A Comprehensive Survey of Trends in Oracles for Software Testing

Mark Harman, Phil McMinn, Muzammil Shahbaz and Shin Yoo

Abstract—Testing involves examining the behaviour of a system in order to discover potential faults. Determining the desired correct behaviour for a given input is called the "oracle problem". Oracle automation is important to remove a current bottleneck which inhibits greater overall test automation; without oracle automation, the human has to determine whether observed behaviour is correct. The literature on 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 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 oracle, even the humble human, involve challenges of reducing cost and increasing benefit. This paper provides a comprehensive survey of current approaches to the 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 testing seeks to automate as much of the test process as practical and desirable, in order to make testing faster, cheaper and more reliable. In order to automate testing, we need an oracle, a procedure that determines what the correct behaviour of a system should be for all input stimuli with which we wish to subject the system under test. There has been much recent progress on the problem of automating the generation of test inputs. However, the problem of automating the oracle remains a topic of much current study; a research agenda that involves multiple diverse avenues of investigation.

Even the problem of automatically generating test inputs is hard. In general 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. However, even the simplest forms of testing involve an attempt to approximate a solution to an undecidable problem. For example, attempting to cover all branches [6], [10], [16], [64], [77], [159], [177] in a system under test can never be complete because branch reachability is known to be undecidable in general [192].

The problem of generating test inputs to achieve coverage of test adequacy criterion has been the subject of research interest for nearly four decades [48], [106] and has been the subject of much recent development in research and practice with significant advances in Search-Based Testing [2], [4], [83], [125], [127] and Dynamic Symbolic Execution [77], [108], [159], both of which automate the generation of test inputs, but do not address the oracle problem.

We therefore find ourselves in a position where the automated generation of test inputs is increasingly being addressed, while the automated checking that these inputs lead to the desired outputs remains less well explored and comparatively less well solved. This current open problem represents a significant bottleneck that inhibits greater test automation and wider uptake of automated testing methods and tools.

Of course, one might hope that the software under test has been developed with respect to 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 [134]. In these situations, the oracle cost problem is ameliorated by the presence of an automatable oracle to which a testing tool can refer to check outputs, free from the need for costly human intervention.

Where there is no full specification of the properties of the system under test, one may hope that it is possible to construct some form of partial oracle that is able to answer oracle questions for some inputs, relying on alternative means to answer the oracle question for others. Such partial oracles can be constructed using metamorphic testing (based on known relationships between desired behaviour) or by deriving oracle information from executions or documentation.

However, for many systems and much of testing as currently practiced in industry, the tester has the luxury of neither formal specification nor assertions, nor even automated partial oracles [90], [91]. The tester must therefore face the potentially daunting task of manually checking the system's behaviour for all test cases generated. In such cases, it is essential that automated software testing approaches address the human oracle cost problem [1], [82], [129].

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

There have been four previous partial surveys of topics relating to test oracles. However, none has provided a comprehensive survey of trends and results. In 2001, Baresi and Young [18] 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 oracles (covered in the present paper). Another early work was the initial motivation for considering the oracle problem contained in Binder's textbook on software testing [24], published in 2000. More recently, in 2009, Shahamiri et al. [162] compared six techniques from the specific category of derived oracles. Finally most recently, in 2011, Staats et al. [171] proposed a theoretical analysis that included test oracles in a revisitation of the fundamentals of testing.

Despite these works, research into the 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 of research in this important area of software testing by providing the first comprehensive analysis and review of work on the oracle problem.

The rest of the paper is organised as follows: Section 2 sets out the definitions relating to oracles that allow us to compare and contrast the techniques in the literature on a more formal basis. Section 3 analyses some of the growth trends in the area and presents a timeline showing the primary development in the field. Sections 4, 5 and 6 cover the survey of the literature in three broad categories:

- where there are **specified** oracles (Section 4;
- where oracles have to be **derived** (Section 5) and
- where some form of oracle information is **implicit** (Section 6).

These three sections cover the situations in which some form of oracle information is available for use in testing, while Section 7 covers techniques designed to cater for situations in which no oracle is available, or for which testing must cover some aspect for which no oracle information is available. This last case is the most prevalent in practice, since oracles seldom cover the *entire* behaviour of the system. Section 8 presents a road map for future research on software test oracles, based on our analysis of the literature and Section 9 concludes.

2 Definitions

This section sets out some foundational definitions to establish a common 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.

A good formalisation should be simple, general and revealing. It should be simple so that it can be easily used. However, though simple, it should be sufficiently general to capture all existing instances of the phenomenon to be studied. It should also be revealing in the sense that the theory should be able to reveal new and interesting possibilities, not obvious without the theoretical foundations and the analysis they support.

We believe our theoretical foundations are conceptually very simple but, as we shall show in the remainder of the paper, sufficiently general to capture all existing forms of oracle in the literature. We also will use the theory to highlight hitherto unrecognised connections between metamorphic and regression testing in Section 8.

Testing is a permutation of two types of test activities, *stimuli* and *observations*, which are supersets of the traditional notion of test input and output respectively. We define stimuli and observations as coming from arbitrary infinite sets, which form a kind of alphabet for the System Under Test (SUT). This definition of alphabets is merely intended to make concrete the set of external stimuli that could be applied to the SUT, and the set of behaviours that we can observe.

Definition 1 (Alphabets and Activities): A stimulus to the System Under Test will be considered to be drawn from an alphabet, $\mathbb{S}$, while an observation will be drawn from an alphabet, $\mathbb{O}$, disjoint from $\mathbb{S}$. The union of all possible stimuli and observations form the test activity alphabet, $\mathbb{A}$.

We use the terms 'stimulus' and 'observation' in the broadest sense possible to cater for various testing scenarios, functional and non-functional. A stimulus can be either an explicit test input from the tester or an environmental factor that can affect the testing. Similarly, an observation can range from an automated, functional test oracle to a non-functional execution profile.

For example, stimuli can include the configuration and platform settings, database table contents, device states, typed values at an input device, inputs on a channel from another system, sensor inputs and so on. Observations can 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 and any other measurable attributes of the execution of the SUT.

Stimuli and observations are members of different sets of test activities, but we combine them in a single test activity sequence. As a notational convenience in such sequences, we shall write $\overline{x}$ when x is a member of the set of observations (such as outputs) and $\underline{x}$ when x is a member of the set of stimuli (such as inputs).

A **test oracle** is a predicate that determines whether a given test activity sequence is an acceptable behaviour or not. We first define a 'Definite Oracle', and then, in the Section 8 relax this definition to 'Probabilistic Oracle'. In the body of the paper we only use definite oracles; the study of their probabilistic counterparts remains an open problem for future work.

Definition 2 (Definite Oracle): A *Definite Oracle* $d \in \mathbb{D}$ is a function from test activity sequences to $\{0, 1\,\text{undefined}\}$, i.e., $\mathbb{D} : \mathbb{I} \rightarrow \{0, 1, \text{undefined}\}$.

A definite oracle responds with either a 1 or a 0 indicating that the test activity sequence is acceptable or unacceptable respectively, for a test activity sequence for which it is defined. We do not require that a definite oracle be a total function, so it may be undefined. However, for a definite oracle, when it is defined, a test activity sequence is either acceptable or it is not; there is no other possibility (in contrast to the probabilistic oracle defined in Section 8).

A test activity sequence is simply a sequence of stimuli and observations. We do not wish to over-constrain our definition of oracle, but we note that, for practical purposes a test activity sequence will almost always have to satisfy additional constraints in order to be useful. In this paper we are only concerned with different approaches, their definition and their interrelationships, so we need not descend too far into the detail of these constraints. However, as an illustration, we might at least constrain a test activity sequence as follows (in order to obtain a practical test sequence):

A test activity sequence, σ, is a practical test sequence iff σ contains at least one observation activity that is preceded by at least one stimulus activity. That is, more formally:

Definition 3 (Test Sequence): A *Practical Test Sequence* is a test activity sequence:

$$\sigma_1 \frown \underline{x} \frown \sigma_2 \frown \overline{y} \frown \sigma_3$$

where $\frown$ denotes sequence concatenation and σ_1, σ_2 and σ_3 are arbitrary, possibly empty, test activity sequences.

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 may depend (the stimulus).

We conclude this section by defining completeness, soundness and correctness of oracles.

Definition 4 (Completeness): An *Oracle* is complete if it is a total function.

In order to define soundness of an 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 establishes the ways in which oracles might behave.

Definition 5 (Ground Truth): The Ground Truth, $\mathcal{G}$ is a definite oracle.

We can now define soundness of an oracle with respect to the Ground Truth, $\mathcal{G}$.

Definition 6 (Soundness): An *Oracle*, $\mathbb{D}$ is sound iff

$$\mathbb{D}\sigma \Leftrightarrow \mathcal{G}\sigma$$

We have a three valued logic, in which the connective '$\Leftrightarrow$' evaluates to false if one of the two terms it relates is undefined. However, in this context, only the evaluation of the oracle $\mathbb{D}$ can be undefined, since the ground truth is always defined.

Finally, we define total and partial correctness, in the usual way:

Definition 7 (Correctness): An oracle is partially correct iff it is sound. An oracle is totally correct iff it is sound and complete.

Observe that an arbitrary oracle AO is totally correct iff $AO = \mathcal{G}$. That is a totally correct oracle is indistinguishable from the ground truth. It is unlikely that such a totally correct oracle exists in practice. Nevertheless, we can define and use partially correct oracles in testing and one could argue, from a purely philosophical point of view, that where the oracle is the human and the ground truth is provided by the same human, then the human is a totally correct oracle; correctness becomes a subjective human assessment. The foregoing definitions allow for this case. However, this is a rather special case of a single user, who is both sole user and sole (acceptance) tester; a scenario that only occurs for comparatively uninteresting 'toy' examples.

3 Analysis of trends in Research on Test Oracles

The term "test oracle" first appeared in William Howden's seminal work in 1978 [98]. In this section we present an analysis of the research on test oracles and its related areas conducted since 1978.

3.1 Volume of Publications

We have constructed a repository of 611 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", for each year through custom searches.

We classify work on test oracles into four categories: specified oracles, derived oracles, implicit oracles and no oracle (handling the lack of an oracle).

Specified oracles, discussed in detail in Section 4, judge all behavioural aspects of a system with respect to a given formal specification. For specified 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 oracles (see Section 5) involve artefacts from which an oracle may be derived – for example a previous version of the system. For derived 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 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 oracle (see Section 7). Here, we applied the queries "human oracle", "test 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. Figure 1 shows the cumulative number of publications on each type of oracles from 1978 onwards. We analyzed the research trend on this data by applying different regression models. The trend line, shown in Figure 1, is fitted using a power model. The high vales for the four coefficients of determination (R^2), one for each of the four types of 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 oracle problem for the future.

3.2 Proposal of New Techniques or Concepts in Test Oracles

We classified the collected publications into types of techniques or concepts that have been proposed to (partially) solve the problem. For example, *DAISTIS* [72], an algebraic specification system to solve the specified oracle prob-

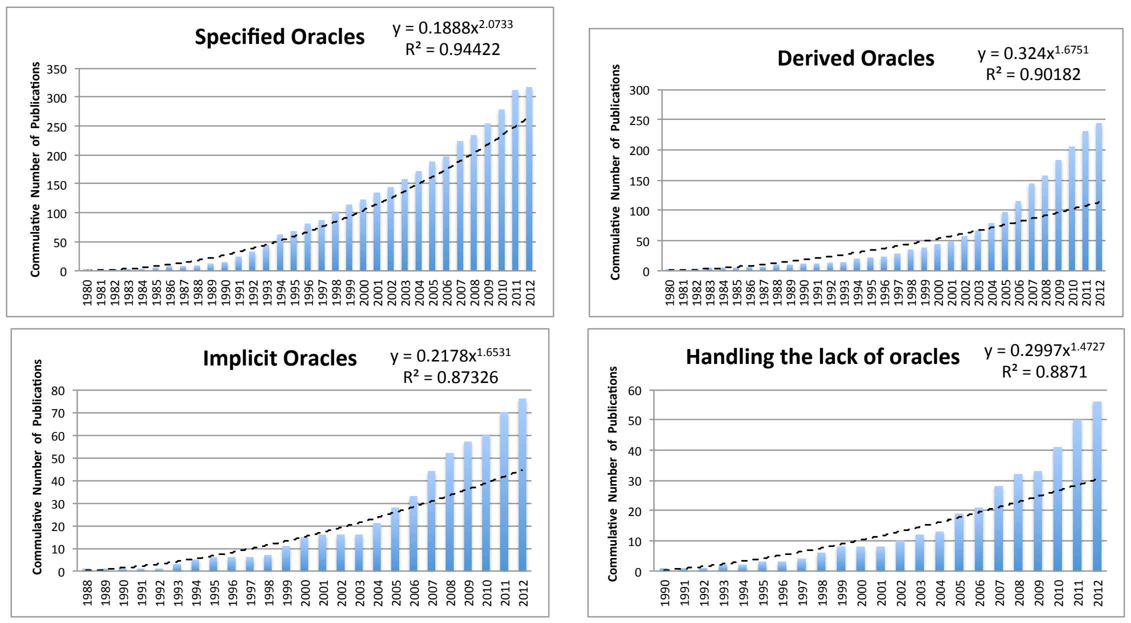

Fig. 1. Cumulative number of publications from 1978 to 2012 and research trend analysis for each type of test oracles. The x-axis represents years (x) and y-axis are the cumulative number of publications (y). The trend analysis is performed through a power regression model. The regression equation and the coefficient of determination (R^2) indicate healthy research trend in future.

lem, *Model Checking* [36] for oracle generation, and *Metamorphic Testing* [37] for oracle derivation, and so on.

