Given an input for a system, the challenge of distinguishing the corresponding desired, correct behaviour from potentially incorrect behavior is called the "test oracle problem". Test oracle automation is important to remove a current bottleneck that inhibits greater overall test automation. Without test oracle automation, the human has to determine whether observed behaviour is correct. The literature on test oracles has introduced techniques for oracle automation, including modelling, specifications, contract-driven development and metamorphic testing. When none of these is completely adequate, the final source of test oracle information remains the human, who may be aware of informal specifications, expectations, norms and domain specific information that provide informal oracle guidance. All forms of test oracles, even the humble human, involve challenges of reducing cost and increasing benefit. This paper provides a comprehensive survey of current approaches to the test oracle problem and an analysis of trends in this important area of software testing research and practice.

Index Terms—Test oracle, automatic testing, testing formalism

◆

1 INTRODUCTION

MUCH work on software testing seeks to automate as much of the test process as practical and desirable, to make testing faster, cheaper, and more reliable. To this end, we need a test oracle, a procedure that distinguishes between the correct and incorrect behaviors of the System Under Test (SUT).

However, compared to many aspects of test automation, the problem of automating the test oracle has received significantly less attention, and remains comparatively less well-solved. This current open problem represents a significant bottleneck that inhibits greater test automation and uptake of automated testing methods and tools more widely. For instance, the problem of automatically generating test inputs has been the subject of research interest for nearly four decades [46], [107]. It involves finding inputs that cause execution to reveal faults, if they are present, and to give confidence in their absence, if none are found. Automated test input generation been the subject of many significant advances in both Search-Based Testing [3], [5], [83], [126], [128] and Dynamic Symbolic Execution [75], [108], [161]; yet none of these advances address the issue of checking generated inputs with respect to expected behaviours—that is, providing an automated solution to the test oracle problem.

Of course, one might hope that the SUT has been developed under excellent design-for-test principles, so that there might be a detailed, and possibly formal, specification of intended behaviour. One might also hope that the code itself contains pre- and post- conditions that implement well-understood contract-driven development approaches [135]. In these situations, the test oracle cost problem is ameliorated by the presence of an automatable test oracle to which a testing tool can refer to check outputs, free from the need for costly human intervention.

Where no full specification of the properties of the SUT exists, one may hope to construct a partial test oracle that can answer questions for some inputs. Such partial test oracles can be constructed using metamorphic testing (built from known relationships between desired behaviour) or by deriving oracular information from execution or documentation.

For many systems and most testing as currently practiced in industry, however, the tester does not have the luxury of formal specifications or assertions, or automated partial test oracles [91], [92]. The tester therefore faces the daunting task of manually checking the system's behaviour for all test cases. In such cases, automated software testing approaches must address the human oracle cost problem [1], [82], [130].

To achieve greater test automation and wider uptake of automated testing, we therefore need a concerted effort to find ways to address the test oracle problem and to integrate automated and partially automated test oracle solutions into testing techniques. This paper seeks to help address this challenge by providing a comprehensive review and analysis of the existing literature of the test oracle problem.

Four partial surveys of topics relating to test oracles precede this one. However, none has provided a comprehensive survey of trends and results. In 2001, Baresi and Young [17] presented a partial survey that covered four topics prevalent at the time the paper was published: assertions, specifications, state-based conformance testing, and log file analysis. While these topics remain important, they capture only a part of the overall landscape of research in test

• E.T. Barr, M. Harman and S. Yoo are with the Department of Computer Science, University College London, Gower Street, London WC2R 2LS, London, United Kingdom. E-mail: {e.barr, m.harman, shin.yoo}@ucl.ac.uk.
• P. McMinn and M. Shahbaz are with the Department of Computer Science, University of Sheffield, Sheffield S1 4DP, South Yorkshire, United Kingdom.
E-mail: p.mcminn@sheffield.ac.uk, muzammil.shahbaz@gmail.com.

oracles, which the present paper covers. Another early work was the initial motivation for considering the test oracle problem contained in Binder's textbook on software testing [23], published in 2000. More recently, in 2009, Shahamiri et al. [164] compared six techniques from the specific category of derived test oracles. In 2011, Staats et al. [174] proposed a theoretical analysis that included test oracles in a revisitation of the fundamentals of testing. Most recently, in 2014, Pezzè and Zhang focus on automated test oracles for functional properties [150].

Despite this work, research into the test oracle problem remains an activity undertaken in a fragmented community of researchers and practitioners. The role of the present paper is to overcome this fragmentation in this important area of software testing by providing the first comprehensive analysis and review of work on the test oracle problem.

The rest of the paper is organised as follows: Section 2 sets out the definitions relating to test oracles that we use to compare and contrast the techniques in the literature. Section 3 relates a historical analysis of developments in the area. Here we identify key milestones and track the volume of past publications. Based on this data, we plot growth trends for four broad categories of solution to the test oracle problem, which we survey in Sections 4, 5, 6, and 7. These four categories comprise approaches to the oracle problem where:

- test oracles can be *specified* (Section 4);
- test oracles can be *derived* (Section 5);
- test oracles can be built from *implicit* information (Section 6); and
- *no automatable oracle* is available, yet it is still possible to reduce human effort (Section 7).

Finally, Section 8 concludes with closing remarks.

2 DEFINITIONS

This section presents definitions to establish a lingua franca in which to examine the literature on oracles. These definitions are formalised to avoid ambiguity, but the reader should find that it is also possible to read the paper using only the informal descriptions that accompany these formal definitions. We use the theory to clarify the relationship between algebraic specification, pseudo oracles, and metamorphic relations in Section 5.

To begin, we define a test activity as a stimulus or response, then test activity sequences that incorporate constraints over stimuli and responses. Test oracles accept or reject test activity sequences, first deterministically then probabilistically. We then define notions of soundness and completeness of test oracles.

2.1 Test Activities

To test is to stimulate a system and observe its response. A stimulus and a response both have values, which may coincide, as when the stimulus value and the response are both reals. A system has a set of components C. A stimulus and its response target a subset of components. For instance, a common pattern for constructing test oracles is to compare the output of distinct components on the same stimulus value. Thus, stimuli and responses are values that target components. Collectively, stimuli and responses are test activities:
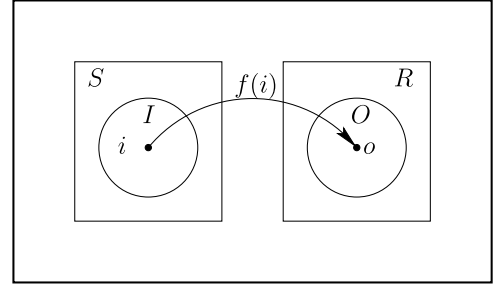

Fig. 1. Stimulus and observations: S is anything that can change the observable behavior of the SUT f; R is anything that can be observed about the system's behavior; I includes f's explicit inputs; O is its explicit outputs; everything not in $S \cup R$ neither affects nor is affected by f.

Definition 2.1 (Test Activities). *For the SUT p, S is the set of stimuli that trigger or constrain p's computation and R is the set of observable responses to a stimulus of p. S and R are disjoint.* Test activities *form the set $A = S \uplus R$.*

The use of disjoint union implicitly labels the elements of A, which we can flatten to the tuple $L \times C \times V$, where $L = \{\text{stimulus}, \text{response}\}$ is the set of activities labels, C is the set of components, and V is an arbitrary set of values. To model those aspects of the world that are independent of any component, like a clock, we set an activity's target to the empty set.

We use the terms "stimulus" and "observation" in the broadest sense possible to cater to various testing scenarios, functional and nonfunctional. As shown in Fig. 1, a stimulus can be either an explicit test input from the tester, $I \subset S$, or an environmental factor that can affect the testing, $S \setminus I$. Similarly, an observation ranges from an output of the SUT, $O \subset R$, to a nonfunctional execution profile, like execution time in $R \setminus O$.

For example, stimuli include the configuration and platform settings, database table contents, device states, resource constraints, preconditions, typed values at an input device, inputs on a channel from another system, sensor inputs and so on. Notably, resetting a SUT to an initial state is a stimulus and stimulating the SUT with an input runs it. Observations include anything that can be discerned and ascribed a meaning significant to the purpose of testing—including values that appear on an output device, database state, temporal properties of the execution, heat dissipated during execution, power consumed, or any other measurable attributes of its execution. Stimuli and observations are members of different sets of test activities, but we combine them into test activities.

2.2 Test Activity Sequence

Testing is a sequence of stimuli and response observations. The relationship between stimuli and responses can often be captured formally; consider a simple SUT that squares its input. To compactly represent infinite relations between stimulus and response values such as $(i, o = i^2)$, we introduce a compact notation for set comprehensions:

$$x{:}[\phi] = \{x \mid \phi\},$$

where x is a dummy variable over an arbitrary set.

Definition 2.2 (Test Activity Sequence). *A test activity sequence is an element of $T_A = \{w \mid T \xrightarrow{*} w\}$ over the grammar*
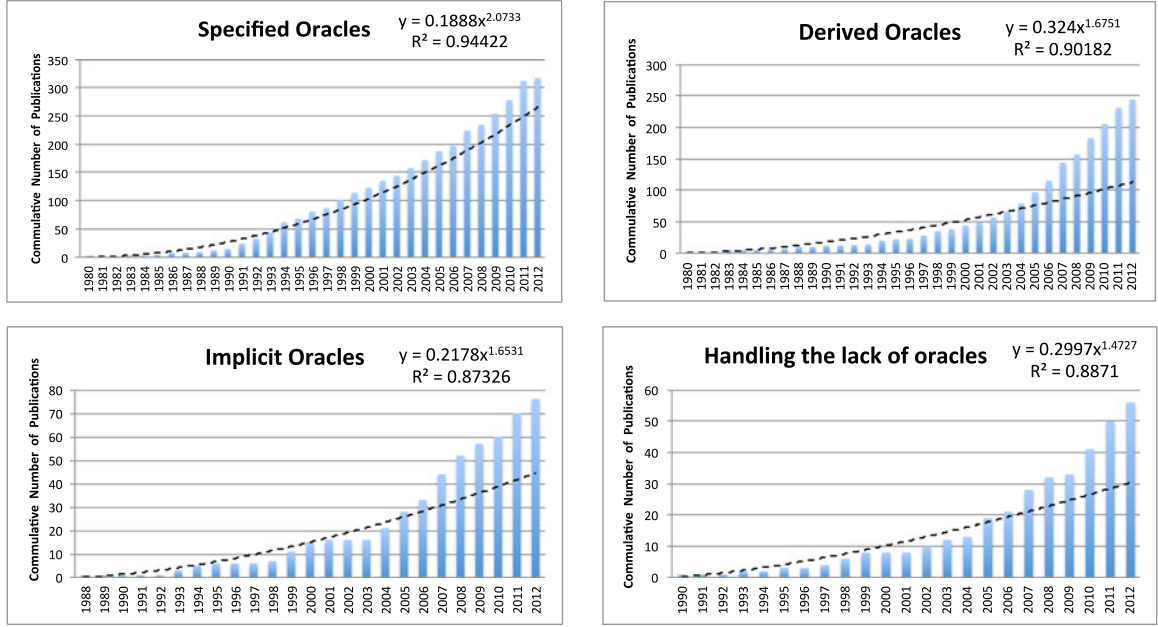
Fig. 2. Cumulative number of publications from 1978 to 2012 and research trend analysis for each type of test oracle. The x-axis represents years and y-axis the cumulative number of publications. We use a power regression model to perform the trend analysis. The regression equation and the coefficient of determination (R^2) indicate a upward future trend, a sign of a healthy research community.

$$T ::= A' : [' \phi ']' T \mid AT \mid \epsilon$$

where A is the test activity alphabet.

Under Definition 2.2, the testing activity sequence $io : [o = i^2]$ denotes the stimulus of invoking f on i, then observing the response output. It further specifies valid responses obeying $o = i^2$. Thus, it compactly represents the infinite set of test activity sequences $i_1 o_1, i_2 o_2, \ldots$ where $o_k = i_k^2$.

For practical purposes, a test activity sequence will almost always have to satisfy constraints in order to be useful. Under our formalism, these constraints differentiate the approaches to test oracle we survey. As an initial illustration, we constrain a test activity sequence to obtain a practical test sequence:

Definition 2.3 (Practical Test Sequence). *A practical test sequence is any test activity sequence w that satisfies*

$$w = TsTrT, \quad \text{for } s \in S, r \in R.$$

Thus, the test activity sequence, w, is *practical* iff w contains at least one stimulus followed by at least one observation.

This notion of a test sequence is nothing more than a very general notion of what it means to test; we must do something to the system (the stimulus) and subsequently observe some behaviour of the system (the observation) so that we have something to check (the observation) and something upon which this observed behaviour depends (the stimulus).

A reliable reset $(\mathrm{p}, \mathrm{r}) \in S$ is a special stimulus that returns the SUT's component p to its start state. The test activity sequence $(\text{stimulus}, \mathrm{p}, \mathrm{r})(\text{stimulus}, \mathrm{p}, i)$ is therefore equivalent to the conventional application notation $\mathrm{p}(i)$. To extract the value of an activity, we write $\mathrm{v}(a)$; to extract its target component, we write $c(a)$. To specify two invocations of a single

component on the different values, we must write $\mathrm{r}_1 i_1 \mathrm{r}_2, i_2 :$ $[\mathrm{r}_1, i_1, \mathrm{r}_2, i_2 \in S, c(\mathrm{r}_1) = c(i_1) = c(\mathrm{r}_2) = c(i_2) \wedge v(i_1) \neq v(i_2)]$. In the sequel, we often compare different executions of a single SUT or compare the output of independently implemented components of the SUT on the same input value. For clarity, we introduce syntactic sugar to express constraints on stimulus values and components. We let $f(x)$ denote $\mathrm{r}i :$ $[c(i) = f \wedge \mathrm{v}(i) = x]$, for $f \in C$.

A *test oracle* is a predicate that determines whether a given test activity sequence is an acceptable behaviour of the SUT or not. We first define a "test oracle", and then relax this definition to "probabilistic test oracle".

Definition 2.4 (Test Oracle). *A test oracle $D : T_A \mapsto \mathbb{B}$ is a partial[1] function from a test activity sequence to true or false.*

When a test oracle is defined for a test activity, it either accepts the test activity or not. Concatenation in a test activity sequence denotes sequential activities; the test oracle D permits parallel activities when it accepts different permutations of the same stimuli and response observations. We use D to distinguish a deterministic test oracle from probabilistic ones. Test oracles are typically computationally expensive, so probabilistic approaches to the provision of oracle information may be desirable even where a deterministic test oracle is possible [124].