For each type of oracle and the advent of a technique or a concept proposed for a type of oracle, we have plotted a timeline in chronological order of publications to study research trends. Figure 2 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 appended by the year of the 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 FSMs to generate oracles can be traced back to Jard and Bochmann [101] and Howden in 1986 [97]. We record, in the timeline, only the earliest available publication for that technique or concept. Moreover, we have only considered a published work in journals, conferences/workshops proceedings or magazines; all other types of documentation (such as technical reports and manuals) were excluded from the data.

There are few techniques or concepts that are shown pre-1978 era in Figure 2. Although not explicitly on test oracles, they address some of the issues for which test oracles have been developed in later research. For example, work on detecting concurrency issues (deadlocks, livelocks and races) can be traced back to the 1960s. This is an example of implicit oracles for which no specification is required and techniques can be employed as test oracles for arbitrary systems. Similarly, *Regression Testing*, as an example of derived oracles, can be used to detect problems in the new system version

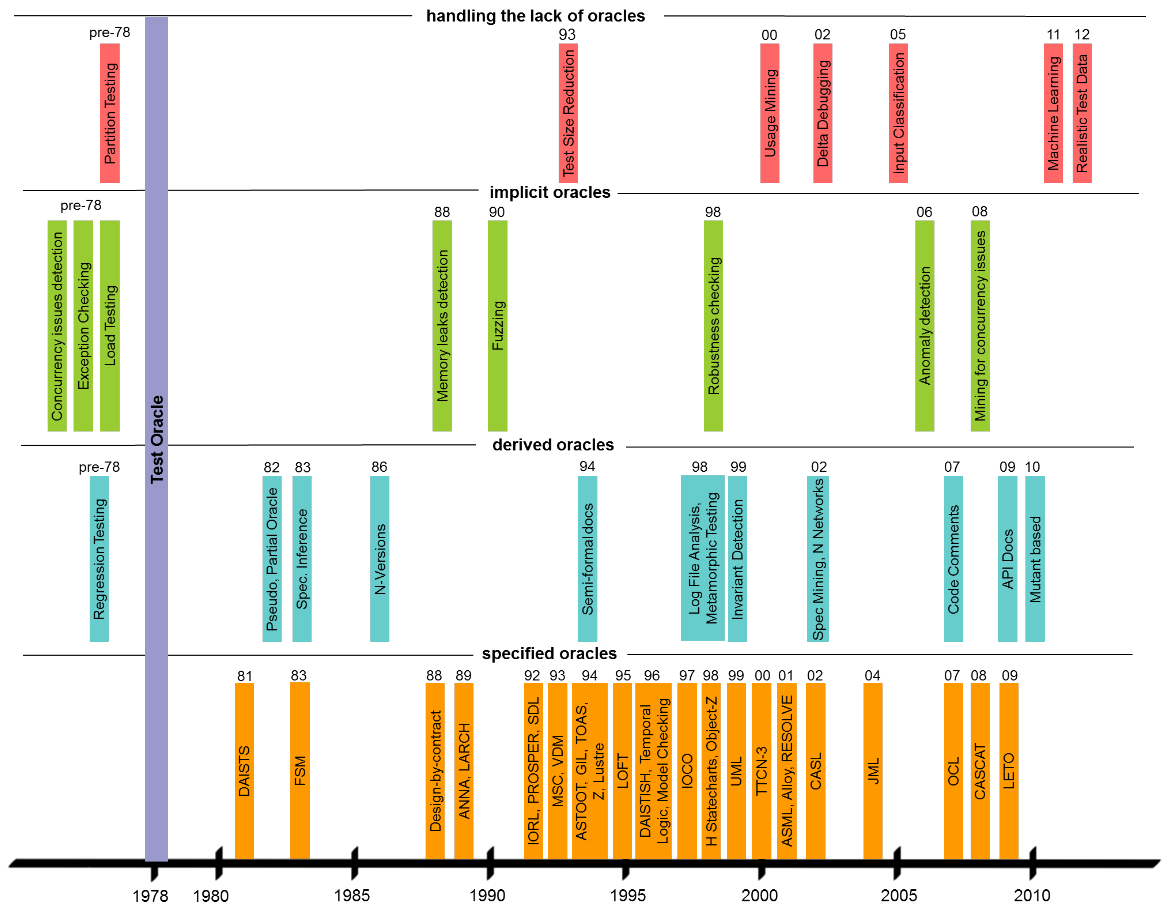

Fig. 2. Chronological order of the techniques/concepts in each type of Test Oracles

for an existing functionality.

The trend analysis suggests that proposals for new techniques and concepts for the formal specification of oracles peaked in 1990s, but have gradually diminished in the last decade. However, it has remained an area of much research activity as can be judged from the amount of publications each year, shown in Figure 1. For derived oracles, there have been quite a few solutions proposed throughout this period; but wide empirical studies on these solutions were not been conducted until the late 1990s. Previously, the solutions were primarily theoretical, such as the proposals for Partial/Pseudo Oracles [195] and Specification Inference [193]. For implicit oracle types, there have been solutions established before 1978 for which research carried on later.

Overall, there are relatively fewer new techniques or concepts proposed for this type of oracle. For handling the lack of an automated oracle, *Partition Testing* is a well-known technique that can help a human oracle in test selection. The trend line suggests that it is only recently that new techniques and concepts have started to emerge, with an explicit focus on the human oracle cost problem.

4 Specified Oracles

There has been a lot of work on specifications and the use of notions of specifications as a source of oracle information. This topic could merit an entire survey on its own right. In this section we attempt to provide an overview of work on the use of specifications as oracles, covering some of the primary types of specification-as-oracle. We include here the partial specifications of system behaviour such as assertions and models.

A specified oracle judges all behavioural aspects of a system with respect to a given formal specification. In this regard, the formal specification *is* $\mathbb{D}$ as defined in Section 2. This purposely general definition covers different notions dependent on what kind of intended behaviours are of interest, what level of abstraction is considered, and what kind of formal semantics are used.

Over the last 30 years, a large body of work has developed with respect to testing based on a formal specification, with the proposal of several methods for various different formalisms. They can grouped into three broad categories: *specification based languages*, *assertions and contracts*, and *algebraic specifications*.

4.1 Specification-Based Languages

Specification-based languages can further be divided into the categories: *model-based languages* and *transition-based languages*, which we cover in more detail below. There are other categories of specification languages that have been used as oracles (such as history-based specifications, which present oracles for ordering and timing of events in the form of properties captured by temporal logic [54]).

4.1.1 Model-Based Languages

Model-based languages are formal languages that define a mathematical model of a system's behaviour, and whose semantics provides a precise meaning for each description in the language in terms of this model. Models (when used for testing) are not usually intended as a full specification of the system. Rather, for testing, a model seeks to capture salient properties of a system so that test cases can be generated from them and/or checked against them. Model-based languages are also state-based and can also be referred to as "state-based specifications" [109] [180], [181] [100] in the literature. A variety of different formal model-based languages exist, for example, B [110], Z [169], UML/OCL [32], VDM/VDM-SL [66] and the LARCH family [74] (which include an algebraic sub-language). These are all appropriate for specifying sequential programs.

Each models the system as a collection of instance variables to represent different instances or states of the system, with operations to alter these states. Preconditions and postconditions are defined to constrain the system operations. A precondition defines a necessary condition on input states for the operation to be applied, while a postcondition describes a (usually strongest) effect condition on output states if the operation is applied [109].

If concrete system output can be interpreted at a higher level of abstraction by a pre-defined mechanism, such as a specific function, and if postconditions can be evaluated in a finite time, the postconditions of the specification can actually serve as an oracle [3].

Model and specification languages, such as VDM, Z, and B have the ability to define invariants. These can be used to drive testing. Any test case that finds a case where such an invariant is broken can be thought of as a failing test case and, therefore, these invariants are partial oracles.

4.1.2 Transition-Based Languages

Transition-based languages are languages whose syntax is often (though not always) graphical, and whose semantics are primarily concerned with the transitions between different states of the system. They have been referred as visual languages in the literature [196], though their syntax need not be graphical and their semantics is often formally defined. A variety of different formal transition-based languages exist, for example, Finite State Machines [111], Mealy/Moore machines [111], I/O Automata [117], Labeled Transition Systems [178], SDL [57], Harel Statecharts [81], UML state machines [29], X-Machines [94], [95], Simulink/Stateflow [176] and PROMELA [96].

Unlike model-based languages, which can describe arbitrarily general systems that can be infinite, transition-based languages are often used for modeling finite systems [92]. However, they can also be used to model systems with infinite sets of states. The behaviours of the system are modelled as a set of states[1], with transitions representing the actions or inputs that cause the system to change from one state to another. The outputs are normally represented by the resulting state of the system (as with Moore machines), a variable value at the resulting state (as with state charts) or a value labeled on the transition (as with Mealy machines).

Much of the work on testing using transition-based languages has been motivated by protocol conformance testing [75] and later work on model-based testing [181]. Given a specification F in a transition-based language, e.g. a finite state machine, a test case can be derived from F to determine whether the system behaviour on the application of the test conforms to F. The test oracle is embedded in the finite state machine model, and therefore, it can be derived automatically.

However, the definition of conformity comes in different flavours, depending on whether the model is deterministic or non-deterministic and the system behaviour on a given test is observable and can be interpreted at the same level of abstraction as the model's. These flavours of conformity include alternate notions such as system is isomorphic to F, equivalent to F, or quasi-equivalent to F. They are defined in the famous survey paper by Lee and Yannakakis [111], and other notable papers, including Bochmann et al. [27] and Tretmans [178].

Börger [30] discusses how Abstract State Machine language, a generalisation of specification languages like B and Z in a transition machine form, can be used to define high level oracles.

A rigorous empirical evaluation of state machine-based testing techniques can be found in the work of Mouchawrab et al. [73], [136].

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 is evaluated to true, the program behaviour is regarded to be 'as intended' for that particular execution. However, when an assertion is evaluated to false, an error has been found in the program for that particular execution. It is straightforward to see that assertions can be used as a test oracle.

Assertions have a long pedigree dating back to Turing [179] who first identified the need to separate out the tester from the developer and suggested that they might communicate through the means of assertions; the developer writing these formally and the tester checking them (equally formally). The idea of assertions gained significant attention as a means of capturing language semantics in the seminal work of Floyd [68] and Hoare [93] and subsequently was championed as a means of assisting testing in the development of the contract-based programming approach (notably in the language Eiffel [134]).

Other widely used programming languages now routinely provide special constructs to support assertions in the program; for instance, C, C++ and Java provide a construct called

1. In state-based testing, the word 'state' can be interpreted in different ways. For example, it can refer to a 'snapshot' of the configuration of s system at some point during execution (the 'system state'). However, in transition-based approaches, the word 'state' typically seeks to abstract for a specific configurations in order to capture a set of such configurations.

assert, and similarly C# provides a *Debug.Assert* method, which simply evaluates a Boolean expression and notifies if the expression does not evaluate to true. Moreover, there have been a variety of assertion systems developed independently for programming languages. For instance, Anna [115] for Ada, APP [153] and Nana [119] for C languages provide means to embed assertions in the program.

In general, assertion-based approaches only ensure a limited set of properties that must hold at a certain point in a program [51]. Languages based on *design by contract* principles attempt to improve the effectiveness 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.

This idea was originally proposed in the design of the Eiffel language [134], but extended to other languages, e.g., JML – a behavioural interface specification language for Java programs [138]. Many other authors have developed this idea of assertions as contracts that allow a formal (and checkable/executable) description of the oracle to be embedded into the program.

Cheon and Leavens [47] showed how the runtime assertion checker of JML can be used as an assertion-based test oracle language for Java. For more on specification-based assertions as test oracles, see Coppit and Haddox-Schatz's evaluation [51], and later a method proposed by Cheon [46]. Both assertions and contracts are enforced observation activity ($o \in \mathbb{O}$) that are embedded into the code. Araujo et al. [9] provide a systematic evaluation of design by contract principle on a large industrial system and show how assertion checks can be implemented for Java Modelling Language (JML) [8], while Briand et al. [34] showed how instrumentation of contracts can be used to support testing.

4.3 Algebraic Specification Languages

Algebraic specification languages describe the behaviour of a system in terms of axioms that are expressed in first-order logic and characterise desired properties. These properties are specified by formally defining abstract data types (ADT) and mathematical operations on those data types. An ADT encapsulates data together with operations that manipulate that data.

One of the earliest algebraic specification systems is DAISTS [72] for implementing, specifying and testing ADTs. The system executes a test set that is derived from the axioms with test data provided manually. Each test compares the execution of the left and right sides of each respective axiom. If the two executions generate different results, a failure is reported with the axiom name and a test number. Therefore, a test oracle in DAISTS is specific to test cases, rather than generic for arbitrary test cases.

Gaudel and her colleagues [20], [21], [75], [76] first provided a generic theory on testing based on algebraic specifications. The idea was that an exhaustive test set composed of all ground values of the axioms would be sufficient to judge program correctness. Of course, there is a practical limitation (that pertains to all testing) because the number of tests could be infinite. However, a possibility is to select a finite subset of tests [21].

Further studies by Frankl and Doong [55] to assess the practicality of Gaudel's theory concluded that tests consisting of short operation sequences may not be adequate for testing. They proposed a notation that is suitable for object-oriented programs and developed an algebraic specification language called LOBAS and a tool called ASTOOT.

One important aspect of the approach was the notion of self-checking test cases for classes, that use a class method that approximates observational equivalence. Chen et al. [42] [43] further expanded the ASTOOT approach in their tool TACCLE and proposed an algorithm for generating a relevant finite number of test sets by employing a heuristic white-box technique. Several other algebraic specification languages and tools exist, e.g., Daistish [99], LOFT [122], CASL [12], CASCAT [204]. Zhu also considered the use of algebraic specifications as test oracles [209], while Bochmann et al. [182] used LOTOS to realise oracle functions from algebraic specifications.

For specified oracles, the interpretation of test results and checking their equivalence with specified results can be a difficult task. The specified results are usually defined on an abstract level, and test results depend on program executions that may appear in a different way such that their equivalence to those specified is not straightforward. Moreover, specified results could be partially represented or oversimplified.

Gaudel [75] remarked that the existence of a formal specification does not guarantee the existence of a successful test driver. It may be necessary to leverage the interpretation of results with some concrete equivalence functions [118]. However, by and large, solutions to this problem depend largely on the level of abstraction and also on the implementation context of the system under test on some extent (see Gaudel's discussion [75] for more on this issue).

5 Derived Oracles

When specified oracles are unavailable, oracles can be derived from various artefacts (e.g. documentation, system executions) or properties (e.g. metamorphic relations) of the system under test, or other versions of it. Of course, the derived oracle might then become a partial 'specified oracle', so that oracles derived by the methods discussed in this section could migrate, over time, to become, those considered to be the 'specified oracles of the previous section. For example, JWalk incrementally learns algebraic properties of the class under test [167]. It allows interactive confirmation from the tester, ensuring that the human is in the 'learning loop'. The following sections discuss research in some of these artefacts for the purpose of oracle derivation.

5.1 Pseudo Oracles and N-versions

One of the earliest versions of a derived oracle is the concept of a pseudo-oracle, introduced by Davis and Weyuker [52], 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." [195]. 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. A similar idea exists in fault-tolerant computing, referred to as multi- or N-versioning [14], [15], where the software is implemented in multiple ways and executed in parallel. Where results differ at run-time, a "voting" mechanism is used to decide which output would be used. More recently, Feldt [61] investigated the possibility of automatically producing different versions using genetic programming, while McMinn [126] 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

Metamorphic testing is a process of generating partial oracles for *follow-up* test cases: it checks important properties of the SUT after certain test cases are executed [37]. The important properties are captured by metamorphic relations. It is often thought that the metamorphic relations need to concern numerical properties that can be captured by arithmetic equations, but metamorphic testing is, in fact, more general than this. For example, Zhou et al. [208] used metamorphic testing to test search engines such as Google and Yahoo!, where the relations considered are clearly non-numeric.

A metamorphic relation is an expected relation among multiple test activity sequences. For example, let us assume that we are testing an implementation of a double precision sine function, f. One possible metamorphic relation can be defined as follows: let i_1 be the first test activity sequence, which is a sequence $\langle x, f(x) \rangle$ for a valid double value x. From the definition of the sine function, we expect to obtain the second test activity sequence i_2, which is a sequence $\langle x + \pi, f(x) \rangle$, if the implementation f is correct.

More formally, we define metamorphic oracles in terms of the definitions from Section 2 as follows: There is a reliable reset, which is a stimulus, $\underline{R}$, that allows us to interpose a reset between a previous test case, $\langle \underline{x_1}, \overline{y_1} \rangle$, and a subsequent test case, $\langle \underline{x_2}, \overline{y_2} \rangle$, to form a five-element test sequence $\langle \underline{x_1}, \overline{y_1}, \underline{R}, \underline{x_2}, \overline{y_2} \rangle$, for which we can define a relationship, the "metamorphic testing relationship", π, between x_1, y_1, x_2 and y_2.

For the formulation of metamorphic testing in the current literature π is a 4-ary predicate[2], relating the first input-output pair to the second:

$$\mathbb{D}\langle \underline{x_1}, \overline{y_1}, \underline{R}, \underline{x_2}, \overline{y_2} \rangle \text{ if } \pi(x_1, y_1, x_2, y_2)$$

It is a natural step to generalise this metamorphic relationship to an arbitrary property, p, of the test activity sequence. Such an arbitrary property must respect the oracle $\mathbb{D}$, for all test activity sequences, σ so

$$\mathbb{D}\sigma \text{ if } p\sigma$$

of which, $\mathbb{D}\langle \underline{x_1}, \overline{y_1}, \underline{R}, \underline{x_2}, \overline{y_2} \rangle$ if $\pi(x_1, y_1, x_2, y_2)$ is clearly a special case. This generalisation is considered in more detail in Section 8.

For deterministic SUTs, oracles from metamorphic relations are complete and sound, but only with respect to the properties that form the relations. For example, if $f : \mathbb{R} \to 0$, it passes the aforementioned metamorphic testing but is still incorrect. However, metamorphic testing provides useful means of testing non-testable programs introduced in the last section.

2. Our formalism is more general than traditional metamorphic testing and can handle arbitrary relations.

When the SUT is not deterministic, it is not possible to use equality relations in the process of metamorphic testing. Murphy et al. [137], [138] relaxed the equality relation to set membership to cope with stochastic machine learning algorithms. Guderlei and Mayer introduced statistical metamorphic testing, where the relations for test output are checked using statistical analysis [80], which was later used to apply metamorphic testing to stochastic optimisation algorithms [202].

The biggest challenge in metamorphic testing is the automated discovery of metamorphic relations. Some of those in the literature are mathematical [37], [39], [44] or combinatorial [137], [138], [158], [202]. There is also work on the discovery of algebraic specifications [87], and the JWalk lazy testing approach [167], both of which might be adapted to discover oracles. There also exists early work on the use of metamorphic relations based on domain specific knowledge [40], however it is still at an early stage, and not automated.

5.3 Regression Test Suites

With regression testing, the aim is to detect whether the modifications made to the new version of SUT have interfered with any existing functionalities [203]. There is an implicit assumption that the previous version can serve as an oracle for regression testing.

For corrective modifications, the functionalities remain the same and the oracle for version i, $\mathbb{D}_i$, can serve for the next version as $\mathbb{D}_{i+1}$. Orstra [199] generates assertion-based test oracles by observing the program states of the previous version while executing the regression test suite. The regression test suite, now augmented with assertions, can be used with the newer versions. Similarly, *spectra*-based approaches uses the program and value spectra obtained from the original version to detect regression faults in the newer versions [85], [200].

For perfective modifications that add new features to the SUT, $\mathbb{D}_i$ should also be modified to cater for newly added behaviours, i.e. $\mathbb{D}_{i+1} = \mathbb{D}_i \cup \Delta D$. Test suite augmentation techniques specialise in identifying and generating ΔD [5], [130], [201].

There is also a class of modifications that proceed by changing the specification (which is deemed to fail to meet requirements, perhaps because the requirements have changed). This is 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 which arise due to changed requirements (or incorrect specifications).

Our formalisation of oracles in Section 2 forces a distinction of these two categories of perfective maintenance, since the two have profoundly different consequences for oracles. We therefore refer to this new category of perfective maintenance "changed requirements". For changed requirements:

$$\exists \sigma \cdot \mathbb{D}_{i+1}\sigma \neq \mathbb{D}_i\sigma$$

which implies, of course $dom(\mathbb{D}_{i+1}) \cap dom(\mathbb{D}_i) \neq \emptyset$ and the new oracle cannot be simply 'unioned' with the old oracle. The relationship between this observation and the emerging issues pertaining to Software Product Lines and their inherent connections to regression testing are all explored in Section 8.

5.4 System Executions

A system execution trace is a critical artifact that can be exploited to derive oracles. The two main techniques, invariant detection and specification mining for oracle derivation, are discussed in the following sections. Both techniques produce automated checking of expected behaviour similar to assertion-based specification, discussed in Section 4.2.

5.4.1 Invariant Detection

Program behaviours can be checked against the given invariants for violations automatically. 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. [59], implemented in the Daikon tool [58], is to execute a program on a collection of inputs (test cases) and infer *likely* invariants from program executions dynamically. The invariants inferred 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.

Invariant detection can be computationally expensive, but there have been incremental [23], [168] and light weight static analyses [41], [67] that can be brought to bear in this problem. There is a technical report available that summarises various dynamic analysis techniques [155]. Model inference [89], [185] 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. [151] used Search-Based Software Engineering (SBSE) to search for invariants, guided by mutation testing.

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 [59]. Nevertheless, inferring "perfect" invariants is almost impossible with the current state of the art, which tends to frequently infer incorrect or irrelevant invariants [149]. Wei et al. [190] and [191] recently improved the quality of inferred invariants.

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 an error-prone process and the misclassification of invariants is frequent. As found by Staats et al. [172] in a recent empirical study, half of the incorrect invariants were misclassified that were originally inferred by Daikon from a set of Java programs. However, despite these issues, research on the dynamic inference of program invariants has gained a strong momentum in the recent past with a prime focus on its application for test generation [11] [140] [206].

5.4.2 Specification Mining / Inference

Specification mining or inductive inference is a concept for hypothesizing a formal model of program behaviours from a set of observations. In her seminal work [193], Weyuker made the connection between inference and testing as inverse processes. 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 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, indicating that the test set T used to infer the program P is not adequate.

The 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 applying monitors, or actively, e.g., by querying the system [104]. However, equivalence checking is undecidable in general, and therefore inference is only possible for programs in a restricted class, e.g., those with behaviours that can be modelled by finite state machines [193]. With this, equivalence can be accomplished by performing checking experiments [88]. 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, i.e, when source code/behavioural models are unavailable. Most work has applied a well-known learning algorithm, called L^* [7] for learning a black-box system B having n states. The algorithm infers a state machine model by querying B and observing the corresponding outputs iteratively. At each iteration, an inferred model M_i with $i < n$ states is given. Then, the model is refined with the help of a distinguishing string (that distinguishes B and M_i) to produce a new model, until the number of states reaches to 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. [113] have applied L^* for integration testing of a system of black-box components. In this context, the oracle is the system architecture that can be stimulated to obtain faulty interactions between the components, by comparing them with their inferred models.

Further work in this context has been compiled by Shahbaz [163] with industrial applications. Similar applications of inference can be found in system analysis [79] [186] [187] [22], [133], component interaction testing [114], [121], regression testing [200] [198], security testing [19] [165] and verification [146] [78] [56].

Zheng et al. [207] mine test oracles for web search engines, which had been thought to be untestable. Frouchni et al. [71] apply machine learning to learn oracles for image segmentation programs. Memon et al. [131], [132], [197] introduced and developed the GUITAR tool, which has been evaluated by treating the current version as correct, inferring the specification, and then executing the generated test inputs. GUITAR is a GUI-based testing tool that allows test cases to be constructed in terms of GUI feature interactions. Artificial Neural Networks have also been applied to learn system behaviour and detect deviations from it [160], [161]

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 therefore can serve as a basis for generating test oracles. In other words, the documentation is $\mathbb{D}$ (as defined in Section 2).

At first sight, it may seem impossible to derive 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 [142]. However, some partial automation can assist the human in testing using documentation as a source of oracle information. Attempts to make such automated use of textual documentation can be grouped into three categories.

5.5.1 Converting Textual Documents to Formal Specification

The first category of work builds techniques to construct a formal specification out of the given textual specification and resolve ambiguities and inconsistencies during the process. Thereafter, techniques to derive test oracles from a formal specification can be employed.