Definition 2.5 (Probabilistic Test Oracle). *A probabilistic test oracle $\tilde{D} : T_A \mapsto [0, 1]$ maps a test activity sequence into the interval $[0, 1] \in \mathbb{R}$.*

A probabilistic test oracle returns a real number in the closed interval $[0, 1]$. As with test oracles, we do not require a probabilistic test oracle to be a total function. A

---

1. Recall that a function is implicitly total: it maps every element of its domain to a single element of its range. The *partial* function $f : X \mapsto Y$ is the total function $f' : X' \rightarrow Y$, where $X' \subseteq X$.

probabilistic test oracle can model the case where the test oracle is only able to efficiently offer a probability that the test case is acceptable, or for other situations where some degree of imprecision can be tolerated in the test oracle's response.

Our formalism combines a language-theoretic view of stimulus and response activities with constraints over those activities; these constraints explicitly capture specifications. The high-level language view imposes a temporal order on the activities. Thus, our formalism is inherently temporal. The formalism of Staats et al. captures any temporal exercising of the SUT's behavior in tests, which are atomic black boxes for them [174]. Indeed, practitioners write test plans and activities, they do not often write specifications at all, let alone a formal one. This fact and the expressivity of our formalism, as evident in our capture of existing test oracle approaches, is evidence that our formalism is a good fit with practice.

2.3   Soundness and Completeness
We conclude this section by defining soundness and completeness of test oracles.

In order to define soundness and completeness of a test oracle, we need to define a concept of the "ground truth", $\mathcal{G}$. The ground truth is another form of oracle, a conceptual oracle, that always gives the "right answer". Of course, it cannot be known in all but the most trivial cases, but it is a useful definition that bounds test oracle behaviour.

Definition 2.6 (Ground Truth). *The* ground truth oracle, $\mathcal{G}$, *is a total test oracle that always gives the "right answer".*

We can now define soundness and completeness of a test oracle with respect to $\mathcal{G}$.

Definition 2.7 (Soundness). *The test oracle D is* sound *iff*
$$D(a) \Rightarrow \mathcal{G}(a).$$

Definition 2.8 (Completeness). *The test oracle D is* complete *iff*
$$\mathcal{G}(a) \Rightarrow D(a).$$

While test oracles cannot, in general, be both sound and complete, we can, nevertheless, define and use partially correct test oracles. Further, one could argue, from a purely philosophical point of view, that human oracles can be sound and complete, or correct. In this view, correctness becomes a subjective human assessment. The foregoing definitions allow for this case.

We relax our definition of soundness to cater for probabilistic test oracles:

Definition 2.9 (Probablistic Soundness and Completeness). *A probabilistic test oracle $\tilde{D}$ is* probabilistically sound *iff*
$$P(\tilde{D}(w) = 1) > \frac{1}{2} + \epsilon \Rightarrow \mathcal{G}(w)$$
and $\tilde{D}$ is probabilistically complete *iff*
$$\mathcal{G}(w) \Rightarrow P(\tilde{D}(w) = 1) > \frac{1}{2} + \epsilon$$
where ϵ is non-negligible.

The non-negligible advantage ϵ requires $\tilde{D}$ to do sufficiently better than flipping a fair coin, which for a binary classifier maximizes entropy, that we can achieve arbitrary confidence in whether the test sequence w is valid by repeatedly sampling $\tilde{D}$ on w.

3   TEST ORACLE RESEARCH TRENDS
The term "test oracle" first appeared in William Howden's seminal work in 1978 [99]. In this section, we analyze the research on test oracles, and its related areas, conducted since 1978. We begin with a synopsis of the volume of publications, classified into specified, derived, implicit, and lack of automated test oracles. We then discuss when key concepts in test oracles were first introduced.

3.1   Volume of Publications
We constructed a repository of 694 publications on test oracles and its related areas from 1978 to 2012 by conducting web searches for research articles on *Google Scholar* and *Microsoft Academic Search* using the queries "software + test + oracle" and "software + test oracle"[2], for each year. Although some of the queries generated in this fashion may be similar, different responses are obtained, with particular differences around more lowly-ranked results.

We classify work on test oracles into four categories: specified test oracles (317), derived test oracles (245), implicit test oracles (76), and no test oracle (56), which handles the lack of a test oracle.

Specified test oracles, discussed in detail in Section 4, judge all behavioural aspects of a system with respect to a given formal specification. For specified test oracles we searched for related articles using queries "formal + specification", "state-based specification", "model-based languages", "transition-based languages", "assertion-based languages", "algebraic specification" and "formal + conformance testing". For all queries, we appended the keywords with "test oracle" to filter the results for test oracles.

Derived test oracles (see Section 5) involve artefacts from which a test oracle may be derived—for instance, a previous version of the system. For derived test oracles, we searched for additional articles using the queries "specification inference", "specification mining", "API mining", "metamorphic testing", "regression testing" and "program documentation".

An implicit oracle (see Section 6) refers to the detection of "obvious" faults such as a program crash. For implicit test oracles we applied the queries "implicit oracle", "null pointer + detection", "null reference + detection", "deadlock + livelock + race + detection", "memory leaks + detection", "crash + detection", "performance + load testing", "non-functional + error detection", "fuzzing + test oracle" and "anomaly detection".

There have also been papers researching strategies for handling the lack of an automated test oracle (see Section 7). Here, we applied the queries "human oracle", "test

2. We use + to separate the keywords in a query; a phrase, not internally separated by +, like "test oracle", is a compound keyword, quoted when given to the search engine.
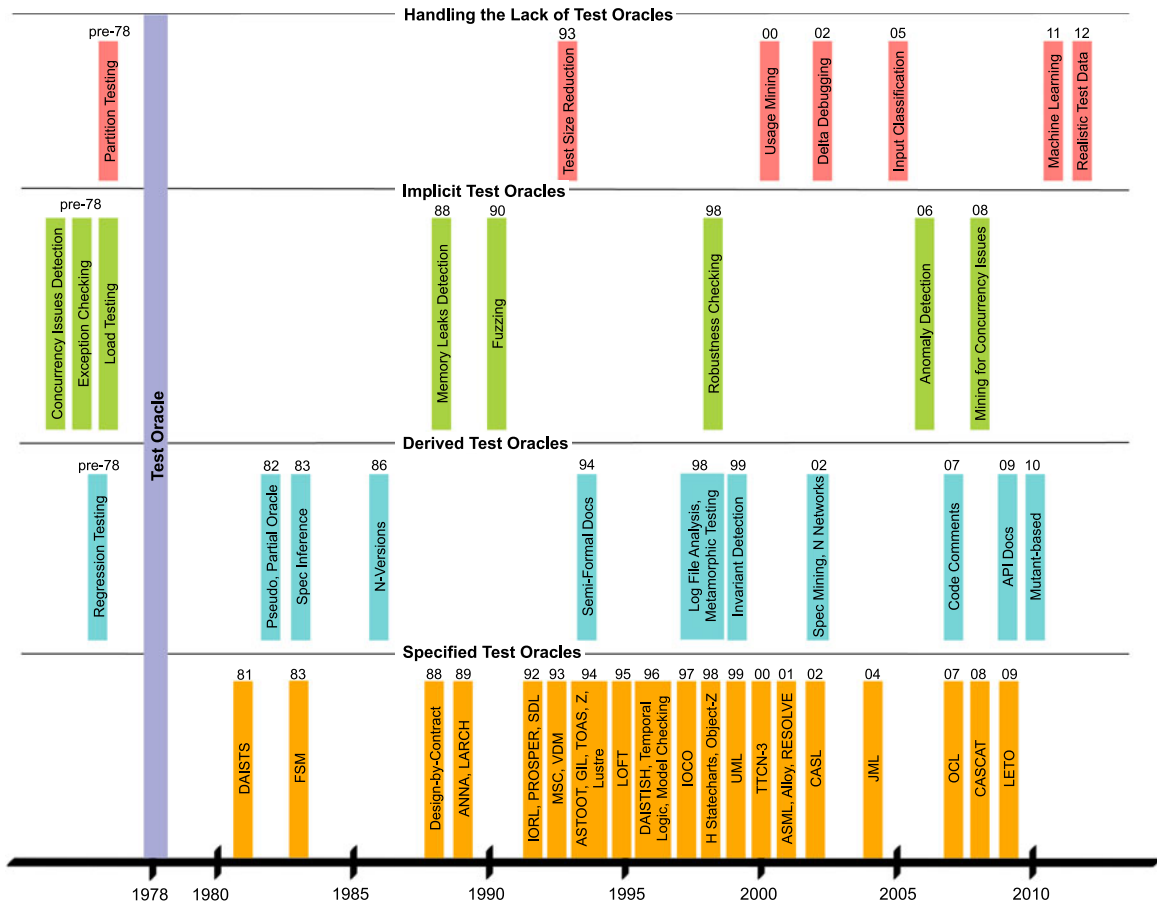
Fig. 3. Chronological introduction of test oracles techniques and concepts.

minimization", "test suite reduction" and "test data + generation + realistic + valid".

Each of the above queries were appended by the keywords "software testing". The results were filtered, removing articles that were found to have no relation to software testing and test oracles. Fig. 2 shows the cumulative number of publications on each type of test oracle from 1978 onwards. We analyzed the research trend on this data by applying different regression models. The trend line, shown in Fig. 2, is fitted using a power model. The high values for the four coefficients of determination (R^2), one for each of the four types of test oracle, confirm that our models are good fits to the trend data. The trends observed suggest a healthy growth in research volumes in these topics related to the test oracle problem in the future.

3.2 The Advent of Test Oracle Techniques

We classified the collected publications by techniques or concepts they proposed to (partially) solve a test oracle problem; for example, *Model Checking* [35] and *Metamorphic Testing* [36] fall into the derived test oracle and *DAISTIS* [69] is an algebraic specification system that addresses the specified test oracle problem.

For each type of test oracle and the advent of a technique or a concept, we plotted a timeline in chronological order of publications to study research trends. Fig. 3 shows the timeline starting from 1978 when the term "test oracle" was first coined. Each vertical bar presents the technique or concept used to solve the problem labeled with the year of its first publication.

The timeline shows only the work that is explicit on the issue of test oracles. For example, the work on test generation using finite state machines (FSM) can be traced back to as early as 1950s. But the explicit use of finite state machines to generate test oracles can be traced back to Jard and Bochmann [102] and Howden in 1986 [98]. We record, in the timeline, the earliest available publication for a given technique or concept. We consider only published work in journals, the proceedings of conferences and workshops, or magazines. We excluded all other types of documentation, such as technical reports and manuals.

Fig. 3 shows a few techniques and concepts that predate 1978. Although not explicitly on test oracles, they identify and address issues for which test oracles were later developed. For example, work on detecting concurrency issues (deadlock, livelock, and races) can be traced back to the 1960s. Since these issues require no specification, implicit test oracles can and have been built that detect them on arbitrary systems. Similarly, *Regression Testing* detects problems in the functionality a new version of a system shares with its predecessors and is a precursor of derived test oracles.

The trend analysis suggests that proposals for new techniques and concepts for the formal specification of test oracles peaked in 1990s, and has gradually diminished in the last decade. However, it remains an area of much research activity, as can be judged from the number of

publications for each year in Fig. 2. For derived test oracles, many solutions have been proposed throughout this period. Initially, these solutions were primarily theoretical, such as Partial/Pseudo-Oracles [196] and Specification Inference [194]; empirical studies however followed in late 1990s.

For implicit test oracles, research into the solutions established before 1978 has continued, but at a slower pace than the other types of test oracles. For handling the lack of an automated test oracle, *Partition Testing* is a well-known technique that helps a human test oracle select tests. The trend line suggests that only recently have new techniques and concepts for tackling this problem started to emerge, with an explicit focus on the human oracle cost problem.

4   SPECIFIED TEST ORACLES

Specification is fundamental to computer science, so it is not surprising that a vast body of research has explored its use as a source of test oracle information. This topic could merit an entire survey on its own right. In this section, we provide an overview of this work. We also include here partial specifications of system behaviour such as assertions and models.

A specification defines, if possible using mathematical logic, the test oracle for particular domain. Thus, a specification language is a notation for defining a *specified test oracle* D, which judges whether the behaviour of a system conforms to a formal specification. Our formalism, defined in Section 2, is, itself, a specification language for specifying test oracles.

Over the last 30 years, many methods and formalisms for testing based on formal specification have been developed. They fall into four broad categories: model-based specification languages, state transition systems, assertions and contracts, and algebraic specifications. Model-based languages define models and a syntax that defines desired behavior in terms of its effect on the model. State transition systems focus on modeling the reaction of a system to stimuli, referred to as "transitions" in this particular formalism. Assertions and contracts are fragments of a specification language that are interleaved with statements of the implementation language and checked at runtime. Algebraic specifications define equations over a program's operations that hold when the program is correct.

4.1   Specification Languages

Specification languages define a mathematical model of a system's behaviour, and are equipped with a formal semantics that defines the meaning of each language construct in terms of the model. When used for testing, models do not usually fully specify the system, but seek to capture salient properties of a system so that test cases can be generated from or checked against them.

4.1.1   Model-Based Specification Languages

Model-based specification languages model a system as a collection of states and operations to alter these states, and are therefore also referred to as "state-based specifications" in the literature [101], [109], [182], [183]. Preconditions and postconditions constrain the system's operations. An operation's precondition imposes a necessary condition over the input states that must hold in a correct application of the operation; a postcondition defines the (usually strongest) effect the operation has on program state [109].

A variety of model-based specification languages exist, including Z [172], B [110], UML/OCL [31], VDM/VDM-SL [62], Alloy [101], and the LARCH family [71], which includes an algebraic specification sub-language. Broadly, these languages have evolved toward being more concrete, closer to the implementation languages programmers use to solve problems. Two reasons explain this phenomenon: the first is the effort to increase their adoption in industry by making them more familiar to practitioners and the second is to establish synergies between specification and implementation that facilitate development as iterative refinement. For instance, Z models disparate entities, like predicates, sets, state properties, and operations, through a single structuring mechanism, its schema construct; the B method, Z's successor, provides a richer array of less abstract language constructs.

Börger discusses how to use the abstract state machine formalism, a very general set-theoretic specification language geared toward the definition of functions, to define high level test oracles [29]. The models underlying specification languages can be very abstract, quite far from concrete execution output. For instance, it may be difficult to compute whether a model's postcondition for a function permits an observed concrete output. If this impedance mismatch can be overcome, by abstracting a system's concrete output or by concretizing a specification model's output, and if a specification's postconditions can be evaluated in finite time, they can serve as a test oracle [4].