Prowell and Poore [150] introduced a sequence enumeration method to develop a formal specification from an informal one. This is done by systematically enumerating all sequences from the input domain and mapping the corresponding outputs to produce an arguably complete, consistent, and correct specification. However, it can suffer from an exponential explosion of the possible input/output sequences. Prowell and Poore suggest to employ abstraction techniques for controlling the growth of an inherently combinatorial process. The end result of the process is a formal specification that can be transferred into a number of notations, e.g., transition-based languages. One benefit of the approach is that many inconsistent and missing requirements can be found during the process, which helps in making the specification more complete and precise.

5.5.2 Formalising Textual Documents

The second category of work imposes structures and formalities to the textual documents in order to make them clear, formal and precise such that oracles can be derived automatically.

Parnas et al. [141] [144] [147] proposed TOG (Test Oracles Generator) from program documentation. In their method, the documentation is written in fully formal tabular expressions [103] in which the method signature, the external variables, and relation between its start and end states are specified. Thus, oracles can be generated automatically to check the outputs against the specified states of a program. The work by Parnas et al.[3] has been developed over a considerable period of more than two decades [50], [62], [63], [143], [148], [188], [189].

5.5.3 Restricting Natural Languages

Restrictions on natural languages can reduce complexities in their grammar and lexicon.

3. This work could also have been categorised as an assertion-based method, but we located this work here in the survey, since it is derived from documentation.

These restrictions allow expressing requirements with minimum vocabulary and ambiguity. Thus, interpretation of the documents is easier and automatic derivation of test oracles can be possible.

Schwitter [157] introduced a computer-processable controlled natural language called PENG. It covers a strict subset of standard English with a restricted grammar and a domain specific lexicon for content words and predefined function words. The documents written in PENG can be translated deterministically into first-order predicate logic. Schwitter et al. [31] provided a writing guideline for test scenarios in PENG which can be used to judge the runtime program behaviours automatically.

6 Implicit Oracles

A critical aspect of the implicit oracle is that it requires no domain knowledge or formal specification to implement, and therefore it can be applied to all runnable programs in a general sense. An oracle that detects anomalies such as abnormal termination (due to a crash or an execution failure) is an example of an implicit oracle [35], [164]. This is because such anomalies are *blatant faults*, that is, no more information is required to ascertain whether the program behaved correctly or not. More generally, an implicit oracle defines a subset of observation alphabets $\mathbb{O}$ as guaranteed failures.

Such implicit oracles are always context sensitive; there is no single universal implicit oracle that can be applied to all programs. Behaviours that are considered abnormal for one system in one context may not be abnormal in a different context (or for a different system). Even crashing *may* be considered acceptable (or even good behaviour for some systems; for example those that are designed to find such issues).

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 [26] [105] [183] [17] [166]. Similarly, research on testing non-functional attributes such as performance [120], [123] [194], robustness [107] and memory leak/access [210] [60] [13] [86] have garnered much attention since the advent of the object-oriented paradigm. Each one of these topics deserves a survey of its own, and therefore the interested reader is directed elsewhere (e.g., Cheng [45], Luecke et al. [116]).

Fuzzing [135] is one effective way of finding implicit anomalies. The main idea is to generate random (or "fuzz") inputs and attack the system to find those anomalies. If an anomaly is detected, the fuzz tester reports the anomaly with the set of inputs or input sequences caused it. Fuzzing is commonly used to detect security vulnerabilities, such as buffer overflows, memory leaks, unhandled exceptions, denial of service etc. [174].

While fuzzing can be used for detecting anomalies in black-box systems, finding concurrency issues in such systems is hard. Groz et al. [79] proposed an approach that extracts behavioural models from systems through active learning techniques (see Section 5.4.2 for details) and then performs reachability analysis [28] to detect issues, notably races, in asynchronous black-box systems.

Other work has focused on developing patterns to detect anomalies. For instance, Ricca and Tonella [152] considered a subset of possible anomalies that can be found in Web applications, e.g., navigation problems, hyperlink inconsistencies etc. Their empirical assessment showed that 60% of the Web applications considered in their study exhibited anomalies and execution failures.

7 Handling the Lack of Oracles

The above sections give solutions to the oracle problem when some artefact exists that can serve as either a full or partial oracle. However, in many cases no such artefact exists (i.e., there is no automated oracle $\mathbb{D}$) and as such, a human tester is required to verify whether software behaviour is correct given some stimuli.

Despite the fact that there is no automated oracle, there is still a role that software engineering research can play, and that is to reduce the effort that the human tester has to expend in being the oracle. This effort is referred to as the *Human Oracle Cost* [124]. Research techniques can be employed to reduce the human oracle cost by a *quantitative* reduction in the amount of work the tester has to do for the same amount of test coverage, or by making the evaluation of test cases easier to process in a *qualitative* sense.

7.1 Reducing Quantitive Human Oracle Cost

Test suites can be unnecessarily large, covering few test goals in each individual test case. Or, 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 an oracle.

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, [84], Offutt et al. [139] and Rothermel et al. [154]. However, there has been recent work in the search-based testing literature that has sought to combine test input generation and test suite reduction into one phase to produce smaller test suites.

Harman et al. [82] propose an approach that attempts the generation of test cases that penetrate the deepest levels of the control dependence graph for the program, in order to encourage each generated test case to exercise as many elements of the program as possible. Ferrer et al. [65] attack the same problem but with 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 amongst others. On a series of randomly-generated programs and small benchmarks, MOCell was found to perform the best.

Taylor et al. [175] use an inferred model as a semantic oracle to reduce a test suite. Fraser and Arcuri [69] report on the EvoSuite tool for search-based generation of test suites for Java with maximal branch coverage. Their work is not directly aimed at reducing oracle cost, but their approach attempts to generate an entire test suite at once. The results show that not only coverage is increased, but test suite sizes are also reduced.

7.1.2 Test Case Reduction

When using randomised algorithms for generating test cases for object-oriented systems, the length of individual test cases can become very long 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). Method calls that do not contribute to a test case unnecessarily increase oracle cost, and there has been work [112] where such calls have been removed using Zeller's Delta Debugging technique [205].

7.2 Reducing Qualitative Human Oracle Cost

Human oracle costs may also be minimised from a qualitative perspective. That is, the extent to which test cases may be easily understood and processed by a human. Due to a lack of involvement of domain knowledge in the test data generation process, machine-generated test data tend not to match the expected input profile of the software under test. While this may be beneficial for trapping certain types of faults, the utility of the approach decreases when 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.

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 [124]. With search-based approaches, it is possible to inject 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, or an already existing, partial test suite.

The generation of string test data is particularly problematic for automatic test data generators, which tend to generate non-sensical strings. In [128], McMinn et al. propose the use of strings mined from the web to assist in the test generation process. Since the content in web pages is generally the result of human effort, the strings contained in web pages tend to be real words or phrases that can be utilized as 'realistic' sources of test data.

A similar idea is employed by Bozkurt and Harman [33], where web services are tested using the outputs of other test services. Since the outputs of web services are 'realistic', they form realistic test inputs for other web services.

Afshan et al. [1] propose the use of a natural language model to help generate readable strings. The language model scores how likely a string is to belong to a language based on the character combinations. By using this probability score as a component of a fitness function, a metaheuristic search can be used not only to cover test goals, but also to generate string inputs that are more comprehensible than the arbitrary strings normally generated. Afshan et al. found that for a number of case studies, test strings generated with the aid of a language model were more accurately and more quickly evaluated by human oracles.

Fraser and Zeller [70] 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. JWalk [167] simplifies longer test sequences, thereby reducing oracle costs where the human is the oracle.

Staats et al. [170] seeks to reduce the human oracle cost by guiding human testers to those parts of the code they need to focus on when writing oracles. This reduces the cost of oracle constriction, rather than reducing the impact of a human testing in the absence of an oracle.

7.3 Crowdsourcing the Oracle

A recent approach to handling the lack of an 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. [145] 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]

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

and uTest [7].

8 Future Research Directions

In this section we set out an agenda for further research on the oracle problem. We use our formalisation of oracles from Section 2 to explore some of the extensions that might be applied for metamorphic oracles and for software product line testing, and show how we might combine notions of metamorphic and regression testing. We also consider extensions to our oracle definition to cater for probabilistic oracles, and to cater for the highly prevalent and therefore practically important problem of handling situations where there simply is no automated oracle.

8.1 Extending Metamorphic Testing

Metamorphic testing has great potential as a means to overcome the lack of an oracle and it continues to be extended (for example to context-sensitive middleware [38] and service-oriented software [39]). In this section, we consider other possible extensions to metamorphic testing. Recall that metamorphic testing can be regarded as a special case of property testing for an arbitrary property p that must respect the oracle $\mathbb{D}$, for all test activity sequences, σ so

$$\mathbb{D}\sigma \text{ if } p\sigma$$

This simple generalisation reveals that metamorphic testing is a form of property testing and, thereby, allows us to consider other forms of property testing that retain the essence of metamorphic testing, but extend current metamorphic concepts. For example, we can, rather trivially, extend the sequence of input-output pairs. We need not consider the metamorphic property to be a 4-ary predicate, but can simply regard it as a unary predicate on arbitrary-length test activity sequences, a simple generalisation that allows metamorphic testing of the form:

$$\mathbb{D}\langle \underline{x_1}, \overline{y_1}, \ldots, \underline{x_k}, \overline{y_k}, \underline{R}, \underline{x_{k+1}}, \overline{y_{k+1}}, \ldots, \underline{x_n}, \overline{y_n}\rangle$$
$$\text{if}$$
$$\pi(x_1, y_1, \ldots, x_n, y_n)$$

for some sequence of n stimulus-observation pairs, the first k of which occur before the reset and the final $n-k$ of which occur after the reset.

We can also relax the constraint that the test input should contain only a single reliable reset to allow multiple resets as is common in state-based testing [53]. We could also relax the constraint that between the reliable resets where there is exactly one stimulus and one observation. Perhaps metamorphic testing would be more useful if it allowed a more complex interplay between stimuli and observations. The constraint that the oracle would place on such a relaxed notion of metamorphic testing would be

$$\mathbb{D}\sigma \text{ if } \pi\sigma$$

for all test activity sequences σ that contain at least one reliable reset. This raises the question about whether or not we should even require the presence of a single reliable reset operation; i.e, why the case of zero reliable resets not considered?

Of course, were we to relax this constraint and allow arbitrarily many reliable resets (including none) then we would have generalised Metamorphic Testing to the most general case; simply property testing, formulated within our framework in terms of test activity sequences. That is

$$\mathbb{D}\sigma \text{ if } \pi\sigma$$

This seems to be an over-generalisation because it does not seem to retain the spirit of metamorphic testing. For instance, using such a formulation, we could capture the property that the first n outputs (observations) of the system under test should be sorted in ascending order and should appear without any input being required (no stimuli):

$$\mathbb{D}\langle \overline{y_1}, \ldots, \overline{y_n}\rangle \text{ if } \forall i . 1 \leq i < n \cdot y_i \leq y_{i+1}$$

7. http://www.utest.com/

This does not have any of the feeling of the original definition of metamorphic testing because there is no concept of relating one input-output observation to another. It seems that the concept of the reliable reset is central to the notion of metamorphic testing. Perhaps a generalised definition of metamorphic testing that may prove useful for future work would define a metamorphic relation to be a relation on a sequence of at least two test sequences, separated by reliable resets and for which each test sequence contains at least one observation preceded by at least one stimulus.

Recall that a test sequence contains at least one observation activity preceded by at least one stimulus activity, according to Definition 3, and that stimuli and observations, as construed in this paper, are deliberately very general: the stimulus could merely consist of calling the main function, starting the system or reseting, while the observation could be that the system terminates within a specified time without any output, or that it prints 42 on a console, or that it consumes no more than 0.002 watts of power in its first second of operation.

We can define a metamorphic test sequence more formally as follows:

$$\sigma_1 \frown \langle \underline{R} \rangle \frown \sigma_2 \frown \langle \underline{R} \rangle \frown \ldots \frown \langle \underline{R} \rangle \frown \sigma_n$$

for some set of test sequences $\{\sigma_1, \ldots, \sigma_n\}$, where $n \geq 2$.

This makes clear the importance of reliable resets and the constraint that a metamorphic test process involves at least two test sequences. Finally, having formally defined a (general) metamorphic test sequence, we can define Generalised Metamorphic Testing (GMT) as follows: Given a metamorphic test sequence, Σ, GMT respects the oracle constraint

$$\mathbb{D}\Sigma \text{ if } \pi\Sigma$$

That is, GMT is nothing more than property testing on (general) metamorphic test sequences.

Using GMT we can capture more complex metamorphic relationships than with traditional metamorphic testing. For example, suppose that a tester has three stimuli activities available, each corresponding to different channels. Stimulus $\underline{s_1}$ stimulates an output on channel c_1, while stimulus $\underline{s_2}$ stimulates an output on channel c_2. There is a third channel that must always contain a log of all outputs on the other two channels. The contents of this third channel are output in response to stimulus $\underline{s_3}$. This metamorphic logging relationship, π_L, can be captured quite naturally by GMT:

$$\pi_L \langle \underline{s_1}, \overline{y_1}, \underline{R}, \underline{s_2}, \overline{y_2}, \underline{R}, \underline{s_3}, \overline{y_3} \rangle \text{ iff } \{y_1, y_2\} \subseteq y_3$$

We can also capture properties between multiple executions that rely on persistence, the archetype of which is a persistent counter. For example, the generalised metamorphic relation below, π_P, captures this archetype of persistence that stimulus $\underline{t}$ causes the observation $\overline{c}$, which reports a count of the number of previous stimuli $\underline{s}$ that have occurred. Let $\{\sigma_1, \ldots, \sigma_n\}$ be a set of n test sequences, such that σ_i contains T_i occurrences of the stimulus $\underline{s}$.

$$\begin{gathered} \pi_P(\sigma_1 \frown \langle \underline{R} \rangle \frown \ldots \frown \langle \underline{R} \rangle \frown \sigma_n \frown \langle \underline{t}, \overline{c} \rangle) \\ \text{iff} \\ c = \sum_i^n T_i \end{gathered}$$

This analysis shows that there may be interesting and potentially useful generalisations of metamorphic testing. Of course, this discussion has been purely theoretical. Future work on metamorphic oracles could explore the practical implications for these more general forms of metamorphic relation, such as those we briefly discussed above.

8.2 Extending Regression Testing

Software Product Lines (SPLs) [49] are sets of related versions of a system. A product 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. Since SPLs are about versions of a system, there should be a connection between regression testing and SPL testing upon which our definition of oracles may shed some light: we should be able to formulate oracles for SPLs in a similar way to those for regression testing. Indeed, the sequence of releases to which we might apply regression testing ought to be merely a special case of an SPL testing, in which the degree of the tree is 1 and thus the tree is a degenerate tree (i.e., a sequence).

Recall that, for regression testing, we have two releases, R_1 and R_2 for which we have oracles D_1 and D_2 respectively. We clarified the distinction between perfective maintenance, corrective maintenance and changed functionality. In corrective maintenance, the implementation is found to be faulty and has to be fixed, but this has no impact on the oracle, which remains unchanged:

$$\mathbb{D}_1 = \mathbb{D}_2$$

In perfective maintenance R_2 adds new functionality to R_1 (in which case the domains of D_1 and D_2 are expected to be distinct):

$$dom(\mathbb{D}_1) \cap dom(\mathbb{D}_2) = \emptyset$$

With changed functionality, the requirements are altered so that the new implementation must change the behaviour of the previous implementation, not because the previous version is incorrect (which would have been merely an instance of corrective maintenance), but because the requirements have changed. In this 'changed requirements' case, there is a clash between the two oracles:

$$\exists \sigma \cdot \mathbb{D}_1\sigma \neq \mathbb{D}_2\sigma$$

Suppose we have two branches of an SPL, B_1 and B_2, each of which shares the functionality of a common base system B. We can think of the sequence of releases: B followed by B_1 as a regression testing sequence. Similarly, we can think of B followed by B_2 as a different regression test sequence.

Now suppose that the oracles for B, B_1 and B_2 are $\mathbb{D}$, $\mathbb{D}_1$ and $\mathbb{D}_2$ respectively. If $\mathbb{D} = \mathbb{D}_1$ then this branch of the SPL is really a correction to the base system and it should be merged back into the base system (and similarly for the case where $\mathbb{D} = \mathbb{D}_2$). Furthermore, we can see that if $\mathbb{D} = \mathbb{D}_1 = \mathbb{D}_2$ then there is no need for a branch at all; the same oracle, $\mathbb{D}$ can be used to test all versions of the system, and so we can meaningfully say that there are no separate versions.

It is not hard to imagine how such a situation might arise in practice: an engineer, burdened with many bug fix requests, is concerned that some specific big fix is really a feature extension and so a new branch is created for a specific customer to implement their 'bug fix' request, while leaving the existing base system as a separate line in the SPL (for all other customers). This is understandable, but it is not good practice. By reasoning about the oracles and their relationships, we can determine whether such a new branch is truly required at the point at which it might be created.

This observation leads us to define the minimal requirements on the oracle of a version for it to be necessary to create a new SPL branch. Suppose we seek to branch into two new versions. This is a typical (but special) case, that easily generalises to the case of one and n new branches. We expect that the two new versions extend the base system with new functionality and that they do so without interfering with the existing core functionality of the base system and that each extension is conflicting, so that two branches are required, one for each different version. More formally:

$$\begin{gathered} dom(\mathbb{D}) \cap dom(\mathbb{D}_1) = \emptyset \\ \wedge \\ dom(\mathbb{D}) \cap dom(\mathbb{D}_2) = \emptyset \\ \wedge \\ \exists \sigma \cdot \mathbb{D}_1\sigma \neq \mathbb{D}_2\sigma \end{gathered}$$

We can see that this is none other than requiring that the two branches are 'perfective maintenance' activities on the base system and that each has changed requirements. Where these constraints are not met, then the branches may be unnecessary and could (perhaps should) be merged. Using such a simple formulation we might search for such "mergeable" branches, thereby reducing the problematic SPL phenomenon of "branchmania" [25].

8.3 Metamorphic Regression Testing

Using the formalisation of the oracle concepts introduced in this paper, we can also define and investigate entirely new forms of oracle that straddle existing forms of testing and approaches to oracle definition. For instance, suppose we consider what it would mean to combine metamorphic testing with regression-based testing.

First, we observe that we can define some forms of regression testing in terms of metamorphic testing. Consider corrective maintenance tasks. For these tasks we seek to correct a bug, but we use an unchanged oracle.

Suppose we interpret the reliable reset as the "bridge" between the two versions of the system. That is, we construct a composition of the original version and the new version with the reliable reset $\underline{R}$ denoting the stimulus of switching execution in the composition from original to corrected version. For clarity, we shall use prime variables when referring to the new version of the system. That is $\underline{x}'$ will refer to the stimulus x, but applied to the new version of the system (to which we could apply stimulus $\underline{x}$), while $\overline{y}'$ will refer to the observation y applied to the new version of the system.

Recall our original definition of (ungeneralised) metamorphic testing:

$$\mathbb{D}\langle \underline{x_1}, \overline{y_1}, \underline{R}, \underline{x_2}, \overline{y_2} \rangle \text{ if } \pi(x_1, y_1, x_2, y_2)$$

Suppose we are regression testing a corrected deterministic system. Let T be the set of stimuli for which the original program is already correct. We can define the regression testing task for corrective maintenance in terms of traditional metamorphic testing of the composition of old and new versions as follows:

$$\forall x_1 \in T.\mathbb{D}\langle \underline{x_1}, \overline{y_1}, \underline{R}, \underline{x_2'}, \overline{y_2'} \rangle \text{ if } \pi(x_1, y_1, x_2, y_2)$$

where $\pi(x_1, y_1, x_2, y_2)$ iff $x_1 = x_2 \Rightarrow y_1 = y_2$. In this way π constrains the oracle such that:

$$\forall x \in T.\mathbb{D}\langle \underline{x}, \overline{y}, \underline{R}, \underline{x'}, \overline{y'} \rangle$$

revealing that we can define the metamorphic relation π in such a way as to exactly capture corrective maintenance. That is, the relationship between the original and the corrected version is that the behaviour is unchanged (all observations remain the same) for all stimuli in the correct set (there are no regressions). In this way we see that, while traditional metamorphic testing is a relationship between two executions of the same system, regression testing can be thought of as a metamorphic relation between the same execution on two different versions of the system.

This treatment covers traditional regression testing, which seeks to check that the software has not regressed. That is, we check that previously correct behaviour remains correct. Of course, regression testing is usually also accompanied by testing new changed functionality that has been added to the system under test. This is traditionally thought of as an entirely separate activity, which requires a new specification (and oracle) for the changes made.

Where those changes are the result of perfective maintenance, we will indeed need a new oracle. However, what of those changes in requirements that are defined in terms of the original behaviour? Customers often report buggy behaviour with an implicit specification of desired behaviour in terms of existing behaviour. For example, consider a customer who makes statements like this:

> "the system is correct, but needs to be at least twice as fast to be useful."

or

"the reading is always off by one; it should be one lower than reported."

or

"the battery life needs to be at least 10% longer."

When testing changes made to achieve these improvements, we will perform regression testing and, at the same time, the tester will also check that the desired improvements have been made. In practice, the tester will not separate out the act of regression testing from testing for improvements, so why should the theory do so and why should the research community regard these two kinds of test activity as entirely different?

Fortunately, with our notion of "metamorphic regression testing", we can capture these kinds of "upgrade"; changes that improve the system's behaviour with respect to its existing behaviour.

Consider our generalised metamorphic requirement, but in which we are testing the composition of the original and upgraded system and, as before, $\underline{R}$ denotes switching execution from the old to the new version of the system. To capture the first form of upgrade above:

$$\mathbb{D}(\sigma \frown \langle \overline{t_1}, \underline{R} \rangle \frown \sigma \frown \langle \overline{t_2'} \rangle \text{ if } \pi(\_, t_1, \_, t_2)$$

where σ is an arbitrary test activity sequence and t_1 and t_2 are observations of execution time and $\pi(\_, t_1, \_, t_2)$ iff $t_2 \geq 2t_1$.

To capture the second form of upgrade above:

$$\mathbb{D}\langle \underline{x}, \overline{y_1}, \underline{R}, \underline{x}, \overline{y_2'} \rangle \text{ if } \pi(\_, y_1, \_, y_2)$$

where x is the stimulus that causes the incorrect reading and $\pi(\_, y_1, \_, y_2)$ iff $y_2 = y_1 - 1$.

To capture the third form of upgrade above:

$$\mathbb{D}(\sigma \frown \langle \overline{b_1}, \underline{R} \rangle \frown \sigma \frown \langle \overline{b_2'} \rangle \text{ if } \pi(\_, b_1, \_, b_2)$$

where σ is an arbitrary test activity sequence and b_1 and b_2 are observations of the remaining battery time and $\pi(\_, b_1, \_, b_2)$ iff $b_2 \geq 1.1b_1$.

As can be seen, metamorphic regression testing can be used to capture situations in which the new system is, in some sense, an upgrade of the behaviour of the original. In such cases the desired behaviour of the upgrade is expressed in terms of the less favourable existing behaviour, and regression testing can be thought of as a new form of metamorphic testing; one which relates two different versions of a system rather than the traditional form of metamorphic testing, which relates two different executions of the same system.

These definitions of regression testing in terms of metamorphic testing open up the possibility of research into new ways of combining these two important, but hitherto separate, forms of testing through a shared understanding of the oracle. Future research could explore the practical implications of the theoretical relationships our analysis has uncovered.

8.4 Generalisations of Oracles and Their Properties

Future work may also consider the way in which our notion of oracles could be generalised and the theoretical properties of oracles. In this section we illustrate these more theoretical possibilities for future work, but exploring probabilistic notions of oracles and concepts of soundness and completeness of oracles. Because oracles (and their approximations) are typically computationally expensive, an approach to the provision of oracle information may use a probabilistic approach even where an complete and precise answer is possible.

Recall that a definite oracle responds with either a 1 or a 0 indicating that the test activity sequence is acceptable or unacceptable respectively, for each sequence of test activities (i.e., stimuli and observations). It may also be useful to generalise this definition to allow for probabilistic oracles:

Definition 8 (Probabilistic Oracle): A *Probabilistic Oracle* $\tilde{\mathbb{D}}$ is a function from a test activity sequence to $[0, 1]$, i.e., $\tilde{\mathbb{D}} : \mathbb{I} \rightarrow [0, 1]$.

A probabilistic oracle returns a real number in the closed interval $[0, 1]$, denoting a less precise response than a definite oracle. As with definite oracles, we do not require a probabilistic oracle to be a total function. A probabilistic oracle can be used to model the case where the oracle is only able to offer a probability that the test case is acceptable, or for other situations where some degree of imprecision is to be tolerated in the oracle's response.

Since a definite oracle is merely a special case of a probabilistic oracle, we shall use the term oracle, hereinafter, to refer to either definite or probabilistic oracles where the terminology applies to both.

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

Definition 9 (Soundness): A *Probabilistic Oracle*, $\tilde{\mathbb{D}}$ is sound iff

$$\tilde{\mathbb{D}}\sigma \in \begin{cases} [0, 0.5) & \text{when } \mathcal{G}\sigma = 0 \\ [0.5, 1] & \text{when } \mathcal{G}\sigma = 1 \end{cases}$$

Notice that the constant oracle $\lambda(i).\frac{1}{2}$ is vacuously sound (but offers no information). Also, notice that Definition 9 above, specialises by instantiation to the case of a definite oracle, such that a definite oracle is sound iff $DO \subseteq \mathcal{G}$.

The definitions of completeness and correctness already defined for definite oracles in Section 2 simply carry over to probabilistic oracles as defined above without modification.