Model-based specification languages, such as VDM, Z, and B can express invariants, which can drive testing. Any test case that causes a program to violate an invariant has discovered an incorrect behavior; therefore, these invariants are partial test oracles.

In search of a model-based specification language accessible to domain experts, Parnas et al. proposed TOG (Test Oracles Generator) from program documentation [142], [145], [148]. In their method, the documentation is written in fully formal tabular expressions in which the method signature, the external variables, and relation between its start and end states are specified [104]. Thus, test oracles can be automatically generated to check the outputs against the specified states of a program. The work by Parnas et al. has been developed over a considerable period of more than two decades [48], [59], [60], [144], [149], [190], [191].

4.1.2   State Transition Systems

State transition systems often present a graphical syntax, and focus on transitions between different states of the system. Here, states typically abstract sets of concrete state of the modeled system. State transition systems have been referred as visual languages in the literature [197]. A wide variety of state transition systems exist, including finite state machines [111], Mealy/Moore machines [111], I/O Automata [117], Labeled Transition Systems [180], SDL [54], Harel Statecharts [81], UML state machines [28], X-Machines [95], [96], Simulink/Stateflow [179] and PROMELA [97]. Mouchawrab et al. conducted a rigorous empirical evaluation of test oracle construction techniques using state transition systems [70], [137].

An important class of state transition systems have a finite set of states and are therefore particularly well-suited for automated reasoning about systems whose behaviour can be abstracted into states defined by a finite set of values [93]. State transition systems capture the behavior of a system under test as a set of states,[3] with transitions representing stimuli that cause the system to change state. State transition systems model the output of a system they abstract either as a property of the states (the final state in the case of Moore machines) or the transitions traversed (as with Mealy machines).

Models approximate a SUT, so behavioral differences between the two are inevitable. Some divergences, however, are spurious and falsely report testing failure. State-transition models are especially susceptible to this problem when modeling embedded systems, for which time of occurrence is critical. Recent work model tolerates spurious differences in time by "steering" model's evaluation: when the SUT and its model differ, the model is backtracked, and a steering action, like modifying timer value or changing inputs, is applied to reduce the distance, under a similarity measure [74].

Protocol conformance testing [72] and, later, model-based testing [183] motivated much of the work applying state transition systems to testing. Given a specification F as a state transition system, e.g. a finite state machine, a test case can be extracted from sequences of transitions in F. The transition labels of such a sequence define an input. A test oracle can then be constructed from F as follows: if F accepts the sequence and outputs some value, then so should the system under test; if F does not accept the input, then neither should the system under test.

Challenges remain, however, as the definition of conformity comes in different flavours, depending on whether the model is deterministic or non-deterministic and whether the behaviour of the system under test on a given test case is observable and can be interpreted at the same level of abstraction as the model's. The resulting flavours of conformity have been captured in alternate notions, in terms of whether the system under test is isomorphic to, equivalent to, or quasi-equivalent to F. These notions of conformity were defined in the mid-1990s in the famous survey paper by Lee and Yannakakis [111] among other notable papers, including those by Bochmann and Petrenko [26] and Tretmans [180].

4.2 Assertions and Contracts

An assertion is a boolean expression that is placed at a certain point in a program to check its behaviour at runtime. When an assertion evaluates to true, the program's behaviour is regarded "as intended" at the point of the assertion, for that particular execution; when an assertion evaluates to false, an error has been found in the program for that particular execution. It is obvious to see how assertions can be used as a test oracle.

The fact that assertions are embedded in an implementation language has two implications that differentiate them from specification languages. First, assertions can directly reference and define relations over program variables, reducing the impedance mismatch between specification and implementation, for the properties an assertion can express and check. In this sense, assertions are a natural consequence of the evolution of specification languages toward supporting development through iterative refinement. Second, they are typically written along with the code whose runtime behavior they check, as opposed to preceding implementation as specification languages tend to do.

Assertions have a long pedigree dating back to Turing [181], who first identified the need to separate the tester from the developer and suggested that they should communicate by means of assertions: the developer writing them and the tester checking them. Assertions gained significant attention as a means of capturing language semantics in the seminal work of Floyd [64] and Hoare [94] and subsequently were championed as a means of increasing code quality in the development of the contract-based programming approach, notably in the language Eiffel [135].

Widely used programming languages now routinely provide assertion constructs; for instance, C, C++, and Java provide a construct called *assert* and C# provides a *Debug. Assert* method. Moreover, a variety of systems have been independently developed for embedding assertions into a host programming languages, such as Anna [116] for Ada, APP [155] and Nana [119] for C languages.

In practice, assertion approaches can check only a limited set of properties at a certain point in a program [49]. Languages based on *design by contract* principles extend the expressivity of assertions by providing means to check contracts between *client* and *supplier* objects in the form of method pre- and post- conditions and class invariants. Eiffel was the first language to offer design by contract [135], a language feature that has since found its way into other languages, such as Java in the form of Java modeling language (JML) [139].

Cheon and Leavens showed how to construct an assertion-based test oracle on top of JML [45]. For more on assertion-based test oracles, see Coppit and Haddox-Schatz's evaluation [49], and, later, a method proposed by Cheon [44]. Both assertions and contracts are enforced observation activity that are embedded into the code. Araujo et al. provide a systematic evaluation of design by contract on a large industrial system [9] and using JML in particular [8]; Briand et al. showed how to support testing by instrumenting contracts [33].

4.3 Algebraic Specification Languages

Algebraic specification languages define a software module in terms of its interface, a signature consisting of sorts and operation symbols. Equational axioms specify the required properties of the operations; their equivalence is often computed using term rewriting [15]. Structuring facilities, which group sorts and operations, allow the composition of interfaces. Typically, these languages employ first-order logic to prove properties of the specification, like the correctness of refinements. Abstract data types (ADT), which combine data and operations over that data, are well-suited to algebraic specification.

One of the earliest algebraic specification systems, for implementing, specifying and testing ADTs, is DAISTS [69]. In this system, equational axioms generally equate a term-rewriting expression in a restricted dialect of ALGOL 60

---

3. Unfortunately, the term 'state' has different interpretation in the context of test oracles. Often, it refers to a 'snapshot' of the configuration of a system at some point during its execution; in context of state transition systems, however, 'state' typically refers to an abstraction of a set of configurations, as noted above.

against a function composition in the implementation language. For example, consider this axiom used in DAISTS:

$$Pop2(Stack\ S, EltType\ I):$$
$$Pop(Push(S, I)) = \textbf{if } Depth(S) = Limit$$
$$\textbf{then } Pop(S)$$
$$\textbf{else } S;$$

This axiom is taken from a specification that differentiates the accessor *Top*, which returns the top element of a stake without modifying the stack, and the mutator *Pop*, which returns a new stack lacking the previous top element. A test oracle simply executes both this axiom and its corresponding composition of implemented functions against a test suite: if they disagree, a failure has been found in the implementation or in the axiom; if they agree, we gain some assurance of their correctness.

Gaudel et al. [19], [20], [72], [73] were the first to provide a general testing theory founded on algebraic specification. Their idea is that an exhaustive test suite composed only of ground terms, i.e., terms with no free variables, would be sufficient to judge program correctness. This approach faces an immediate problem: the domain of each variable in a ground term might be infinite and generate an infinite number of test cases. Test suites, however, must be finite, a practical limitation to which all forms of testing are subject. The workaround is, of course, to abandon exhaustive coverage of all bindings of values to ground terms and select a finite subset of test cases [20].

Gaudel's theory focuses on observational equivalence. Observational inequivalence is, however, equally important [210]. For this reason, Frankl and Doong extended Gaudel's theory to express inequality as well an equality [52]. They proposed a notation that is suitable for object-oriented programs and developed an algebraic specification language called LOBAS and a test harness called ASTOOT. In addition to handling object-orientation, Frankl and Doong require classes to implement the testing method EQN that ASTOOT uses to check the equivalence or inequivalence of two instances of a given class. From the vantage point of an observer, an object has observable and unobservable, or hidden, state. Typically, the observable state of an object is its public fields and method return values. EQN enhances the testability of code and enables ASTOOT to approximate the observational equivalence of two objects on a sequence of messages, or method calls. When ASTOOT checks the equivalence of an object and a specification in LOBAS, it realizes a specified test oracle.

Expanding upon ASTOOT, Chen et al. [40], [41] built TACCLE, a tool that employs a white-box heuristic to generate a relevant, finite number of test cases. Their heuristic builds a data relevance graph that connects two fields of a class if one affects the other. They use this graph to consider only that can affect an observable attributes of a class when considering the (in)equivalence of two instances. Algebraic specification has been a fruitful line of research; many algebraic specification languages and tools exist, including Daistish [100], LOFT [122], CASL [11], CASCAT [205]. The projects have been evolving toward testing a wider array of entities, from ADTS, to classes, and most recently, components; they also differ in their degree of automation of test case generation and test harness creation. Bochmann et al. used LOTOS to realise test oracle functions from algebraic specifications [184]; most recently, Zhu also considered the use of algebraic specifications as test oracles [210].

4.3.1   Specified Test Oracle Challenges

Three challenges must be overcome to build specified test oracles. The first is the lack of a formal specification. Indeed, the other classes of test oracles, discussed in this survey, all address the problem of test oracle construction in the absence of a formal specification. Formal specifications models necessarily rely on abstraction that can lead to the second problem: imprecision, models that include infeasible behavior or that do not capture all the behavior relevant to checking a specification [68]. Finally, one must contend with the problem of interpreting model output and equating it to concrete program output.

Specified results are usually quite abstract, and the concrete test results of a program's executions may not be represented in a form that makes checking their equivalence to the specified result straightforward. Moreover, specified results can be partially represented or oversimplified. This is why Gaudel remarked that the existence of a formal specification does not guarantee the existence of a successful test driver [72]. Formulating concrete equivalence functions may be necessary to correctly interpret results [118]. In short, solutions to this problem of equivalence across abstraction levels depend largely on the degree of abstraction and, to a lesser extent, on the implementation of the system under test.

5   DERIVED TEST ORACLES

A *derived test oracle* distinguishes a system's correct from incorrect behavior based on information derived from various artefacts (e.g. documentation, system executions) or properties of the system under test, or other versions of it. Testers resort to derived test oracles when specified test oracles are unavailable, which is often the case, since specifications rapidly fall out of date when they exist at all. Of course, the derived test oracle might become a partial "specified test oracle", so that test oracles derived by the methods discussed in this section could migrate, over time, to become, those considered to be the "specified test oracles" of the previous section. For example, JWalk incrementally learns algebraic properties of the class under test [170]. It allows interactive confirmation from the tester, ensuring that the human is in the "'learning loop".

The following sections discuss research on deriving test oracles from development artefacts, beginning in Section 5.1 with pseudo-oracles and N-version programming, which focus on agreement among independent implementations. Section 5.2 then introduces metamorphic relations which focuses on relations that must hold among distinct executions of a single implementation. Regression testing, Section 5.3, focuses on relations that should hold across different versions of the SUT. Approaches for inferring models from system executions, including invariant inference and specification mining, are described in Section 5.4. Section 5.5 closes with a discussion of research into extracting test oracle information from textual documentation, like comments, specifications, and requirements.

5.1 Pseudo-Oracles

One of the earliest versions of a derived test oracle is the concept of a pseudo-oracle, introduced by Davis and Weyuker [50], as a means of addressing so-called *non-testable programs*:

> "Programs which were written in order to determine the answer in the first place. There would be no need to write such programs, if the correct answer were known." [196].

A pseudo-oracle is an alternative version of the program produced independently, e.g. by a different programming team or written in an entirely different programming language. In our formalism (Section 2), a pseudo-oracle is a test oracle D that accepts test activity sequences of the form

$$f_1(x)o_1 f_2(x)o_2 : [f_1 \neq f_2 \wedge o_1 = o_2], \quad (1)$$

where $f_1, f_2 \in C$, the components of the SUT (Section 2), are alternative, independently produced, versions of the SUT on the same value. We draw the reader's attention to the similarity between pseudo-oracles and algebraic specification systems (Section 4.3), like DIASTIS, where the function composition expression in the implementation language and the term-rewriting expression are distinct implementations whose output must agree and form a pseudo-oracle.

A similar idea exists in fault-tolerant computing, referred to as multi- or N-version programming [13], [14], where the software is implemented in multiple ways and executed in parallel. Where results differ at run-time, a "voting" mechanism decides which output to use. In our formalism, an N-version test oracle accepts test activities of the following form:

$$\begin{aligned} &f_1(x)o_1 f_2(x)o_2 \cdots f_k(x)o_k : \\ &[\forall i, j \in [1..k], \ i \neq j \Rightarrow f_i \neq f_j \quad (2) \\ &\wedge \arg\max_{o_i} m(o_i) \geq t]. \end{aligned}$$

In Equation (2), the outputs form a multiset and m is the multiplicity, or number of repetitions of an element in the multiset. The $\arg\max$ operator finds the argument that maximizes a function's output, here an output with greatest multiplicity. Finally, the maximum multiplicity is compared against the threshold t. We can now define a N-version test oracle as $D_{nv}(w, x)$ where w obeys Equation (2) with t bound to x. Then $D_{\text{maj}}(w) = D_{nv}(w, \lceil \frac{k}{2} \rceil)$ is an N-version oracle that requires a majority of the outputs to agree and $D_{pso}(w) = D_{nv}(w, k)$ generalizes pseudo oracles to agreement across k implementations.

More recently, Feldt [58] investigated the possibility of automatically producing different versions using genetic programming, and McMinn [127] explored the idea of producing different software versions for testing through program transformation and the swapping of different software elements with those of a similar specification.

5.2 Metamorphic Relations

For the SUT p that implements the function f, a *metamorphic relation* is a relation over applications of f that we expect to hold across *multiple* executions of p. Suppose $f(x) = e^x$,

then $e^a e^{-a} = 1$ is a metamorphic relation. Under this metamorphic relation, `p(0.3) * p(-0.3) = 1` will hold if p is correct [43]. The key idea is that reasoning about the properties of f will lead us to relations that its implementation p must obey.

Metamorphic testing is a process of exploiting metamorphic relations to generate partial test oracles for *follow-up* test cases: it checks important properties of the SUT after certain test cases are executed [36]. Although metamorphic relations are properties of the ground truth, the correct phenomenon (f in the example above) that a SUT seeks to implement and could be considered a mechanism for creating specified test oracles. We have placed them with derived test oracles, because, in practice, metamorphic relations are usually manually inferred from a white-box inspection of a SUT.