8.5 Measurement of Oracles and Their Properties

There is work on using oracle as the measure of how well the program has been tested (a kind of oracle coverage), [184], [102], [173] and other measurements regarding oracles in general such as to assess the quality of assertions [156].

However, more work is required on the measurement of oracles and their properties. Perhaps this denotes a challenge to the "software metrics" community to consider "oracle metrics". In a world in which oracles become more prevalent, it will be important for testers to be able to assess the features offered by different alternative oracles.

It is also important to understand (and *measure*) the interplay between oracles and testing. That is, if testing is poor at revealing faults this may be due to a poor oracle that only partly covers the system's intended behaviour. Alternatively, it might be that the oracle achieves higher coverage, but with great inaccuracy relative to the ground truth, thereby misleading the testing process. We therefore need ways to measure the coverage (and "quality") of overall behaviour captured by an oracle.

8.6 What if There Simply is No Automated Oracle?

A recent area of growth has been handling the lack of an automated oracle, by *reducing* the number of test cases, and *easing* test case evaluation.

In terms of test suite reduction, further research is required to establish which techniques are the best in reducing test suite size whilst obtaining high levels of a test adequacy criterion. Work to date has largely concentrated on heuristics without a full understanding of how adequacy criteria relate to particular programs. For example, optimal minimisation of a test suite for branch coverage involves knowledge about which branches are executed by the same inputs – i.e. how so-called "collateral coverage" [82] can be maximised. Research to date has largely concentrated only on reducing test suite size with respect to branch coverage, where high branch coverage is taken to mean that the test suite will be good at detecting faults. Instead of branch coverage, some other test adequacy criterion could be tested, for example the ability of a test suite to kill as many mutants as possible in as few test cases as possible—and whether it is feasible to produce smaller test suites for mutation testing as opposed to structural coverage.

Further in test case reduction, there has been no work on "reducing" test input values. For example, test values such as "0.0" convey much less information than a long double value such as "0.15625", and should be preferred for use in test case generation. Moreover, there has been no work on the converse problem of outputs. In order to reduce manual checking time, short, readable outputs are also required. This will produce further reductions in terms of the checking load placed on a human oracle. Much of the techniques have also concentrated solely on test suites produced for structural coverage, ignoring the numerous other ways in which to test a program.

Easing the test burden is a qualitative problem that involves reducing the time to understand test case scenarios and evaluate corresponding program outputs. In this area, there has not been much work on improving the realism of automatically generated test cases. With the exception of Afshan et al. [1], very little of this work, however, has been evaluated using humans. Further work needs to develop a deeper understanding of the cognitive and psychological aspects behind test case evaluation in order to further reduce oracle cost.

9 Conclusions

This paper has provided a comprehensive survey of oracles in software testing, covering specified, derived and implicit oracles and techniques that cater for the absence of oracles. The paper also provided an analysis of trends in the literature and sets out a road map for future work at the interfaces between existing definitions (such as metamorphic and regression-based oracles). The paper is accompanied by a repository of papers on oracles which we make available to the community[8].

10 Acknowledgements

We would like to thank Bob Binder for helpful information and discussions when we began work on this paper. We would also like thank all who attended the CREST Open Workshop on the Oracle Problem (21–22 May 2012) at University College London, and gave feedback on an early presentation of the work. We are further indebted to the very many responses to our emails from authors cited in this survey, who provided several useful comments on an earlier draft of our paper.

8. Repository will be made public if/when paper is accepted

References

[1] Sheeva Afshan, Phil McMinn, and Mark Stevenson. Evolving readable string test inputs using a natural language model to reduce human oracle cost. In *International Conference on Software Testing, Verification and Validation (ICST 2013)*. IEEE, March 2013.

[2] Wasif Afzal, Richard Torkar, and Robert Feldt. A systematic review of search-based testing for non-functional system properties. *Information and Software Technology*, 51(6):957–976, 2009.

[3] Bernhard K. Aichernig. Automated black-box testing with abstract VDM oracles. In *SAFECOMP*, pages 250–259. Springer-Verlag, 1999.

[4] Shaukat Ali, Lionel C. Briand, Hadi Hemmati, and Rajwinder Kaur Panesar-Walawege. A systematic review of the application and empirical investigation of search-based test-case generation. *IEEE Transactions on Software Engineering*, pages 742–762, 2010.

[5] Nadia Alshahwan and Mark Harman. Automated session data repair for web application regression testing. In *Proceedings of 2008 International Conference on Software Testing, Verification, and Validation*, pages 298–307. IEEE Computer Society, 2008.

[6] Nadia Alshahwan and Mark Harman. Automated web application testing using search based software engineering. In *ASE*, pages 3–12, 2011.

[7] Dana Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):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 *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 786–795, 2011.

[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 *Empirical Software Engineering and Measurement (ESEM), 2011 International Symposium on*, pages 10–19, 2011.

[10] Andrea Arcuri and Xin Yao. Search based software testing of object-oriented containers. *Information Sciences*, 178(15):3075–3095, 2008.

[11] Shay Artzi, Michael D. Ernst, Adam Kieżun, Carlos Pacheco, and Jeff H. Perkins. Finding the needles in the haystack: Generating legal test inputs for object-oriented programs. In *1st Workshop on Model-Based Testing and Object-Oriented Systems (M-TOOS)*, Portland, OR, October 23, 2006.

[12] Egidio Astesiano, Michel Bidoit, Hélène Kirchner, Bernd Krieg-Brückner, Peter D. Mosses, Donald Sannella, and Andrzej Tarlecki. CASL: the common algebraic specification language. *Theor. Comput. Sci.*, 286(2):153–196, 2002.

[13] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *PLDI*, pages 290–301. ACM, 1994.

[14] A. Avizienis. The N-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, 11:1491–1501, 1985.

[15] A. Avizienis and L. Chen. On the implementation of N-version programming for software fault-tolerance during execution. In *Proceedings of the First International Computer Software and Application Conference (COMPSAC '77)*, pages 149–155, 1977.

[16] Arthur Baars, Mark Harman, Youssef Hassoun, Kiran Lakhotia, Phil McMinn, Paolo Tonella, and Tanja Vos. Symbolic search-based testing. In *ASE*, pages 53–62, 2011.

[17] A. F. Babich. Proving total correctness of parallel programs. *IEEE Trans. Softw. Eng.*, 5(6):558–574, November 1979.

[18] Luciano Baresi and Michal Young. Test oracles. Technical Report CIS-TR-01-02, University of Oregon, Dept. of Computer and Information Science, August 2001. http://www.cs.uoregon.edu/~michal/pubs/oracles.html.

[19] Sofia Bekrar, Chaouki Bekrar, Roland Groz, and Laurent Mounier. Finding software vulnerabilities by smart fuzzing. In *ICST*, pages 427–430, 2011.

[20] Gilles Bernot. Testing against formal specifications: a theoretical view. In *Proceedings of the International Joint Conference on Theory and Practice of Software Development on Advances in Distributed Computing (ADC) and Colloquium on Combining Paradigms for Software Development (CCPSD): Vol. 2*, TAPSOFT '91, pages 99–119, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[21] Gilles Bernot, Marie Claude Gaudel, and Bruno Marre. Software testing based on formal specifications: a theory and a tool. *Softw. Eng. J.*, 6(6):387–405, November 1991.

[22] Antonia Bertolino, Paola Inverardi, Patrizio Pelliccione, and Massimo Tivoli. Automatic synthesis of behavior protocols for composable web-services. In *Proceedings of ESEC/SIGSOFT FSE*, ESEC/FSE 2009, pages 141–150, 2009.

[23] D. Beyer, T. Henzinger, R. Jhala, and R. Majumdar. Checking memory safety with Blast. In M. Cerioli, editor, *Fundamental Approaches to Software Engineering, 8th International Conference, FASE 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3442 of *Lecture Notes in Computer Science*, pages 2–18. Springer, 2005.

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

[25] Christian Bird and Thomas Zimmermann. Assessing the value of branches with what-if analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 45:1–45:11. ACM, 2012.

[26] A. Blikle. Proving programs by sets of computations. In *Mathematical Foundations of Computer Science*, pages 333–358. Springer, 1975.

[27] Gregor V. Bochmann and Alexandre Petrenko. Protocol testing: review of methods and relevance for software testing. In *Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '94, pages 109–124. ACM, 1994.

[28] G.V. Bochmann. Finite state description of communication protocols. *Computer Networks*, 2(4):361–372, 1978.

[29] E. Börger, A. Cavarra, and E. Riccobene. Modeling the dynamics of UML state machines. In *Abstract State Machines-Theory and Applications*, pages 167–186. Springer, 2000.

[30] Egon Börger. High level system design and analysis using abstract state machines. In *Proceedings of the International Workshop on Current Trends in Applied Formal Method: Applied Formal Methods*, FM-Trends 98, pages 1–43, London, UK, UK, 1999. Springer-Verlag.

[31] Kathrin Böttger, Rolf Schwitter, Diego Mollá, and Debbie Richards. Towards reconciling use cases via controlled language and graphical models. In *INAP*, pages 115–128, Berlin, Heidelberg, 2003. Springer-Verlag.

[32] F. Bouquet, C. Grandpierre, B. Legeard, F. Peureux, N. Vacelet, and M. Utting. A subset of precise UML for model-based testing. In *Proceedings of the 3rd International Workshop on Advances in Model-Based Testing*, A-MOST '07, pages 95–104, New York, NY, USA, 2007. ACM.

[33] Mustafa Bozkurt and Mark Harman. Automatically generating realistic test input from web services. In *IEEE 6th International Symposium on Service Oriented System Engineering (SOSE)*, pages 13–24, 2011.

[34] L. C. Briand, Y. Labiche, and H. Sun. Investigating the use of analysis contracts to improve the testability of object-oriented code. *Softw. Pract. Exper.*, 33(7):637–672, June 2003.

[35] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically generating inputs of death. *ACM Trans. Inf. Syst. Secur.*, 12(2):10:1–10:38, December 2008.

[36] J. Callahan, F. Schneider, S. Easterbrook, et al. Automated software testing using model-checking. In *Proceedings 1996 SPIN workshop*, volume 353. Citeseer, 1996.

[37] F. T. Chan, T. Y. Chen, S. C. Cheung, M. F. Lau, and S. M. Yiu. Application of metamorphic testing in numerical analysis. In *Proceedings of the IASTED International Conference on Software Engineering*, pages 191–197, 1998.

[38] W.K. Chan, T.Y. Chen, Heng Lu, T.H. Tse, and Stephen S. Yau. Integration testing of context-sensitive middleware-based applications: a metamorphic approach. *Special Issue on Quality Software (for QSIC 2005) of International Journal of Software*

Engineering and Knowledge Engineering, 16(5):677–703, 2006.

[39] W.K. Chan, S.C. Cheung, and Karl R.P.H. Leung. *A metamorphic testing approach for online testing of service-oriented software applications*, chapter 7, pages 2894–2914. IGI Global, 2009.

[40] W.K. Chan, S.C. Cheung, and K.R.P.H. Leung. Towards a metamorphic testing methodology for service-oriented software applications. In *QSIC*, pages 470–476, September 2005.

[41] F. Chen, N. Tillmann, and W. Schulte. Discovering specifications. Technical Report MSR-TR-2005-146, Microsoft Research, October 2005.

[42] Huo Yan 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.*, 7:250–295, July 1998.

[43] Huo Yan 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.*, 10(1):56–109, January 2001.

[44] T. Y. Chen, F.-C. Kuo, T. H. Tse, and Zhi Quan Zhou. Metamorphic testing and beyond. In *Proceedings of the International Workshop on Software Technology and Engineering Practice (STEP 2003)*, pages 94–100, September 2004.

[45] J. Cheng. A survey of tasking deadlock detection methods. *ACM SIGAda Ada Letters*, 11(1):82–91, 1991.

[46] Yoonsik Cheon. Abstraction in assertion-based test oracles. In *Proceedings of the Seventh International Conference on Quality Software*, pages 410–414, Washington, DC, USA, 2007. IEEE Computer Society.

[47] Yoonsik Cheon and Gary T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, ECOOP '02, pages 231–255, London, UK, 2002. Springer-Verlag.

[48] L.A. Clarke. A system to generate test data and symbolically execute programs. *Software Engineering, IEEE Transactions on*, SE-2(3):215 – 222, sept. 1976.

[49] Paul C. Clements. Managing variability for software product lines: Working with variability mechanisms. In 10^{th} *International Conference on Software Product Lines (SPLC 2006)*, pages 207–208, Baltimore, Maryland, USA, 2006. IEEE Computer Society.

[50] Markus Clermont and David Parnas. Using information about functions in selecting test cases. In *Proceedings of the 1st international workshop on Advances in model-based testing*, A-MOST '05, pages 1–7, New York, NY, USA, 2005. ACM.

[51] David Coppit and Jennifer M. Haddox-Schatz. On the use of specification-based assertions as test oracles. In *Proceedings of the 29th Annual IEEE/NASA on Software Engineering Workshop*, pages 305–314, Washington, DC, USA, 2005. IEEE Computer Society.