Metamorphic relations differ from algebraic specifications in that a metamorphic relation relates different executions, not necessarily on the same input, of the same implementation relative to its specification, while algebraic specifications equates two distinct implementations of the specification, one written in an implementation language and the other written in formalism free of implementation details, usually term rewriting [15].

Under the formalism of Section 2, a metamorphic relation is

$$f(x_1)o_1 f(x_2)o_2 \cdots f(i_k)o_k : [expr \wedge k \geq 2],$$

where $expr$ is a constraint, usually arithmetic, over the inputs x_i and o_x. This definition makes clear that a metamorphic relation is a constraint on the values of stimulating the single SUT f at least twice, observing the responses, and imposing a constraint on how they interrelate. In contrast, algebraic specification is a type of pseudo-oracle, as specified in Equation (1), which stimulates two distinct implementations on the same value, requiring their output to be equivalent.

It is often thought that metamorphic relations need to concern numerical properties that can be captured by arithmetic equations, but metamorphic testing is, in fact, more general. For example, Zhou et al. [209] used metamorphic testing to test search engines such as Google and Yahoo!, where the relations considered are clearly non-numeric. Zhou et al. build metamorphic relations in terms of the consistency of search results. A motivating example they give is of searching for a paper in the ACM digital library: two attempts, the second quoted, using advanced search fail, but a general search identical to the first succeeds. Using this insight, the authors build metamorphic relations, like $R_{OR} : A_1 = (A_2 \cup A_3) \Rightarrow |A_2| \leq |A_1|$, where the A_i are sets of web pages returned by queries. Metamorphic testing is also means of testing Weyuker's "non-testable programs", introduced in the last section.

When the SUT is nondeterministic, such as a classifier whose exact output varies from run to run, defining metamorphic relations solely in terms of output equality is usually insufficient during metamorphic testing. Murphy et al. [138], [139] investigate relations other than equality, like set intersection, to relate the output of stochastic machine learning algorithms, such as classifiers. Guderlei and Mayer

introduced statistical metamorphic testing, where the relations for test output are checked using statistical analysis [80], a technique later exploited to apply metamorphic testing to stochastic optimisation algorithms [203].

The biggest challenge in metamorphic testing is automating the discovery of metamorphic relations. Some of those in the literature are mathematical [36], [37], [42] or combinatorial [138], [139], [160], [203]. Work on the discovery of algebraic specifications [88] and JWalk's lazy systematic unit testing, in which the specification is lazily, and incrementally, learned through interactions between JWalk and the developer [170] might be suitable for adaptation to the discovery metamorphic relations. For instance, the programmer's development environment might track relationships among the output of test cases run during development, and propose ones that hold across many runs to the developer as possible metamorphic relations. Work has already begun that exploits domain knowledge to formulate metamorphic relations [38], but it is still at an early stage and not yet automated.

5.3 Regression Test Suites

Regression testing aims to detect whether the modifications made to the new version of a SUT have disrupted existing functionality [204]. It rests on the implicit assumption that the previous version can serve as an oracle for existing functionality.

For corrective modifications, desired functionality remains the same so the test oracle for version i, D_i, can serve as the next version's test oracle, D_{i+1}. Corrective modifications may fail to correct the problem they seek to address or disrupt existing functionality; test oracles may be constructed for these issues by symbolically comparing the execution of the faulty version against the newer, allegedly fixed version [79]. Orstra generates assertion-based test oracles by observing the program states of the previous version while executing the regression test suite [199]. The regression test suite, now augmented with assertions, is then applied to the newer version. Similarly, *spectra*-based approaches use the program and value spectra obtained from the original version to detect regression faults in the newer versions [86], [200].

For perfective modifications, those that add new features to the SUT, D_i must be modified to cater for newly added behaviours, i.e. $D_{i+1} = D_i \cup \Delta D$. Test suite augmentation techniques specialise in identifying and generating ΔD [6], [131], [202]. However, more work is required to develop these augmentation techniques so that they augment, not merely the test input, but also the expected output. In this way, test suite augmentation could be extended to augment the existing *oracles* as well as the test data.

Changes in the specification, which is deemed to fail to meet requirements perhaps because the requirements have themselves changed, drives another class of modifications. These changes are generally regarded as "perfective" maintenance in the literature but no distinction is made between perfections that add new functionality to code (without changing requirements) and those changes that arise due to changed requirements (or incorrect specifications).

Our formalisation of test oracles in Section 2 forces a distinction of these two categories of perfective maintenance,

since the two have profoundly different consequences for test oracles. We therefore refer to this new category of perfective maintenance as "changed requirements". Recall that, for the function $f : X \to Y$, $\mathrm{dom}(f) = X$. For changed requirements:

$$\exists \alpha \cdot D_{i+1}(\alpha) \neq D_i(\alpha),$$

which implies, of course, $\mathrm{dom}(D_{i+1}) \cap \mathrm{dom}(D_i) \neq \emptyset$ and the new test oracle cannot simply union the new behavior with the old test oracle. Instead, we have

$$D_{i+1} = \begin{cases} \Delta D & \text{if } \alpha \in \mathrm{dom}(\Delta D) \\ D_i & \text{otherwise.} \end{cases}$$

5.4 System Executions

A system execution trace can be exploited to derive test oracles or to reduce the cost of a human test oracle by aligning an incorrect execution against the expected execution, as expressed in temporal logic [51]. This section discusses the two main techniques for deriving test oracles from traces—invariant detection and specification mining. Derived test oracles can be built on both techniques to automatically check expected behaviour similar to assertion-based specification, discussed in Section 4.2.

5.4.1 Invariant Detection

Program behaviours can be automatically checked against invariants. Thus, invariants can serve as test oracles to help determine the correct and incorrect outputs.

When invariants are not available for a program in advance, they can be learned from the program (semi-) automatically. A well-known technique proposed by Ernst et al. [56], implemented in the Daikon tool [55], is to execute a program on a collection of inputs (test cases) against a collection of potential invariants. The invariants are instantiated by binding their variables to the program's variables. Daikon then dynamically infers *likely* invariants from those invariants not violated during the program executions over the inputs. The inferred invariants capture program behaviours, and thus can be used to check program correctness. For example, in regression testing, invariants inferred from the previous version can be checked as to whether they still hold in the new version.

In our formalism, Daikon invariant detection can define an unsound test oracle that gathers likely invariants from the prefix of a testing activity sequence, then enforces those invariants over its suffix. Let I_j be the set of likely invariants at observation j; I_0 are the initial invariants; for the test activity sequence $r_1 r_2 \ldots r_n$, $I_n = \{x \in I \mid \forall i \in [1..n], r_i \models x\}$, where $\models$ is logical entailment. Thus, we take an observation to define a binding of the variables in the world under which a likely invariant either holds or does not: only those likely invariants remain that no observation invalidates. In the suffix $r_{n+1} r_{n+2} \ldots r_m$, the test oracle then changes gear and accepts only those activities whose response observations obey I_n, i.e. $r_i : [r_i \models I_n], i > n$.

Invariant detection can be computationally expensive, so incremental [22], [171] and light weight static analyses [39], [63] have been brought to bear. A technical report summarises various dynamic analysis techniques [157]. Model

inference [90], [187] could also be regarded as a form of invariant generation in which the invariant is expressed as a model (typically as an FSM). Ratcliff et al. used Search-Based Software Engineering (SBSE) [84] to search for invariants, guided by mutation testing [153].

The accuracy of inferred invariants depends in part on the quality and completeness of the test cases; additional test cases might provide new data from which more accurate invariants can be inferred [56]. Nevertheless, inferring "perfect" invariants is almost impossible with the current state of the art, which tends to frequently infer incorrect or irrelevant invariants [151]. Wei et al. recently leveraged existing contracts in Eiffel code to infer postconditions on commands (as opposed to queries) involving quantification or implications whose premises are conjunctions of formulae [192], [193].

Human intervention can, of course, be used to filter the resulting invariants, i.e., retaining the correct ones and discarding the rest. However, manual filtering is error-prone and the misclassification of invariants is frequent. In a recent empirical study, Staats et al. found that half of the incorrect invariants Daikon inferred from a set of Java programs were misclassified [175]. Despite these issues, research on the dynamic inference of program invariants has exhibited strong momentum in the recent past with the primary focus on its application to test generation [10], [141], [207].

5.4.2 Specification Mining

Specification mining or inference infers a formal model of program behaviour from a set of observations. In terms of our formalism, a test oracle can enforce these formal models over test activities. In her seminal work on using inference to assess test data adequacy, Weyuker connected inference and testing as inverse processes [194]. The testing process starts with a program, and looks for I/O pairs that characterise every aspect of both the intended and actual behaviours, while inference starts with a set of I/O pairs, and derives a program to fit the given behaviour. Weyuker defined this relation for assessing test adequacy which can be stated informally as follows.

A set of I/O pairs T is an *inference adequate test set* for the program P intended to fulfil specification S iff the program I_T inferred from T (using some inference procedure) is equivalent to both P and S. Any difference would imply that the inferred program is not equivalent to the actual program and, therefore, that the test set T used to infer the program P is inadequate.

This inference procedure mainly depends upon the set of I/O pairs used to infer behaviours. These pairs can be obtained from system executions either passively, e.g., by runtime monitoring, or actively, e.g., by querying the system [105]. However, equivalence checking is undecidable in general, and therefore inference is only possible for programs in a restricted class, such as those whose behaviour can be modelled by finite state machines [194]. With this, equivalence can be accomplished by experiment [89]. Nevertheless, serious practical limitations are associated with such experiments (see the survey by Lee and Yannakakis [111] for complete discussion).

The marriage between inference and testing has produced wealth of techniques, especially in the context of "black-box" systems, when source code/behavioural models are unavailable. Most work has applied L^*, a well-known learning algorithm, to learn a black-box system B as a finite state machine with n states [7]. The algorithm infers an FSM by iteratively querying B and observing the corresponding outputs. A string distinguishes two FSMs when only one of the two machines ends in a final state upon consuming the string. At each iteration, an inferred model M_i with $i < n$ states is given. Then, the model is refined with the help of a string that distinguishes B and M_i to produce a new model, until the number of states reaches n.

Lee and Yannakakis [111] showed how to use L^* for conformance testing of B with a specification S. Suppose L^* starts by inferring a model M_i, then we compute a string that distinguishes M_i from S and refine M_i through the algorithm. If, for $i = n$, M_n is S, then we declare B to be correct, otherwise faulty.

Apart from conformance testing, inference techniques have been used to guide test generation to focus on particular system behavior and to reduce the scope of analysis. For example, Li et al. applied L^* to the integration testing of a system of black-box components [113]. Their analysis architecture derives a test oracle from a test suite by using L^* to infer a model of the systems from dynamically observing system's behavior; this model is then searched to find incorrect behaviors, such as deadlocks, and used to verify the system's behaviour under fuzz testing (Section 6).

To find concurrency issues in asynchronous black-box systems, Groz et al. proposed an approach that extracts behavioural models from systems through active learning techniques [78] and then performs reachability analysis on the models [27] to detect issues, notably races.

Further work in this context has been compiled by Shahbaz [165] with industrial applications. Similar applications of inference can be found in system analysis [21], [78], [134], [188], [189], component interaction testing [114], [121], regression testing [200], security testing [168] and verification [53], [77], [147].

Zheng et al. [208] extract item sets from web search queries and their results, then apply association rule mining to infer rules. From these rules, they construct derived test oracles for web search engines, which had been thought to be untestable. Image segmentation delineates objects of interest in an image; implementing segmentation programs is a tedious, iterative process. Frouchni et al. successfully apply semi-supervised machine learning to create test oracles for image segmentation programs [67]. Memon et al. [132], [133], [198] introduced and developed the GUITAR tool, which has been evaluated by treating the current version of the SUT as correct, inferring the specification, and then executing the generated test inputs. Artificial Neural Networks have also been applied to learn system behaviour and detect deviations from it [162], [163].

The majority of specification mining techniques adopt finite state machines as the output format to capture the functional behaviour of the SUT [21], [27], [53], [77], [78], [89], [111], [113], [134], [147], [165], [168], [189], sometimes extended with temporal constraints [188] or data constraints [114], [121] which are, in turn, inferred by Daikon [56]. Büchi automata have been used to check properties against black-box systems [147]. Annotated call trees have been

used to represent the program behaviour of different versions in the regression testing context [200]. GUI widgets have been directly modelled with objects and properties for testing [132], [133], [198]. Artificial Neural Nets and machine learning classifiers have been used to learn the expected behaviour of SUT [67], [162], [163]. For dynamic and fuzzy behaviours such as the result of web search engine queries, association rules between input (query) and output (search result strings) have been used as the format of an inferred oracle [208].

5.5 Textual Documentation

Textual documentation ranges from natural language descriptions of requirements to structured documents detailing the functionalities of APIs. These documents describe the functionalities expected from the SUT to varying degrees, and can therefore serve as a basis for generating test oracles. They are usually informal, intended for other humans, not to support formal logical or mathematical reasoning. Thus, they are often partial and ambiguous, in contrast to specification languages. Their importance for test oracle construction rests on the fact that developers are more likely to write them than formal specifications. In other words, the documentation defines the constraints that the test oracle D, as defined in Section 2, enforces over testing activities.

At first sight, it may seem impossible to derive test oracles automatically because natural languages are inherently ambiguous and textual documentation is often imprecise and inconsistent. The use of textual documentation has often been limited to humans in practical testing applications [143]. However, some partial automation can assist the human in testing using documentation as a source of test oracle information.

Two approaches have been explored. The first category builds techniques to construct a formal specification out of an informal, textual artefact, such as an informal textual specification, user and developer documentation, and even source code comments. The second restricts a natural language to a semi-formal fragment amenable to automatic processing. Next, we present representative examples of each approach.

5.5.1 Converting Text into Specifications

Prowell and Poore [152] introduced a sequential enumeration method for developing a formal specification from an informal one. The method systematically enumerates all sequences from the input domain and maps the corresponding outputs to produce an arguably complete, consistent, and correct specification. However, it can suffer from an exponential explosion in the number of input/output sequences. Prowell and Poore employ abstraction techniques to control this explosion. The end result is a formal specification that can be transferred into a number of notations, e.g., state transition systems. A notable benefit of this approach is that it tends to discover many inconsistent and missing requirements, making the specification more complete and precise.

5.5.2 Restricting Natural Language

Restrictions on a natural language reduce complexities in its grammar and lexicon and allow the expression of requirements in a concise vocabulary with minimal ambiguity. This, in turn, eases the interpretation of documents and makes the automatic derivation of test oracles possible. The researchers who have proposed specification languages based on (semi-) formal subsets of a natural language are motivated by the fact that model-based specification languages have not seen wide-spread adoption, and believe the reason is the inaccessibility their formalism and set-theoretic underpinnings to the average programmer.

Schwitter introduced a computer-processable, restricted natural language called PENG [159]. It covers a strict subset of standard English with a restricted grammar and a domain specific lexicon for content words and predefined function words. Documents written in PENG can be translated deterministically into first-order predicate logic. Schwitter et al. [30] provided guidelines for writing test scenarios in PENG that can automatically judge the correctness of program behaviours.

6 IMPLICIT TEST ORACLES

An *implicit test oracle* is one that relies on general, implicit knowledge to distinguish between a system's correct and incorrect behaviour. This generally true implicit knowledge includes such facts as "buffer overflows and segfaults are nearly always errors". The critical aspect of an implicit test oracle is that it requires neither domain knowledge nor a formal specification to implement, and it applies to nearly all programs.

Implicit test oracle can be built on any procedure that detects anomalies such as abnormal termination due to a crash or an execution failure [34], [167]. This is because such anomalies are *blatant faults*; that is, no more information is required to ascertain whether the program behaved correctly or not. Under our formalism, an implicit oracle defines a subset of stimulus and response relations as guaranteed failures, in some context.

Implicit test oracles are not universal. Behaviours abnormal for one system in one context may be normal for that system in a different context or normal for a different system. Even crashing *may* be considered acceptable, or even desired behaviour, as in systems designed to find crashes.

Research on implicit oracles is evident from early work in software engineering. The very first work in this context was related to deadlock, livelock and race detection to counter system concurrency issues [24], [106], [185], [16], [169]. Similarly, research on testing non-functional attributes have garnered much attention since the advent of the object-oriented paradigm. In performance testing, system throughput metrics can highlight degradation errors [120], [123], as when a server fails to respond when a number of requests are sent simultaneously. A case study by Weyuker and Vokolos showed how a process with excessive CPU usage caused service delays and disruptions [195]. Similarly, test oracles for memory leaks can be built on a profiling technique that detects dangling references during the run of a program [12], [57], [87], [211]. For example, Xie and Aiken proposed a boolean constraint system to represent the dynamically allocated objects in a program [201]. Their system raises an alarm when an object becomes unreachable but has not yet been deallocated.

Fuzzing is an effective way to find implicit anomalies, such a crashes [136]. The main idea is to generate random, or "fuzz", inputs and feed them to the system to find anomalies. This works because the implicit specification usually holds over all inputs, unlike explicit specifications which tend to relate subsets of inputs to outputs. If an anomaly is detected, the fuzz tester reports it along with the input that triggers it. Fuzzing is commonly used to detect security vulnerabilities, such as buffer overflows, memory leaks, unhandled exceptions, denial of service, etc. [18], [177].

Other work has focused on developing patterns to detect anomalies. For instance, Ricca and Tonella [154] considered a subset of the anomalies that Web applications can harbor, such as navigation problems, hyperlink inconsistencies, etc. In their empirical study, 60 percent of the web applications exhibited anomalies and execution failures.

7 THE HUMAN ORACLE PROBLEM

The above sections give solutions to the test oracle problem when some artefact exists that can serve as the foundation for either a full or partial test oracle. In many cases, however, no such artefact exists so a human tester must verify whether software behaviour is correct given some stimuli. Despite the lack of an automated test oracle, software engineering research can still play a key role: finding ways to reduce the effort that the human tester has to expend in directly creating, or in being, the test oracle.

This effort is referred to as the Human Oracle Cost [125]. It aims to reduce the cost of human involvement along two dimensions: 1) writing test oracles and 2) evaluating test outcomes. Concerning the first dimension, the work of Staats et al. is a representative. They seek to reduce the human oracle cost by guiding human testers to those parts of the code they need to focus on when writing test oracles [173]. This reduces the cost of test oracle construction, rather than reducing the cost of a human involvement in testing in the absence of an automated test oracle. Additional recent work on test oracle construction includes Dodona, a tool that suggests oracle data to a human who then decides whether to use it to define a test oracle realized as a Java unit test [115]. Dodona infers relations among program variables during execution, using network centrality analysis and data flow.

Research that seeks to reduce the human oracle cost broadly focuses on finding a *quantitative* reduction in the amount of work the tester has to do for the same amount of test coverage or finding a *qualitative* reduction in the work needed to understand and evaluate test cases.

7.1 Quantitative Human Oracle Cost

Test suites can be unnecessarily large, covering few test goals in each individual test case. Additionally, the test cases themselves may be unnecessarily long—for example containing large numbers of method calls, many of which do not contribute to the overall test case. The goal of *quantitative human oracle cost reduction* is to reduce test suite and test case size so as to maximise the benefit of each test case and each component of that test case. This consequently reduces the amount of manual checking effort that is required on behalf of a human tester performing the role of a test oracle. Cast in

terms of our formalism, quantitative reduction aims to partition the set of test activity sequences so the human need only consider representative sequences, while test case reduction aims to shorten test activity sequences.

7.1.1 Test Suite Reduction

Traditionally, test suite reduction has been applied as a post-processing step to an existing test suite, e.g. the work of Harrold et al, [85], Offutt et al. [140] and Rothermel et al. [156]. Recent work in the search-based testing literature has sought to combine test input generation and test suite reduction into one phase to produce smaller test suites.

Harman et al. proposed a technique for generating test cases that penetrate the deepest levels of the control dependence graph for the program, in order to create test cases that exercise as many elements of the program as possible [82]. Ferrer et al. [61] attack a multi-objective version of the problem in which they sought to simultaneously maximize branch coverage and minimize test suite size; their focus was not this problem per se, but its use to compare a number of multi-objective optimisation algorithms, including the well-known Non-dominated Sorting Genetic Algorithm II (NSGA-II), Strength Pareto EA 2 (SPEA2), and MOCell. On a series of randomly-generated programs and small benchmarks, they found MOCell performed best.

Taylor et al. [178] use an inferred model as a semantic test oracle to shrink a test suite. Fraser and Arcuri [65] generate test suites for Java using their EvoSuite tool. By generating the entire suite at once, they are able to simultaneously maximize coverage and minimize test suite size, thereby aiding human oracles and alleviating the human oracle cost problem.

7.1.2 Test Case Reduction

When using randomised algorithms for generating test cases for object-oriented systems, individual test cases can generate very long traces very quickly—consisting of a large number of method calls that do not actually contribute to a specific test goal (e.g. the coverage of a particular branch). Such method calls unnecessarily increase test oracle cost, so Leitner et al. remove such calls [112] using Zeller's and Hildebrandt's Delta Debugging [206]. JWalk simplifies test sequences by removing side-effect free functions from them, thereby reducing test oracle costs where the human is the test oracle [170]. Quick tests seek to efficiently spend a small test budget by building test suites whose execution is fast enough for it to be run after compilations [76]. These quick tests must be likely to trigger bugs and therefore generate short traces, which, as a result, are easier for humans to comprehend.

7.2 Qualitative Human Oracle Cost

Human oracle costs may also be minimised from a qualitative perspective. That is, the extent to which test cases, more generally testing activities, may be easily understood and processed by a human. The input profile of a SUT is the distribution of inputs it actually processes when running in its operational environment. Learning an input profile requires domain knowledge. If such domain knowledge is not built into the test data generation process, machine-generated test data tend to be drawn from a different distribution over

the SUT's inputs than its input profile. While this may be beneficial for trapping certain types of faults, the utility of the approach decreases when test oracle costs are taken into account, since the tester must invest time *comprehending* the scenario represented by test data in order to correctly evaluate the corresponding program output. Arbitrary inputs are much harder to understand than recognisable pieces of data, thus adding time to the checking process.

All approaches to qualitatively alleviating the human oracle cost problem incorporate human knowledge to improve the understandability of test cases. The three approaches we cover are

1) augmenting test suites designed by the developers;
2) computing "realistic" inputs from web pages, web services, and natural language; and
3) mining usage patterns to replicate them in the test cases.

In order to improve the readability of automatically-generated test cases, McMinn et al. propose the incorporation of human knowledge into the test data generation process [125]. With search-based approaches, they proposed injecting this knowledge by "seeding" the algorithm with test cases that may have originated from a human source such as a "sanity check" performed by the programmer, an already existing, partial test suite, or input–output examples generated by programming paradigms that involve the developer in computation, like prorogued programming [2].

The generation of string test data is particularly problematic for automatic test data generators, which tend to generate nonsensical strings. These nonsensical strings are, of course, a form of fuzz testing (Section 6) and good for exploring uncommon, shallow code paths and finding corner cases, but they are unlikely to exercise functionality deeper in a program's control flow. This is because string comparisons in control expressions are usually stronger than numerical comparisons, making one of a control point's branches much less likely to traverse via uniform fuzzing. We see here the seminal computer science trade-off between breadth first and depth first search in the choice between fuzz testing with nonsensical inputs and testing with realistic inputs.

Bozkurt and Harman, introduced the idea of mining web services for realistic test inputs, using the outputs of known and trusted test services as more realistic inputs to the service under test [32]. The idea is that realistic test cases are more likely to reveal faults that developers care about and yield test cases that are more readily understood. McMinn et al. also mine the web for realistic test cases. They proposed mining strings from the web to assist in the test generation process [129], [166]. Since web page content is generally the result of human effort, the strings contained therein tend to be real words or phrases with high degrees of semantic and domain relevant context that can thus be used as sources of realistic test data.

Afshan et al. [1] combine a natural language model and metahueristics, strategies that guide a search process [25], to help generate readable strings. The language model scores how likely a string is to belong to a language based on the character combinations. Incorporating this probability score

into a fitness function, a metaheuristic search can not only cover test goals, but generate string inputs that are more comprehensible than the arbitrary strings generated by the previous state of the art. Over a number of case studies, Afshan et al. found that human oracles more accurately and more quickly evaluated their test strings.

Fraser and Zeller [66] improve the familiarity of test cases by mining the software under test for common usage patterns of APIs. They then seek to replicate these patterns in generated test cases. In this way, the scenarios generated are more likely to be realistic and represent actual usages of the software under test.

7.3 Crowdsourcing the Test Oracle

A recent approach to handling the lack of a test oracle is to outsource the problem to an online service to which large numbers of people can provide answers—i.e., through crowdsourcing. Pastore et al. [146] demonstrated the feasibility of the approach but noted problems in presenting the test problem to the crowd such that it could be easily understood, and the need to provide sufficient code documentation so that the crowd could determine correct outputs from incorrect ones. In these experiments, crowdsourcing was performed by submitting tasks to a generic crowdsourcing platform—Amazon's Mechanical Turk.[4] However, some dedicated crowdsourcing services now exist for the testing of mobile applications. They specifically address the problem of the exploding number of devices on which a mobile application may run, and which the developer or tester may not own, but which may be possessed by the crowd at large. Examples of these services include Mob4Hire,[5] MobTest[6] and uTest.[7]

8 FUTURE DIRECTIONS AND CONCLUSION

This paper has provided a comprehensive survey of test oracles, covering specified, derived and implicit oracles and techniques that cater for the absence of test oracles. The paper has also analyzed publication trends in the test oracle domain. This paper has necessarily focused on the traditional approaches to the test oracle problem. Much work on test oracles remains to be done. In addition to research deepening and interconnecting these approaches, test oracle problem is open to new research directions. We close with a discussion of two of these that we find noteworthy and promising: test oracle reuse and test oracle metrics.

As this survey has shown, test oracles are difficult to construct. Oracle reuse is therefore an important problem that merits attention. Two promising approaches to oracle reuse are generalizations of reliable reset and the sharing of oracular data across software product lines (SPLs). Generalizing reliable reset to arbitrary states allows the interconnection of different versions of a program, so we can build test oracles that based on older versions of a program, using generalized reliable reset to ignore or handle new inputs and functionality. SPLs are sets of related versions of a system [47]. A product

---

4. http://www.mturk.com
5. http://www.mob4hire.com
6. http://www.mobtest.com
7. http://www.utest.com/

line can be thought of as a tree of related software products in which branches contain new alternative versions of the system, each of which shares some core functionality enjoyed by a base version. Research on test oracles should seek to leverage these SPL trees to define trees of test oracles that share oracular data where possible.

Work has already begun on using test oracle as the measure of how well the program has been tested (a kind of test oracle coverage) [103], [176], [186] and measures of oracles such as assessing the quality of assertions [158]. More work is needed. "Oracle metrics" is a challenge to, and an opportunity for, the "software metrics" community. In a world in which test oracles become more prevalent, it will be important for testers to be able to assess the features offered by alternative test oracles.

A repository of papers on test oracles accompanies this paper at http://crestweb.cs.ucl.ac.uk/resources/oracle_repository.

ACKNOWLEDGMENTS

REFERENCES

[1] S. Afshan, P. McMinn, and M. Stevenson, "Evolving readable string test inputs using a natural language model to reduce human oracle cost," in *Proc. Int. Conf. Softw. Testing, Verification Validation*, Mar. 2013, pp. 352–361.

[2] M. Afshari, E. T. Barr, and Z. Su, "Liberating the programmer with prorogued programming," in *Proc ACM Int. Symp. New Ideas, New Paradigms, Reflections Programm. Softw.*, 2012, pp. 11–26.

[3] W. Afzal, R. Torkar, and R. Feldt, "A systematic review of search-based testing for non-functional system properties," *Inf. Softw. Technol.*, vol. 51, no. 6, pp. 957–976, 2009.

[4] B. K. Aichernig, "Automated black-box testing with abstract VDM oracles," in *Proc. 18th Int. Conf. Comput. Comput. Safety, Rel. Security*, 1999, pp. 250–259.

[5] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege, "A systematic review of the application and empirical investigation of search-based test-case generation," *IEEE Trans. Softw. Eng.*, vol. 36, no. 6, pp. 742–762, Nov. 2010.

[6] N. Alshahwan and M. Harman, "Automated session data repair for web application regression testing," in *Proc. Int. Conf. Softw. Testing, Verification, Validation*, 2008, pp. 298–307.

[7] D. Angluin, "Learning regular sets from queries and counter-examples," *Inf. Comput.*, vol. 75, no. 2, pp. 87–106, 1987.

[8] W. Araujo, L. C. Briand, and Y. Labiche, "Enabling the runtime assertion checking of concurrent contracts for the Java modeling language," in *Proc. 33rd Int. Conf. Softw. Eng.*, 2011, pp. 786–795.

[9] W. Araujo, L. C. Briand, and Y. Labiche, "On the effectiveness of contracts as test oracles in the detection and diagnosis of race conditions and deadlocks in concurrent object-oriented software," in *Proc. Int. Symp. Empirical Softw. Eng. Meas.*, 2011, pp. 10–19.