[52] M. Davies and E. Weyuker. Pseudo-oracles for non-testable programs. In *Proceedings of the ACM '81 Conference*, pages 254–257, 1981.

[53] Karnig Derderian, Robert Hierons, Mark Harman, and Qiang Guo. Automated Unique Input Output sequence generation for conformance testing of FSMs. *The Computer Journal*, 49(3):331–344, 2006.

[54] Laura K. Dillon. Automated support for testing and debugging of real-time programs using oracles. *SIGSOFT Softw. Eng. Notes*, 25(1):45–46, January 2000.

[55] Roong-Ko Doong and Phyllis G. Frankl. The ASTOOT approach to testing object-oriented programs. *ACM Trans. Softw. Eng. Methodol.*, 3:101–130, April 1994.

[56] Edith Elkind, Blaise Genest, Doron Peled, and Hongyang Qu. Grey-box checking. In *FORTE*, pages 420–435, 2006.

[57] J. Ellsberger, D. Hogrefe, and A. Sarma. *SDL: formal object-oriented language for communicating systems*. Prentice Hall, 1997.

[58] 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. *Science of Computer Programming*, 69(1):35–45, 2007.

[59] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Software Eng.*, 27(2):99–123, 2001.

[60] R.A. Eyre-Todd. The detection of dangling references in C++ programs. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 2(1-4):127–134, 1993.

[61] R. Feldt. Generating diverse software versions with genetic programming: an experimental study. *Software, IEE Proceedings*, 145, December 1998.

[62] Xin Feng, David Lorge Parnas, T. H. Tse, and Tony O'Callaghan. A comparison of tabular expression-based testing strategies. *IEEE Trans. Softw. Eng.*, 37(5):616–634, September 2011.

[63] Xin Feng, David Lorge Parnas, and T.H. Tse. Tabular expression-based testing strategies: A comparison. *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, 0:134, 2007.

[64] Roger Ferguson and Bogdan Korel. The chaining approach for software test data generation. *ACM Transactions on Software Engineering and Methodology*, 5(1):63–86, January 1996.

[65] J. Ferrer, F. Chicano, and E. Alba. Evolutionary algorithms for the multi-objective test data generation problem. *Software: Practice and Experience*, 42(11):1331–1362, 2011.

[66] John S. Fitzgerald and Peter Gorm Larsen. *Modelling Systems - Practical Tools and Techniques in Software Development (2. ed.)*. Cambridge University Press, 2009.

[67] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. *Lecture Notes in Computer Science*, 2021:500–517, 2001.

[68] Robert W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Symposia in Applied Mathematics*, pages 19–32. American Mathematical Society, Providence, RI, 1967.

[69] Gordon Fraser and Andrea Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276–291, 2013.

[70] Gordon Fraser and Andreas Zeller. Exploiting common object usage in test case generation. In *Proceedings of the 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, ICST '11, pages 80–89. IEEE Computer Society, 2011.

[71] Kambiz Frounchi, Lionel C. Briand, Leo Grady, Yvan Labiche, and Rajesh Subramanyan. Automating image segmentation verification and validation by learning test oracles. *Information and Software Technology*, 53(12):1337–1348, 2011.

[72] J. Gannon, P. McMullin, and R. Hamlet. Data abstraction, implementation, specification, and testing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 3(3):211–223, 1981.

[73] Angelo Gargantini and Elvinia Riccobene. ASM-based testing: Coverage criteria and automatic test sequence. *Journal of Universal Computer Science*, 7(11):1050–1067, nov 2001.

[74] Stephen J. Garland, John V. Guttag, and James J. Horning. An overview of Larch. In *Functional Programming, Concurrency, Simulation and Automated Reasoning*, pages 329–348, 1993.

[75] Marie-Claude Gaudel. Testing from formal specifications, a generic approach. In *Proceedings of the 6th Ade-Europe International Conference Leuven on Reliable Software Technologies*, Ada Europe '01, pages 35–48, London, UK, 2001. Springer-Verlag.

[76] Marie-Claude Gaudel and Perry R. James. Testing algebraic data types and processes: A unifying theory. *Formal Asp. Comput.*, 10(5-6):436–451, 1998.

[77] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *PLDI*, pages 213–223. ACM, 2005.

[78] Alex Groce, Doron Peled, and Mihalis Yannakakis. Amc: An adaptive model checker. In *CAV*, pages 521–525, 2002.

[79] Roland Groz, Keqin Li, Alexandre Petrenko, and Muzammil Shahbaz. Modular system verification by inference, testing and reachability analysis. In *TestCom/FATES*, pages 216–233, 2008.

[80] R. Guderlei and J. Mayer. Statistical metamorphic testing testing programs with random output by means of statistical hypothesis tests and metamorphic testing. In *QSIC*, pages 404–409, October 2007.

[81] D. Harel. Statecharts: A visual formalism for complex systems. *Science of computer programming*, 8(3):231–274, 1987.

[82] Mark Harman, Sung Gon Kim, Kiran Lakhotia, Phil McMinn, and Shin Yoo. Optimizing for the number of tests generated in search based test data generation with an application to the oracle cost problem. In *International Workshop on Search-Based Software Testing (SBST 2010)*, pages 182–191. IEEE, 6 April 2010.

[83] Mark Harman, Afshin Mansouri, and Yuanyuan Zhang. Search based software engineering: A comprehensive analysis and review of trends techniques and applications. Technical Report TR-09-03, Department of Computer Science, King's College London, April 2009.

[84] M. Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. A methodology for controlling the size of a test suite. *ACM Trans. Softw. Eng. Methodol.*, 2(3):270–285, July 1993.

[85] Mary Jean Harrold, Gregg Rothermel, Kent Sayre, Rui Wu, and Liu Yi. An empirical investigation of the relationship between spectra differences and regression faults. *Software Testing, Verification, and Reliability*, 10(3):171–194, 2000.

[86] David L. Heine and Monica S. Lam. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In *PLDI*, pages 168–181. ACM, 2003.

[87] J. Henkel and A. Diwan. Discovering algebraic specifications from Java classes. *Lecture Notes in Computer Science*, 2743:431–456, 2003.

[88] F.C. Hennie. *Finite-state models for logical machines*. Wiley, 1968.

[89] MarijnJ.H. Heule and Sicco Verwer. Software model synthesis using satisfiability solvers. *Empirical Software Engineering*, pages 1–32, 2012.

[90] R.M. Hierons. Oracles for distributed testing. *Software Engineering, IEEE Transactions on*, 38(3):629–641, 2012.

[91] Robert M. Hierons. Verdict functions in testing with a fault domain or test hypotheses. *ACM Trans. Softw. Eng. Methodol.*, 18(4):14:1–14:19, July 2009.

[92] Robert M. Hierons, Kirill Bogdanov, Jonathan P. Bowen, Rance Cleaveland, John Derrick, Jeremy Dick, Marian Gheorghe, Mark Harman, Kalpesh Kapoor, Paul Krause, Gerald Lüttgen, Anthony J. H. Simons, Sergiy Vilkomir, Martin R. Woodward, and Hussein Zedan. Using formal specifications to support testing. *ACM Comput. Surv.*, 41:9:1–9:76, February 2009.

[93] Charles Anthony Richard Hoare. An Axiomatic Basis of Computer Programming. *Communications of the ACM*, 12:576–580, 1969.

[94] M. Holcombe. X-machines as a basis for dynamic system specification. *Software Engineering Journal*, 3(2):69–76, 1988.

[95] Mike Holcombe and Florentin Ipate. Correct systems: building a business process solution. *Software Testing Verification and Reliability*, 9(1):76–77, 1999.

[96] Gerard J. Holzmann. The model checker SPIN. *IEEE Trans. Softw. Eng.*, 23(5):279–295, May 1997.

[97] W E Howden. A functional approach to program testing and analysis. *IEEE Trans. Softw. Eng.*, 12(10):997–1005, October 1986.

[98] W.E. Howden. Theoretical and empirical studies of program testing. *IEEE Transactions on Software Engineering*, 4(4):293–298, July 1978.

[99] Merlin Hughes and David Stotts. Daistish: systematic algebraic testing for OO programs in the presence of side-effects. *SIGSOFT Softw. Eng. Notes*, 21(3):53–61, May 1996.

[100] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[101] Claude Jard and Gregor v. Bochmann. An approach to testing specifications. *Journal of Systems and Software*, 3(4):315 – 323, 1983.

[102] D. Jeffrey and R. Gupta. Test case prioritization using relevant slices. In *Computer Software and Appli-*

cations Conference, 2006. COMPSAC '06. 30th Annual International, volume 1, pages 411–420, 2006.

[103] Ying Jin and David Lorge Parnas. Defining the meaning of tabular mathematical expressions. *Sci. Comput. Program.*, 75(11):980–1000, November 2010.

[104] M.J. Kearns and U.V. Vazirani. *An Introduction to Computational Learning Theory*. MIT Press, 1994.

[105] R.M. Keller. Formal verification of parallel programs. *Communications of the ACM*, 19(7):371–384, 1976.

[106] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[107] N.P. Kropp, P.J. Koopman, and D.P. Siewiorek. Automated robustness testing of off-the-shelf software components. In *Fault-Tolerant Computing, 1998. Digest of Papers. Twenty-Eighth Annual International Symposium on*, pages 230–239. IEEE, 1998.

[108] Kiran Lakhotia, Phil McMinn, and Mark Harman. An empirical investigation into branch coverage for C programs using CUTE and AUSTIN. *Journal of Systems and Software*, 83(12):2379–2391, 2010.

[109] Axel van Lamsweerde. Formal specification: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 147–159, New York, NY, USA, 2000. ACM.

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

[111] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines—a survey. *Proceedings of the IEEE*, 84(8):1090–1123, aug 1996.

[112] A. Leitner, M. Oriol, A Zeller, I. Ciupa, and B. Meyer. Efficient unit test case minimization. In *Automated Software Engineering (ASE 2007)*, pages 417–420, Atlanta, Georgia, USA, 2007. ACM Press.

[113] Keqin Li, Roland Groz, and Muzammil Shahbaz. Integration testing of components guided by incremental state machine learning. In *TAIC PART*, pages 59–70, 2006.

[114] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *proceedings of the 30th International Conference on Software Engineering (ICSE)*, 2008.

[115] D. Luckham and F.W. Henke. An overview of ANNA-a specification language for ADA. Technical report, Stanford University, 1984.

[116] G.R. Luecke, J. Coyle, J. Hoekstra, M. Kraeva, Y. Li, O. Taborskaia, and Y. Wang. A survey of systems for detecting serial run-time errors. *Concurrency and Computation: Practice and Experience*, 18(15):1885–1907, 2006.

[117] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2:219–246, 1989.

[118] Patrícia D. L. Machado. On oracles for interpreting test results against algebraic specifications. In *AMAST*, pages 502–518. Springer-Verlag, 1999.

[119] Phil Maker. GNU Nana: improved support for assertion checking and logging in GNU C/C++, 1998. http://gnu.cs.pu.edu.tw/software/nana/.

[120] Haroon Malik, Hadi Hemmati, and Ahmed E. Hassan. Automatic detection of performance deviations in the load testing of large scale systems. In *Proceedings of International Conference on Software Engineering - Software Engineering in Practice Track*, ICSE 2013, page *to appear*, 2013.

[121] L. Mariani, F. Pastore, and M. Pezzè. Dynamic analysis for diagnosing integration faults. *IEEE Transactions on Software Engineering*, 37(4):486–508, 2011.

[122] Bruno Marre. Loft: A tool for assisting selection of test data sets from algebraic specifications. In *TAPSOFT*, pages 799–800. Springer-Verlag, 1995.

[123] A.P. Mathur. Performance, effectiveness, and reliability issues in software testing. In *COMPSAC*, pages 604–605. IEEE, 1991.

[124] P. McMinn, M. Stevenson, and M. Harman. Reducing qualitative human oracle costs associated with automatically generated test data. In *1st International Workshop on Software Test Output Validation (STOV 2010), Trento, Italy, 13th July 2010*, pages 1–4, 2010.

[125] Phil McMinn. Search-based software test data generation: a survey. *Softw. Test. Verif. Reliab.*, 14(2):105–156, June 2004.

[126] Phil McMinn. Search-based failure discovery using testability transformations to generate pseudo-oracles. In *Genetic and Evolutionary Computation Conference (GECCO 2009)*, pages 1689–1696. ACM Press, 8-12 July 2009.

[127] Phil McMinn. Search-based software testing: Past, present and future. In *International Workshop on Search-Based Software Testing (SBST 2011)*, pages 153–163. IEEE, 21 March 2011.

[128] Phil McMinn, Muzammil Shahbaz, and Mark Stevenson. Search-based test input generation for string data types using the results of web queries. In *ICST*, pages 141–150, 2012.