[10] S. Artzi, M. D. Ernst, A. Kieżun, C. Pacheco, and J. H. Perkins, "Finding the needles in the haystack: Generating legal test inputs for object-oriented programs," in *Proc. 1st Workshop Model-Based Testing Object-Oriented Syst.*, Oct. 23, 2006.

[11] E. Astesiano, M. Bidoit, H. Kirchner, B. Krieg-Brückner, P. D. Mosses, D. Sannella, and A. Tarlecki, "CASL: The common algebraic specification language," *Theor. Comput. Sci.*, vol. 286, no. 2, pp. 153–196, 2002.

[12] T. M. Austin, S. E. Breach, and G. S. Sohi, "Efficient detection of all pointer and array access errors," in *Proc. ACM SIGPLAN 1994 Conf. Programm. Lang. Des. Implementation*, 1994, pp. 290–301.

[13] A. Avizienis, "The N-version approach to fault-tolerant software," *IEEE Trans. Softw. Eng.*, vol. 11, no. 12, pp. 1491–1501, Dec. 1985.

[14] A. Avizienis and L. Chen, "On the implementation of N-version programming for software fault-tolerance during execution," in *Proc. 1st Int. Comput. Softw. Appl. Conf.*, 1977, pp. 149–155.

[15] F. Baader, and T. Nipkow, *Term Rewriting and All That*. New York, NY, USA: Cambridge Univ. Press, 1998.

[16] A. F. Babich, "Proving total correctness of parallel programs," *IEEE Trans. Softw. Eng.*, vol. 5, no. 6, pp. 558–574, Nov. 1979.

[17] L. Baresi and M. Young. (2001, Aug.). Test oracles. Univ. of Oregon, Dept. Comput. Inform. Sci., Eugene, OR, USA. Tech. Rep. CIS-TR-01-02. [Online]. Available: http://www.cs.uoregon.edu/~michal/pubs/oracles.html

[18] S. Bekrar, C. Bekrar, R. Groz, and L. Mounier, "Finding software vulnerabilities by smart fuzzing," in *IEEE 4th Int. Conf. Softw. Testing, Verification Validation*, 2011, pp. 427–430.

[19] G. Bernot, "Testing against formal specifications: A theoretical view," in *Proc. Int. Joint Conf. Theory Prac. Softw. Develop. Adv. Distrib. Comput. Colloquium Combining Paradigms Softw. Develop.*, 1991, pp. 99–119.

[20] G. Bernot, M. C. Gaudel, and B. Marre, "Software testing based on formal specifications: A theory and a tool," *Softw. Eng. J.*, vol. 6, no. 6, pp. 387–405, Nov. 1991.

[21] A. Bertolino, P. Inverardi, P. Pelliccione, and M. Tivoli, "Automatic synthesis of behavior protocols for composable web-services," in *Proc. 7th Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng.*, 2009, pp. 141–150.

[22] D. Beyer, T. Henzinger, R. Jhala, and R. Majumdar, "Checking memory safety with Blast," in *Proc. 8th Int. Conf. Held as Part Joint Eur. Conf. Theory Pract. Softw. Conf. Fundam. Approaches Softw. Eng.*, 2005, pp. 2–18.

[23] R. Binder, *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Reading, MA, USA: Addison-Wesley, 2000.

[24] A. Blikle, "Proving programs by sets of computations," in *Proc. 3rd Symp. Math. Found. Comput. Sci.*, 1975, pp. 333–358.

[25] C. Blum and A. Roli, "Metaheuristics in combinatorial optimization: Overview and conceptual comparison," *ACM Comput. Surveys*, vol. 35, no. 3, pp. 268–308, 2003.

[26] G. V. Bochmann and A. Petrenko, "Protocol testing: Review of methods and relevance for software testing," in *Proc. ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 1994, pp. 109–124.

[27] G. V. Bochmann, "Finite state description of communication protocols," *Comput. Netw.*, vol. 2, no. 4, pp. 361–372, 1978.

[28] E. Börger, A. Cavarra, and E. Riccobene, "Modeling the dynamics of UML state machines," in *Abstract State Machines-Theory and Applications*. New York, NY, USA: Springer, 2000, pp. 167–186.

[29] E. Börger, "High level system design and analysis using abstract state machines," in *Proc. Int. Workshop Current Trends Appl. Formal Method: Appl. Formal Methods*, 1999, pp. 1–43.

[30] K. Böttger, R. Schwitter, D. Mollá, and D. Richards, "Towards reconciling use cases via controlled language and graphical models," in *Proc. Appl. Prolog 14th Int. Conf. Web Knowl. Manag. Decis. Support*, 2003, pp. 115–128.

[31] F. Bouquet, C. Grandpierre, B. Legeard, F. Peureux, N. Vacelet, and M. Utting, "A subset of precise UML for model-based testing," in *Proc. 3rd Int. Workshop Adv. Model-Based Testing*, 2007, pp. 95–104.

[32] M. Bozkurt and M. Harman, "Automatically generating realistic test input from web services," in *Proc. IEEE 6th Int. Symp. Serv. Oriented Syst. Eng.*, 2011, pp. 13–24.

[33] L. C. Briand, Y. Labiche, and H. Sun, "Investigating the use of analysis contracts to improve the testability of object-oriented code," *Softw. Pract. Exp.*, vol. 33, no. 7, pp. 637–672, Jun. 2003.

[34] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: Automatically generating inputs of death," *ACM Trans. Inf. Syst. Secur.*, vol. 12, no. 2, pp. 10:1–10:38, Dec. 2008.

[35] J. Callahan, F. Schneider, and S. Easterbrook, "Automated software testing using model-checking," in *Proc. SPIN Workshop*, volume 353, 1996.

[36] F. T. Chan, T. Y. Chen, S. C. Cheung, M. F. Lau, and S. M. Yiu, "Application of metamorphic testing in numerical analysis," in *Proc. IASTED Int. Conf. Softw. Eng.*, 1998, pp. 191–197.

[37] W. K. Chan, S. C. Cheung, and Karl R. P. H. Leung, "*A metamorphic testing approach for online testing of service-oriented software applications,*" in *Software Applications: Concepts, Methodologies, Tools, and Applications*. Hershey, PA, USA: IGI Global, 2009, ch. 7, pp. 2894–2914.

[38] W. K. Chan, S. C. Cheung, and K. R. P. H. Leung, "Towards a metamorphic testing methodology for service-oriented software applications," in *Proc. 5th Int. Conf. Quality Softw.*, Sep. 2005, pp. 470–476.

[39] F. Chen, N. Tillmann, and W. Schulte, "Discovering specifications," Microsoft Research, Tech. Rep. MSR-TR-2005-146, Oct. 2005.

[40] H. Y. Chen, T. H. Tse, F. T. Chan, and T. Y. Chen, "In black and white: An integrated approach to class-level testing of object-oriented programs," *ACM Trans. Softw. Eng. Methodol.*, vol. 7, pp. 250–295, Jul. 1998.

[41] H. Y. Chen, T. H. Tse, and T. Y. Chen, "TACCLE: A methodology for object-oriented software testing at the class and cluster levels," *ACM Trans. Softw. Eng. Methodol.*, vol. 10, no. 1, pp. 56–109, Jan. 2001.

[42] T. Y. Chen, F.-C. Kuo, T. H. Tse, and Z. Q. Zhou, "Metamorphic testing and beyond," in *Proc. Int. Workshop Softw. Technol. Eng. Prac.*, Sep. 2004, pp. 94–100.

[43] T. Chen, D. Huang, H. Huang, T.-H. Tse, Z. Yang, and Z. Zhou, "Metamorphic testing and its applications," in *Proc. 8th Int. Symp. Future Softw. Technol.*, 2004, pp. 310–319.

[44] Y. Cheon, "Abstraction in assertion-based test oracles," in *Proc. 7th Int. Conf. Quality Softw.*, 2007, pp. 410–414.

[45] Y. Cheon and G. T. Leavens, "A simple and practical approach to unit testing: The JML and JUnit way," in *Proc. 16th Eur. Conf. Object-Oriented Programm.*, 2002, pp. 231–255.

[46] L. A. Clarke, "A system to generate test data and symbolically execute programs," *IEEE Trans. Softw. Eng.*, vol. 2, no. 3, pp. 215–222, Sep. 1976.

[47] P. C. Clements, "Managing variability for software product lines: Working with variability mechanisms," in *Proc. 10th Int. Conf. Softw Product Lines*, 2006, pp. 207–208.

[48] M. Clermont and D. Parnas, "Using information about functions in selecting test cases," in *Proc. 1st Int. Workshop Adv. Model-Based Testing*, 2005, pp. 1–7.

[49] D. Coppit and J. M. Haddox-Schatz, "On the use of specification-based assertions as test oracles," in *Proc. 29th Annu. IEEE/NASA Softw. Eng. Workshop*, 2005, pp. 305–314.

[50] M. Davies and E. Weyuker, "Pseudo-oracles for non-testable programs," in *Proc. ACM '81 Conf.*, 1981, pp. 254–257.

[51] L. K. Dillon, "Automated support for testing and debugging of real-time programs using oracles," *SIGSOFT Softw. Eng. Notes*, vol. 25, no. 1, pp. 45–46, Jan. 2000.

[52] R.-K. Doong and P. G. Frankl, "The ASTOOT approach to testing object-oriented programs," *ACM Trans. Softw. Eng. Methodol.*, vol. 3, pp. 101–130, Apr. 1994.

[53] E. Elkind, B. Genest, D. Peled, and H. Qu, "Grey-box checking," in *Proc. 26th IFIP WG 6.1 Int. Conf. Formal Techn. Netw. Distrib. Syst.*, 2006, pp. 420–435.

[54] J. Ellsberger, D. Hogrefe, and A. Sarma, *SDL: Formal Object-Oriented Language for Communicating Systems*. Englewood Cliffs, NJ, USA: Prentice-Hall, 1997.

[55] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The Daikon system for dynamic detection of likely invariants," *Sci. Comput. Programm.*, vol. 69, no. 1, pp. 35–45, 2007.

[56] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *IEEE Trans. Softw. Eng.*, vol. 27, no. 2, pp. 99–123, Feb. 2001.

[57] R. A. Eyre-Todd, "The detection of dangling references in C++ programs," *ACM Lett. Programm. Lang. Syst.*, vol. 2, no. 1–4, pp. 127–134, 1993.

[58] R. Feldt, "Generating diverse software versions with genetic programming: An experimental study," in *Softw., IEE Proc.*, vol. 145, pp. 228–236, Dec. 1998.

[59] X. Feng, D. L. Parnas, T. H. Tse, and T. O'Callaghan, "A comparison of tabular expression-based testing strategies," *IEEE Trans. Softw. Eng.*, vol. 37, no. 5, pp. 616–634, Sep./Oct. 2011.

[60] X. Feng, D. L. Parnas, and T. H. Tse, "Tabular expression-based testing strategies: A comparison," in *Proc. Testing: Acad. Indus. Conf. Prac. Res. Techn.—MUTATION*, 2007, p. 134.

[61] J. Ferrer, F. Chicano, and E. Alba, "Evolutionary algorithms for the multi-objective test data generation problem," *Softw.: Prac. Exp.*, vol. 42, no. 11, pp. 1331–1362, 2011.

[62] J. S. Fitzgerald and P. G. Larsen, "*Modelling Systems—Practical Tools and Techniques in Software Development*, 2nd ed. Cambridge, U.K.: Cambridge Univ. Press, 2009.

[63] C. Flanagan and K. R. M. Leino, "Houdini, an annotation assistant for ESC/Java," *Lecture Notes in Comput. Sci.*, vol. 2021, pp. 500–517, 2001.

[64] R. W. Floyd, "Assigning meanings to programs," in *Mathematical Aspects of Computer Science*, volume 19 of *Symposia in Applied Mathematics*, J. T. Schwartz Ed., American Providence, RI, USA: Mathematical Society, 1967, pp. 19–32. .

[65] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Trans. Softw. Eng.*, vol. 39, no. 2, pp. 276–291, Feb. 2013.

[66] G. Fraser and A. Zeller, "Exploiting common object usage in test case generation," in *Proc. 4th IEEE Int. Conf. Softw. Testing, Verification Validation*, 2011, pp. 80–89.

[67] K. Frounchi, L. C. Briand, L. Grady, Y. Labiche, and R. Subramanyan, "Automating image segmentation verification and validation by learning test oracles," *Inf. Softw. Technol.*, vol. 53, no. 12, pp. 1337–1348, 2011.

[68] P. Gall and A. Arnould, "Formal specifications and test: Correctness and oracle," in *Recent Trends in Data Type Specification*, M. Haveraaen, O. Owe, and O.-J. Dahl, Ed. Berlin, Germany, Springer, 1996, pp. 342–358.

[69] J. Gannon, P. McMullin, and R. Hamlet, "Data abstraction, implementation, specification, and testing," *ACM Trans. Programm. Lang. Syst*, vol. 3, no. 3, pp. 211–223, 1981.

[70] A. Gargantini and E. Riccobene, "ASM-based testing: Coverage criteria and automatic test sequence," *J. Universal Comput. Sci.*, vol. 7, no. 11, pp. 1050–1067, Nov. 2001.

[71] S. J. Garland, J. V. Guttag, and J. J. Horning, "An overview of Larch," in *Functional Programming, Concurrency, Simulation and Automated Reasoning*, Springer Berlin Heidelberg, 1993, pp. 329–348.

[72] M.-C. Gaudel, "Testing from formal specifications, a generic approach," in *Proc. 6th Ade-Eur. Int. Conf. Leuven Rel. Softw. Technol.*, 2001, pp. 35–48.

[73] M.-C. Gaudel and P. R. James, "Testing algebraic data types and processes: A unifying theory," *Formal Asp. Comput.*, vol. 10, no. 5–6, pp. 436–451, 1998.

[74] G. Gay, S. Rayadurgam, and M. Heimdah, "Improving the accuracy of oracle verdicts through automated model steering," in *Proc. 29th ACM/IEEE Int. Conf. Automated Softw. Eng.*, 2014, pp. 527–538.

[75] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *Proc. ACM SIGPLAN Conf. Programm. Lang. Des. Implementation*, 2005, pp. 213–223.

[76] A. Groce, M. A. Alipour, C. Zhang, Y. Chen, and J. Regehr, "Cause reduction for quick testing," in *Proc. IEEE Int. Conf. Softw. Testing, Verification, Validation*, 2014, pp. 243–252.

[77] A. Groce, D. Peled, and M. Yannakakis, "AMC: An adaptive model checker," in *Proc. Int. Conf. Comput. Aided Verification*, 2002, pp. 521–525.

[78] R. Groz, K. Li, A. Petrenko, and M. Shahbaz, "Modular system verification by inference, testing and reachability analysis," in *Proc. TestCom/FATES*, 2008, pp. 216–233.

[79] Z. Gu, E. T. Barr, D. J. Hamilton, and Z. Su, "Has the bug really been fixed?" in *Proc. Int. Conf. Softw. Eng.*, 2010, pp. 55–64.

[80] R. Guderlei and J. Mayer, "Statistical metamorphic testing testing programs with random output by means of statistical hypothesis tests and metamorphic testing," in *7th Int. Conf. Quality Softw.*, Oct. 2007, pp. 404–409.

[81] D. Harel, "Statecharts: A visual formalism for complex systems," *Sci. Comput. Programm.*, vol. 8, no. 3, pp. 231–274, 1987.

[82] M. Harman, S. G. Kim, K. Lakhotia, P. McMinn, and S. Yoo, "Optimizing for the number of tests generated in search based test data generation with an application to the oracle cost problem," in *Proc. Int. Workshop Search-Based Softw. Testing*, Apr. 2010, pp. 182–191.

[83] M. Harman, A. Mansouri, and Y. Zhang, "Search based software engineering: A comprehensive analysis and review of trends techniques and applications," Dept. of Comput. Sci., King's College London, London, U.K., Tech. Rep. TR-09-03, Apr. 2009.

[84] M. Harman, P. McMinn, J. T. de Souza, and S. Yoo, "Search based software engineering: Techniques, taxonomy, tutorial," in *Empirical Software Engeneering and Verification*, B. Meyer and M. Nordio, Eds. New York, NY, USA: Springer, 2012 pp. 1–59.

[85] M. J. Harrold, R. Gupta, and M. L. Soffa, "A methodology for controlling the size of a test suite," *ACM Trans. Softw. Eng. Methodol.*, vol. 2, no. 3, pp. 270–285, Jul. 1993.

[86] M. J. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi, "An empirical investigation of the relationship between spectra differences and regression faults," *Softw. Testing, Verification Reliability*, vol. 10, no. 3, pp. 171–194, 2000.

[87] D. L. Heine and M. S. Lam, "A practical flow-sensitive and context-sensitive C and C++ memory leak detector," in *Proc. ACM SIGPLAN Conf. Programm. Lang. Des. Implementation*, 2003, pp. 168–181.

[88] J. Henkel and A. Diwan, "Discovering algebraic specifications from Java classes," *Lecture Notes in Comput. Sci.*, vol. 2743, pp. 431–456, 2003.

[89] F. C. Hennie, *Finite-State Models for Logical Machines*. Hoboken, NJ, USA: Wiley, 1968.

[90] M. H. Heule and S. Verwer, "Software model synthesis using satisfiability solvers," *Empirical Softw. Eng.*, vol. 18, pp. 825–856, Aug. 2013.

[91] R. M. Hierons, "Oracles for distributed testing," *IEEE Trans. Softw. Eng.*, vol. 38, no. 3, pp. 629–641, May/Jun. 2012.

[92] R. M. Hierons, "Verdict functions in testing with a fault domain or test hypotheses," *ACM Trans. Softw. Eng. Methodol.*, vol. 18, no. 4, pp. 14:1–14:19, Jul. 2009.

[93] R. M. Hierons, K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Lüttgen, A. J. H. Simons, S. Vilkomir, M. R. Woodward, and H. Zedan, "Using formal specifications to support testing," *ACM Comput. Surv.*, vol. 41, pp. 9:1–9:76, Feb. 2009.

[94] C. A. R. Hoare, "An axiomatic basis of computer programming," *Commun. ACM*, vol. 12, pp. 576–580, 1969.

[95] M. Holcombe, "X-machines as a basis for dynamic system specification," *Softw. Eng. J.*, vol. 3, no. 2, pp. 69–76, 1988.

[96] M. Holcombe and F. Ipate, "Correct systems: Building a business process solution," *Softw. Testing VerificationRel.*, vol. 9, no. 1, pp. 76–77, 1999.

[97] G. J. Holzmann, "The model checker SPIN," *IEEE Trans. Softw. Eng.*, vol. 23, no. 5, pp. 279–295, May 1997.

[98] W. E. Howden, "A functional approach to program testing and analysis," *IEEE Trans. Softw. Eng.*, vol. 12, no. 10, pp. 997–1005, Oct. 1986.

[99] W. E. Howden, "Theoretical and empirical studies of program testing," *IEEE Trans. Softw. Eng.*, vol. 4, no. 4, pp. 293–298, Jul. 1978.

[100] M. Hughes and D. Stotts, "Daistish: Systematic algebraic testing for OO programs in the presence of side-effects," *SIGSOFT Softw. Eng. Notes*, vol. 21, no. 3, pp. 53–61, May 1996.

[101] D. Jackson, "*Software Abstractions: Logic, Language, and Analysis*. Cambridge, MA, USA: MIT Press, 2006.

[102] C. Jard and G. V. Bochmann, "An approach to testing specifications," *J. Syst. Softw.*, vol. 3, no. 4, pp. 315–323, 1983.

[103] D. Jeffrey and R. Gupta, "Test case prioritization using relevant slices," in *Proc. 30th Annu. Int. Comput. Softw. Appl. Conf.*, 2006, pp. 411–420, 2006.

[104] Y. Jin and D. L. Parnas, "Defining the meaning of tabular mathematical expressions," *Sci. Comput. Program.*, vol. 75, no. 11, pp. 980–1000, Nov. 2010.

[105] M. J. Kearns and U. V. Vazirani, *An Introduction to Computational Learning Theory*. Cambridge, MA, USA: MIT Press, 1994.

[106] R. M. Keller, "Formal verification of parallel programs," *Commun. ACM*, vol. 19, no. 7, pp. 371–384, 1976.

[107] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976.

[108] K. Lakhotia, P. McMinn, and M. Harman, "An empirical investigation into branch coverage for C programs using CUTE and AUSTIN," *J. Syst. Softw.*, vol. 83, no. 12, pp. 2379–2391, 2010.

[109] A. van Lamsweerde, "Formal specification: A roadmap," in *Proc. Conf. Future Softw. Eng.*, 2000, pp. 147–159.

[110] K. Lano and H. Haughton, "*Specification in B: An Introduction Using the B Toolkit*. London, U.K.: Imperial College Press, 1996.

[111] D. Lee and M. Yannakakis, "Principles and methods of testing finite state machines—a survey," *Proc. IEEE*, vol. 84, no. 8, pp. 1090–1123, Aug. 1996.

[112] A. Leitner, M. Oriol, A Zeller, I. Ciupa, and B. Meyer, "Efficient unit test case minimization," in *Proc. Autom. Softw. Eng.*, 2007, pp. 417–420.

[113] K. Li, R. Groz, and M. Shahbaz, "Integration testing of components guided by incremental state machine learning," in *Proc. Testing: Acad. Ind. Conf. Prac. Res. Tech.*, 2006, pp. 59–70.

[114] D. Lorenzoli, L. Mariani, and M. Pezzè, "Automatic generation of software behavioral models," in *Proc. 30th Int. Conf. Softw. Eng.*, 2008, pp. 501–510.

[115] P. Loyola, M. Staats, I.-Y. Ko, and G. Rothermel, "Dodona: Automated oracle data set selection," in *Proc. Int. Symp. Softw. Testing Anal.*, 2014, pp. 193–203.

[116] D. Luckham and F. W. Henke, "An overview of ANNA-a specification language for ADA," *IEEE Soft.*, vol. 2, no. 2, pp. 9–22, 1984.

[117] N. A. Lynch and M. R. Tuttle, "An introduction to input/output automata," *CWI Quarterly*, vol. 2, pp. 219–246, 1989.

[118] P. D. L. Machado, "On oracles for interpreting test results against algebraic specifications," in *Proc. Int. Conf. Algebraic Methodol. Softw. Technol.*, 1999, pp. 502–518.

[119] P. Maker.(1998). GNU Nana: Improved support for assertion checking and logging in GNU C/C++ [Online]. Available: http://gnu.cs.pu.edu.tw/software/nana/

[120] H. Malik, H. Hemmati, and A. E. Hassan, "Automatic detection of performance deviations in the load testing of large scale systems," in *Proc. Int. Conf. Softw. Eng.—Softw. Eng. Prac. Track*, 2013, pp. 1012–1021.

[121] L. Mariani, F. Pastore, and M. Pezzè, "Dynamic analysis for diagnosing integration faults," *IEEE Trans. Softw. Eng.*, vol. 37, no. 4, pp. 486–508, Jul./Aug. 2011.

[122] B. Marre, "Loft: A tool for assisting selection of test data sets from algebraic specifications," in *Proc. 6th Int. Joint Conf. CAAP/FASE Theory Pract. Softw. Develop.*, 1995, pp. 799–800.

[123] A. P. Mathur, "Performance, effectiveness, and reliability issues in software testing," in *Proc. 15th Annu. Int. Comput. Softw. Appl. Conf.*, 1991, pp. 604–605.

[124] J. Mayer and R. Guderlei, "Test oracles using statistical methods," in *Proc. 1st Int. Workshop Softw. Quality, Lecture Notes Inform. P-58, Kllen Druck+Verlag GmbH*, 2004, pp. 179–189.

[125] P. McMinn, M. Stevenson, and M. Harman, "Reducing qualitative human oracle costs associated with automatically generated test data," in *Proc. 1st Int. Workshop Softw. Test Output Validation*, 2010, pp. 1–4.

[126] P. McMinn, "Search-based software test data generation: A survey," *Softw. Test. Verif. Rel.*, vol. 14, no. 2, pp. 105–156, Jun. 2004.

[127] P. McMinn, "Search-based failure discovery using testability transformations to generate pseudo-oracles," in *Proc. Genetic Evol. Comput. Conf.*, 2009, pp. 1689–1696.

[128] P. McMinn, "Search-based software testing: Past, present and future," in *Proc. Int. Workshop Search-Based Softw. Testing*, 2011, pp. 153–163.

[129] P. McMinn, M. Shahbaz, and M. Stevenson, "Search-based test input generation for string data types using the results of web queries," in *Proc. IEEE 5th Int. Conf. Softw. Testing, Verification Validation*, 2012, pp. 141–150.

[130] P. McMinn, M. Stevenson, and M. Harman, "Reducing qualitative human oracle costs associated with automatically generated test data," in *Proc. Int. Workshop Softw. Test Output Validation*, Jul. 2010, pp. 1–4.

[131] A. M. Memon, "Automatically repairing event sequence-based GUI test suites for regression testing," *ACM Trans. Softw. Eng. Methodol.*, vol. 18, no. 2, pp. 1–36, 2008.

[132] A. M. Memon, M. E. Pollack, and M. L. Soffa, "Automated test oracles for GUIs," in *Proc. 8th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2000, pp. 30–39.

[133] A. M. Memon and Q. Xie, "Using transient/persistent errors to develop automated test oracles for event-driven software," in *Proc. 19th IEEE Int. Conf. Automated Softw. Eng.*, 2004, pp. 186–195.

[134] M. Merten, F. Howar, B. Steffen, P. Pellicione, and M. Tivoli, "Automated inference of models for black box systems based on interface descriptions," in *Proc. 5th Int. Conf. Leveraging Appl. Formal Methods, Verification Validation: Technol. Mastering Change—Volume Part I*, 2012, pp. 79–96.

[135] B. Meyer, "Eiffel: A language and environment for software engineering," *J. Syst. Softw.*, vol. 8, no. 3, pp. 199–246, Jun. 1988.

[136] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Commun. ACM*, vol. 33, no. 12, pp. 32–44, 1990.

[137] S. Mouchawrab, L. C. Briand, Y. Labiche, and M. Di Penta, "Assessing, comparing, and combining state machine-based testing and structural testing: A series of experiments," *IEEE Trans. Softw. Eng.*, vol. 37, no. 2, pp. 161–187, Mar./Apr. 2011.

[138] C. Murphy, K. Shen, and G. Kaiser, "Automatic system testing of programs without test oracles," in *Proc. 18th Int. Symp. Softw. Testing Anal.*, 2009, pp. 189–200.

[139] C. Murphy, K. Shen, and G. Kaiser, "Using JML runtime assertion checking to automate metamorphic testing in applications without test oracles," in *Proc. Int. Conf. Softw. Testing Verification and Validation*, 2009, pp. 436–445.

[140] A. J. Offutt, J. Pan, and J. M. Voas, "Procedures for reducing the size of coverage-based test sets," in *Int. Conf. Testing Comput. Softw.*, 1995, pp. 111–123.

[141] C. Pacheco and M. Ernst, "Eclat: Automatic generation and classification of test inputs," in *Proc. 19th Eur. Conf. Object-Oriented Programming*, pp. 734–734, 2005.

[142] D. L. Parnas, J. Madey, and M. Iglewski, "Precise documentation of well-structured programs," *IEEE Trans. Softw. Eng.*, vol. 20, no. 12, pp. 948–976, Dec. 1994.

[143] D. L. Parnas, "Document based rational software development," *J. Knowl. Based Syst.*, vol. 22, pp. 132–141, Apr. 2009.

[144] D. L. Parnas, "Precise documentation: The key to better software," in *The Future of Software Engineering*, S. Nanz, Ed. Berlin, Germany, Springer, 2011, pp. 125–148.

[145] D. L. Parnas and J. Madey, "Functional documents for computer systems," *Sci. Comput. Program.*, vol. 25, no. 1, pp. 41–61, Oct. 1995.

[146] F. Pastore, L. Mariani, and G. Fraser, "Crowdoracles: Can the crowd solve the oracle problem?" in *Proc. Int. Conf. Softw. Testing, Verification Validation*, 2013, pp. 342–351.

[147] D. Peled, M. Y. Vardi, and M. Yannakakis, "Black box checking," *J. Automata, Lang. Combinatorics*, vol 7, no. 2, pp. 225–246, 2002.

[148] D. K. Peters and D. L. Parnas, "Using test oracles generated from program documentation," *IEEE Trans. Softw. Eng.*, vol. 24, no. 3, pp. 161–173, Mar. 1998.

[149] D. K. Peters and D. L. Parnas, "Requirements-based monitors for real-time systems," *IEEE Trans. Softw. Eng.*, vol. 28, no. 2, pp. 146–158, Feb. 2002.

[150] M. Pezzè, and C. Zhang, "Automated test oracles: A survey," in A. Hurson and A. Memon, Eds., *Advances in Computers*, vol. 95, pp. 1–48. Elsevier Ltd., 2014.

[151] N. Polikarpova, I. Ciupa, and B. Meyer, "A comparative study of programmer-written and automatically inferred contracts," in *Proc. 18th Int. Symp. Softw. Testing Anal.*, 2009, pp. 93–104.

[152] S. J. Prowell and J. H. Poore, "Foundations of sequence-based software specification," *IEEE Trans, Softw. Eng.*, vol. 29, no. 5, pp. 417–429, May 2003.

[153] S. Ratcliff, D. R. White, and J. A. Clark, "Searching for invariants using genetic programming and mutation testing," in *Proc. 13th Annu. Conf. Genetic Evol. Comput.*, 2011, pp. 1907–1914.

[154] F. Ricca and P. Tonella, "Detecting anomaly and failure in web applications," *IEEE Multimedia*, vol. 13, no. 2, pp. 44–51, Apr.-Jun. 2006.

[155] D. S. Rosenblum, "A practical approach to programming with assertions," *IEEE Trans., Softw. Eng.*, vol. 21, no. 1, pp. 19–31, Jan. 1995.

[156] G. Rothermel, M. J. Harrold, J. von Ronne, and C. Hong, "Empirical studies of test-suite reduction," *Softw. Testing, Verification Rel.*, vol. 12, pp. 219–249, 2002.

[157] N. Walkinshaw S. Ali, K. Bogdanov. (2007, Oct.). A comparative study of methods for dynamic reverse-engineering of state models. The Univ. of Sheffield, Dept. of Comput. Sci., Sheffield, U.K., Tech. Rep. CS-07-16, Oct.[Online]. Available: http://www.dcs.shef.ac.uk/intranet/research/resmes/CS0716.pdf

[158] D. Schuler and A. Zeller, "Assessing oracle quality with checked coverage," in *Proc. 4th IEEE Int. Conf. Softw. Testing, Verification Validation*, 2011, pp. 90–99.

[159] R. Schwitter, "English as a formal specification language," in *Proc. 13th Int. Workshop Database Expert Syst. Appl.*, Sep. 2002, pp. 228–232.

[160] S. Segura, R. M. Hierons, D. Benavides, and A. Ruiz-Cortés, "Automated metamorphic testing on the analyses of feature models," *Inf. Softw. Technol.*, vol. 53, no. 3, pp. 245–258, 2011.

[161] K. Sen, D. Marinov, and G. Agha, "CUTE: A concolic unit testing engine for C," in *Proc. 10th Eur. Softw. Eng. Conf. Held Jointly 13th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2005, pp. 263–272.

[162] S. Shahamiri, W. Wan-Kadir, S. Ibrahim, and S. Hashim, "Artificial neural networks as multi-networks automated test oracle," *Autom. Softw. Eng.*, vol. 19, no. 3, pp. 303–334, 2012.

[163] S. R. Shahamiri, W. M. N. Wan-Kadir, S. Ibrahim, and S. Z. M. Hashim, "An automated framework for software test oracle," *Inf. Softw. Technol.*, vol. 53, no. 7, pp. 774–788, 2011.

[164] S. R. Shahamiri, W. M. N. Wan-Kadir, and S. Z. M. Hashim, "A comparative study on automated software test oracle methods," in *Proc. 4th Int. Conf. Softw. Eng. Adv.*, Porto, 2009, pp. 140–145.

[165] M. Shahbaz, *Reverse Engineering and Testing of Black-Box Software Components*. LAP Lambert Academic Publishing, 2012.

[166] M. Shahbaz, P. McMinn, and M. Stevenson, "Automatic generation of valid and invalid test data for string validation routines," *Sci. Comput. Programm.*, vol. 97, no. 4, pp. 405–425, 2015.

[167] K. Shrestha and M. J. Rutherford, "An empirical evaluation of assertions as oracles," in *Proc. 4th IEEE Int. Conf. Softw. Testing, Verification Validation*, 2011, pp. 110–119.

[168] G. Shu, Y. Hsu, and D. Lee, "Detecting communication protocol security flaws by formal fuzz testing and machine learning," in *Proc. 28th IFIP WG 6.1 Int. Conf. Formal Techn. Networked Distrib. Syst.*, 2008, pp. 299–304.

[169] J. Sifakis, "Deadlocks and livelocks in transition systems," in *Proc. 9th Symp. Math. Found. Comput. Sci.*, 1980, pp. 587–600.

[170] A. J. H. Simons, "JWalk: A tool for lazy systematic testing of Java classes by design introspection and user interaction," *Autom. Softw. Eng.*, vol. 14, no. 4, pp. 369–418, Dec. 2007.

[171] R. Singh, D. Giannakopoulou, and C. S. Pasareanu, "Learning component interfaces with may and must abstractions," in *Proc. 22nd Int. Conf. Comput. Aided Verification*, 2010, pp. 527–542.

[172] J. M. Spivey, *Z Notation—a Reference Manual*, 2nd ed. Englewood Cliffs, NJ, USA: Prentice-Hall, 1992.

[173] M. Staats, G. Gay, and M. P. E. Heimdahl, "Automated oracle creation support, or: How I learned to stop worrying about fault propagation and love mutation testing," in *Proc. 34th Int. Conf. Softw. Eng.*, 2012, pp. 870–880.

[174] M. Staats, M. W. Whalen, and M. P. E. Heimdahl, "Programs, tests, and oracles: The foundations of testing revisited," in *Proc. 33rd Int. Conf. Softw. Eng.*, 2011, pp. 391–400.

[175] M. Staats, S. Hong, M. Kim, and G. Rothermel, "Understanding user understanding: Determining correctness of generated program invariants," in *Proc. Int. Symp. Softw. Testing Anal.*, 2012, pp. 188–198.

[176] M. Staats, P. Loyola, and G. Rothermel, "Oracle-centric test case prioritization," in *Proc. 23rd Int. Symp. Softw. Rel. Eng.*, 2012, pp. 311–320.

[177] A. Takanen, J. DeMott, and C. Miller, *Fuzzing for Software Security Testing and Quality Assurance*, 1st ed. Norwood, MA, USA: Artech House, Inc., 2008.

[178] R. Taylor, M. Hall, K. Bogdanov, and J. Derrick, "Using behaviour inference to optimise regression test sets," in *Proc. 24th IFIP WG 601 Int. Conf. Testing Softw. Syst.*, 2012, pp. 184–199.

[179] A. Tiwari. (2002). Formal semantics and analysis methods for Simulink Stateflow models. SRI International, Menlo Park, CA, USA: Tech. Rep. [Online]. Available: http://www.csl.sri.com/users/tiwari/html/stateflow.html

[180] J. Tretmans, "Test generation with inputs, outputs and repetitive quiescence," *Softw.—Concepts Tools*, vol. 17, no. 3, pp. 103–120, 1996.

[181] A. M. Turing, "Checking a large routine," in *Report of a Conference on High Speed Automatic Calculating Machines*. Cambridge, England, University Mathematical Laboratory, Jun. 1949, pp. 67–69.

[182] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*. San Francisco, CA, USA: Morgan Kaufmann, 2007.

[183] M. Utting, A. Pretschner, and B. Legeard, "A taxonomy of model-based testing approaches," *Softw. Test. Verif. Reliab.*, vol. 22, no. 5, pp. 297–312, Aug. 2012.

[184] G. V. Bochmann, C. He, and D. Ouimet, "Protocol testing using automatic trace analysis," in *Proc. Can. Conf. Elect. Comput. Eng.*, 1989, pp. 814–820.

[185] A. Van Lamsweerde and M. Sintzoff, "Formal derivation of strongly correct concurrent programs," *Acta Informatica*, vol. 12, no. 1, pp. 1–31, 1979.

[186] J. M. Voas, "PIE: A dynamic failure-based technique," *IEEE Trans. Softw. Eng.*, vol. 18, no. 8, pp. 717–727, Aug. 1992.

[187] N. Walkinshaw, K. Bogdanov, J. Derrick, and J. Paris, "Increasing functional coverage by inductive testing: A case study," in *Proc. 22nd IFIP WG 6.1 Int. Conf. Testing Softw. Syst.*, 2010, pp. 126–141.

[188] N. Walkinshaw and K. Bogdanov, "Inferring finite-state models with temporal constraints," in *Proc. 23rd IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2008, pp. 248–257.

[189] N. Walkinshaw, J. Derrick, and Q. Guo, "Iterative refinement of reverse-engineered models by model-based testing," in *Proc. 2nd World Congress Formal Methods*, 2009, pp. 305–320.

[190] Y. Wang and D. Parnas, "Trace rewriting systems," in *Conditional Term Rewriting Systems*, vol. 656, M. Rusinowitch and J.-L. Rémy Eds. Berlin, Germany, Springer, 1993, pp. 343–356.

[191] Y. Wang and D. L. Parnas, "Simulating the behaviour of software modules by trace rewriting," in *Proc. 15th Int. Conf. Softw. Eng.*, 1993, pp. 14–23.

[192] Y. Wei, C. A. Furia, N. Kazmin, and B. Meyer, "Inferring better contracts," in *Proc. 33rd Int. Conf. Softw. Eng.*, 2011, pp. 191–200.

[193] Y. Wei, H. Roth, C. A. Furia, Y. Pei, A. Horton, M. Steindorfer, M. Nordio, and B. Meyer, "Stateful testing: Finding more errors in code and contracts," in *Proc. 26th IEEE/ACM Int. conf. Autom. Softw. Eng.*, 2011, pp. 440–443.

[194] E. J. Weyuker, "Assessing test data adequacy through program inference," *ACM Trans. Programm. Lang. Syst.*, vol. 5, no. 4, pp. 641–655, 1983.

[195] E. J. Weyuker and F. I. Vokolos, "Experience with performance testing of software systems: Issues, an approach, and case study," *IEEE Trans. Softw. Eng.*, vol. 26, no. 12, pp. 1147–1156, Dec. 2000.

[196] E. J. Weyuker, "On testing non-testable programs," *Comput. J.*, vol. 25, no. 4, pp. 465–470, Nov. 1982.

[197] J. M. Wing, "A specifier's introduction to formal methods," *IEEE Comput.*, vol. 23, no. 9, pp. 8–24, Sep. 1990.

[198] Q. Xie and A. M. Memon, "Designing and comparing automated test oracles for GUI-based software applications," *ACM Trans. Softw. Eng. Methodol.*, vol. 16, no. 1, p. 4, 2007.

[199] T. Xie, "Augmenting automatically generated unit-test suites with regression oracle checking," in *Proc. 20th Eur. Conf. Object-Oriented Programm.*, Jul. 2006, pp. 380–403.

[200] T. Xie and D. Notkin, "Checking inside the black box: Regression testing by comparing value spectra," *IEEE Trans. Softw. Eng.*, vol. 31, no. 10, pp. 869–883, Oct. 2005.

[201] Y. Xie and A. Aiken, "Context-and path-sensitive memory leak detection," *ACM SIGSOFT Softw. Eng. Notes*, vol. 30, no. 5, pp. 115–125, 2005.

[202] Z. Xu, Y. Kim, M. Kim, G. Rothermel, and M. B. Cohen, "Directed test suite augmentation: Techniques and tradeoffs," in *Proc. 18th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2010, pp. 257–266.

[203] S. Yoo, "Metamorphic testing of stochastic optimisation," in *Proc. 3rd Int. Conf. Softw. Testing, Verification, Validation Workshops*, 2010, pp. 192–201.

[204] S. Yoo and M. Harman, "Regression testing minimisation, selection and prioritisation: A survey," *Softw. Testing, Verification, Rel.*, vol. 22, no. 2, pp. 67–120, Mar. 2012.

[205] B. Yu, L. Kong, Y. Zhang, and H. Zhu, "Testing Java components based on algebraic specifications," in *Proc. Int. Conf. Softw. Testing, Verification, Validation*, 2008, pp. 190–199.

[206] A. Zeller and R Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Trans. Softw. Eng.*, vol. 28, no. 2, pp. 183–200, Feb. 2002.

[207] S. Zhang, D. Saff, Y. Bu, and M. D. Ernst, "Combined static and dynamic automated test generation," in *Proc. Int. Symp. Softw. Testing Anal.*, 2011, vol. 11, pp. 353–363.

[208] W. Zheng, H. Ma, M. R. Lyu, T. Xie, and I. King, "Mining test oracles of web search engines," in *Proc. 26th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2011, pp. 408–411.

[209] Z. Q. Zhou, S. Zhang, M. Hagenbuchner, T. H. Tse, F.-C. Kuo, and T. Y. Chen, "Automated functional testing of online search services," *Softw. Testing, Verification Rel.*, vol. 22, no. 4, pp. 221–243, 2012.

[210] H. Zhu, "A note on test oracles and semantics of algebraic specifications," in *Proc. 3rd Int. Conf. Q. Softw.*, 2003, pp. 91–98.

[211] B. Zorn and P. Hilfinger, "A memory allocation profiler for C and Lisp programs," in *Proc. Summer USENIX Conf.*, 1988, pp. 223–237.

Earl Barr received the PhD degree in computer science in 2009 from the University of California at Davis. He is a lecturer at the University College London in its Systems and Software Engineering Research Group and a member of its CREST centre. His research interests include testing and program analysis, empirical software engineering, and cyber-security.

Mark Harman is a professor of software engineering in the Department of Computer Science at University College London, where he directs the CREST centre and is the head of Software Systems Engineering. He is widely known for work on source code analysis (particularly program slicing) and testing and cofounded the field of search-based software engineering (SBSE). SBSE research has rapidly grown over the past five years and now includes over 1,600 authors, from nearly 300 institutions spread over more than 40 countries. A recent tutorial paper on SBSE can be found here: http://www.cs.ucl.ac.uk/staff/mharman/laser.pdf

Phil McMinn is a senior lecturer in the Department of Computer Science at the University of Sheffield, where he has been researching and teaching software engineering since 2006. He is an assistant director at the University's Advanced Computing Research Centre, where he leads the Software Quality team. His main research interests include search-based software engineering, software testing, program transformation, and reverse engineering.

Muzammil Shahbaz received the PhD degree in software engineering from the Grenoble Institute of Technology, France. He is senior engineer at Ultra Electronics based in United Kingdom. He is working on developing synergies between formal methods and software testing in the aviation industry. He was a researcher at Orange Labs, France, Fraunhofer Institute of Experimental Software Engineering, Germany and University of Sheffield, United Kingdom.

Shin Yoo received the PhD degree in computer science from King's College London, United Kingdom. He is a lecturer (assistant professor) of software engineering at Centre for Research on Evolution, Search, and Testing at University College London, United Kingdom. His research interests include software testing and debugging, evolutionary computation, program slicing, and the use of Information Theory in software engineering.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.