[129] Phil McMinn, Mark Stevenson, and Mark Harman. Reducing qualitative human oracle costs associated with automatically generated test data. In *International Workshop on Software Test Output Validation (STOV 2010)*, pages 1–4. ACM, 13 July 2010.

[130] Atif M. Memon. Automatically repairing event sequence-based GUI test suites for regression testing. *ACM Transactions on Software Engineering Methodology*, 18(2):1–36, 2008.

[131] Atif M. Memon, Martha E. Pollack, and Mary Lou Soffa. Automated test oracles for GUIs. In *SIGSOFT '00/FSE-8: Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 30–39, New York, NY, USA, 2000. ACM Press.

[132] Atif M. Memon and Qing Xie. Using transient/persistent errors to develop automated test oracles for event-driven software. In *ASE '04: Proceedings of the 19th IEEE international conference on Automated software engineering*, pages 186–195, Washington, DC, USA, 2004. IEEE Computer Society.

[133] Maik Merten, Falk Howar, Bernhard Steffen, Patrizio Pellicione, and Massimo Tivoli. Automated inference of models for black box systems based

on interface descriptions. In *Proceedings of the 5th international conference on Leveraging Applications of Formal Methods, Verification and Validation: technologies for mastering change - Volume Part I*, ISoLA'12, pages 79–96. Springer-Verlag, 2012.

[134] Bertrand Meyer. Eiffel: A language and environment for software engineering. *The Journal of Systems and Software*, 8(3):199–246, June 1988.

[135] B.P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[136] 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. *Software Engineering, IEEE Transactions on*, 37(2):161–187, 2011.

[137] Christian Murphy, Kuang Shen, and Gail Kaiser. Automatic system testing of programs without test oracles. In *ISSTA*, pages 189–200. ACM Press, 2009.

[138] Christian Murphy, Kuang Shen, and Gail Kaiser. Using JML runtime assertion checking to automate metamorphic testing in applications without test oracles. *2009 International Conference on Software Testing Verification and Validation*, pages 436–445, 2009.

[139] A. J. Offutt, J. Pan, and J. M. Voas. Procedures for reducing the size of coverage-based test sets. In *International Conference on Testing Computer Software*, pages 111–123, 1995.

[140] C. Pacheco and M. Ernst. Eclat: Automatic generation and classification of test inputs. *ECOOP 2005-Object-Oriented Programming*, pages 734–734, 2005.

[141] D L Parnas, J Madey, and M Iglewski. Precise documentation of well-structured programs. *IEEE Transactions on Software Engineering*, 20(12):948–976, 1994.

[142] David Lorge Parnas. Document based rational software development. *Journal of Knowledge Based Systems*, 22:132–141, April 2009.

[143] David Lorge Parnas. Precise documentation: The key to better software. In Sebastian Nanz, editor, *The Future of Software Engineering*, pages 125–148. Springer Berlin Heidelberg, 2011.

[144] David Lorge Parnas and Jan Madey. Functional documents for computer systems. *Sci. Comput. Program.*, 25(1):41–61, October 1995.

[145] F. Pastore, L. Mariani, and G. Fraser. Crowdoracles: Can the crowd solve the oracle problem? In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, 2013.

[146] Doron Peled, Moshe Y. Vardi, and Mihalis Yannakakis. Black box checking. *Journal of Automata, Languages and Combinatorics*, 7(2), 2002.

[147] D K Peters and D L Parnas. Using test oracles generated from program documentation. *IEEE Transactions on Software Engineering*, 24(3):161–173, 1998.

[148] Dennis K. Peters and David Lorge Parnas. Requirements-based monitors for real-time systems. *IEEE Trans. Softw. Eng.*, 28(2):146–158, February 2002.

[149] Nadia Polikarpova, Ilinca Ciupa, and Bertrand Meyer. A comparative study of programmer-written and automatically inferred contracts. In *ISSTA*, pages 93–104. ACM, 2009.

[150] S.J. Prowell and J.H. Poore. Foundations of sequence-based software specification. *Software Engineering, IEEE Transactions on*, 29(5):417–429, 2003.

[151] Sam Ratcliff, David R. White, and John A. Clark. Searching for invariants using genetic programming and mutation testing. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, GECCO '11, pages 1907–1914, New York, NY, USA, 2011. ACM.

[152] F. Ricca and P. Tonella. Detecting anomaly and failure in web applications. *MultiMedia, IEEE*, 13(2):44 – 51, april-june 2006.

[153] D.S. Rosenblum. A practical approach to programming with assertions. *Software Engineering, IEEE Transactions on*, 21(1):19–31, 1995.

[154] G. Rothermel, M. J. Harrold, J. von Ronne, and C. Hong. Empirical studies of test-suite reduction. *Software Testing, Verification and Reliability*, 12:219—249, 2002.

[155] N. Walkinshaw S. Ali, K. Bogdanov. A comparative study of methods for dynamic reverse-engineering of state models. Technical Report CS-07-16, The University of Sheffield, Department of Computer Science, October 2007. http://www.dcs.shef.ac.uk/intranet/research/resmes/CS0716.pdf.

[156] David Schuler and Andreas Zeller. Assessing oracle quality with checked coverage. In *Proceedings of the 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, ICST '11, pages 90–99, Washington, DC, USA, 2011. IEEE Computer Society.

[157] R. Schwitter. English as a formal specification language. In *Database and Expert Systems Applications, 2002. Proceedings. 13th International Workshop on*, pages 228–232, sept. 2002.

[158] Sergio Segura, Robert M. Hierons, David Benavides, and Antonio Ruiz-Cortés. Automated metamorphic testing on the analyses of feature models. *Information and Software Technology*, 53(3):245 – 258, 2011.

[159] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In *ESEC/FSE*, pages 263–272. ACM, 2005.

[160] Seyed Shahamiri, Wan Wan-Kadir, Suhaimi Ibrahim, and Siti Hashim. Artificial neural networks as multi-networks automated test oracle. *Automated Software Engineering*, 19(3):303–334, 2012.

[161] Seyed Reza Shahamiri, Wan Mohd Nasir Wan Kadir, Suhaimi Ibrahim, and Siti Zaiton Mohd Hashim. An automated framework for software test oracle. *Information and Software Technology*, 53(7):774 – 788, 2011.

[162] Seyed Reza Shahamiri, Wan Mohd Nasir Wan-Kadir, and Siti Zaiton Mohd Hashim. A comparative study on automated software test oracle methods. In *ICSEA*, pages 140–145, 2009.

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

[164] K. Shrestha and M.J. Rutherford. An empirical

evaluation of assertions as oracles. In *ICST*, pages 110–119. IEEE, 2011.

[165] Guoqiang Shu, Yating Hsu, and David Lee. Detecting communication protocol security flaws by formal fuzz testing and machine learning. In *FORTE*, pages 299–304, 2008.

[166] J. Sifakis. Deadlocks and livelocks in transition systems. *Mathematical Foundations of Computer Science 1980*, pages 587–600, 1980.

[167] A. J. H. Simons. JWalk: a tool for lazy systematic testing of java classes by design introspection and user interaction. *Automated Software Engineering*, 14(4):369–418, December 2007.

[168] Rishabh Singh, Dimitra Giannakopoulou, and Corina S. Pasareanu. Learning component interfaces with may and must abstractions. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, volume 6174 of *Lecture Notes in Computer Science*, pages 527–542. Springer, 2010.

[169] J. Michael Spivey. *Z Notation - a reference manual (2. ed.)*. Prentice Hall International Series in Computer Science. Prentice Hall, 1992.

[170] 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 *Proceedings of the 34th International Conference on Software Engineering*, ICSE 2012, pages 870–880, 2012.

[171] M. Staats, M.W. Whalen, and M.P.E. Heimdahl. Programs, tests, and oracles: the foundations of testing revisited. In *ICSE*, pages 391–400. IEEE, 2011.

[172] Matt Staats, Shin Hong, Moonzoo Kim, and Gregg Rothermel. Understanding user understanding: determining correctness of generated program invariants. In *ISSTA*, pages 188–198. ACM, 2012.

[173] Matt Staats, Pablo Loyola, and Gregg Rothermel. Oracle-centric test case prioritization. In *Proceedings of the 23rd International Symposium on Software Reliability Engineering*, ISSRE 2012, pages 311–320, 2012.

[174] Ari Takanen, Jared DeMott, and Charlie Miller. *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, Inc., Norwood, MA, USA, 1 edition, 2008.

[175] Ramsay Taylor, Mathew Hall, Kirill Bogdanov, and John Derrick. Using behaviour inference to optimise regression test sets. In Brian Nielsen and Carsten Weise, editors, *Testing Software and Systems - 24th IFIP WG 6.1 International Conference, ICTSS 2012, Aalborg, Denmark, November 19-21, 2012. Proceedings*, volume 7641 of *Lecture Notes in Computer Science*, pages 184–199. Springer, 2012.

[176] A. Tiwari. Formal semantics and analysis methods for Simulink Stateflow models. Technical report, SRI International, 2002. http://www.csl.sri.com/users/tiwari/html/stateflow.html.

[177] Paolo Tonella. Evolutionary testing of classes. In *ISSTA*, pages 119–128, Boston, Massachusetts, USA, 11-14 July 2004. ACM.

[178] Jan Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software - Concepts and Tools*, 17(3):103–120, 1996.

[179] Alan M. Turing. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69, Cambridge, England, June 1949. University Mathematical Laboratory.

[180] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[181] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. *Softw. Test. Verif. Reliab.*, 22(5):297–312, August 2012.

[182] G. v. Bochmann, C. He, and D. Ouimet. Protocol testing using automatic trace analysis. In *Proceedings of Canadian Conference on Electrical and Computer Engineering*, pages 814–820, 1989.

[183] A. van Lamsweerde and M. Sintzoff. Formal derivation of strongly correct concurrent programs. *Acta Informatica*, 12(1):1–31, 1979.

[184] J.M. Voas. PIE: a dynamic failure-based technique. *Software Engineering, IEEE Transactions on*, 18(8):717–727, 1992.

[185] N. Walkinshaw, K. Bogdanov, J. Derrick, and J. Paris. Increasing functional coverage by inductive testing: A case study. In Alexandre Petrenko, Adenilso da Silva Simão, and José Carlos Maldonado, editors, *Testing Software and Systems - 22nd IFIP WG 6.1 International Conference, ICTSS 2010, Natal, Brazil, November 8-10, 2010. Proceedings*, volume 6435 of *Lecture Notes in Computer Science*, pages 126–141. Springer, 2010.

[186] Neil Walkinshaw and Kirill Bogdanov. Inferring finite-state models with temporal constraints. In *ASE*, pages 248–257, 2008.

[187] Neil Walkinshaw, John Derrick, and Qiang Guo. Iterative refinement of reverse-engineered models by model-based testing. In *FM*, pages 305–320, 2009.

[188] Yabo Wang and DavidLorge Parnas. Trace rewriting systems. In Michaël Rusinowitch and Jean-Luc Rémy, editors, *Conditional Term Rewriting Systems*, volume 656 of *Lecture Notes in Computer Science*, pages 343–356. Springer Berlin Heidelberg, 1993.

[189] Yabo Wang and D.L. Parnas. Simulating the behaviour of software modules by trace rewriting. In *Software Engineering, 1993. Proceedings., 15th International Conference on*, pages 14–23, 1993.

[190] Yi Wei, Carlo A. Furia, Nikolay Kazmin, and Bertrand Meyer. Inferring better contracts. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 191–200, New York, NY, USA, 2011. ACM.

[191] Yi Wei, H. Roth, C.A. Furia, Yu Pei, A. Horton, M. Steindorfer, M. Nordio, and B. Meyer. Stateful testing: Finding more errors in code and contracts. In *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, pages 440–443, 2011.

[192] EJ Weyuker. Translatability and decidability questions for restricted classes of program schemas. *SIAM Journal on Computing*, 8(4):587–598, 1979.

[193] E.J. Weyuker. Assessing test data adequacy through program inference. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(4):641–655, 1983.

[194] E.J. Weyuker and F.I. Vokolos. Experience with performance testing of software systems: issues, an approach, and case study. *Software Engineering, IEEE Transactions on*, 26(12):1147–1156, 2000.

[195] Elaine J. Weyuker. On testing non-testable programs. *The Computer Journal*, 25(4):465–470, November 1982.

[196] Jeannette M. Wing. A specifier's introduction to formal methods. *IEEE Computer*, 23(9):8–24, 1990.

[197] Qing Xie and Atif M. Memon. Designing and comparing automated test oracles for GUI-based software applications. *ACM Transactions on Software Engineering and Methodology*, 16(1):4, 2007.

[198] T. Xie and D. Notkin. Mutually enhancing test generation and specification inference. In *FATES*, pages 60–69, 2003.

[199] Tao Xie. Augmenting automatically generated unit-test suites with regression oracle checking. In *Proc. 20th European Conference on Object-Oriented Programming (ECOOP 2006)*, pages 380–403, July 2006.

[200] Tao Xie and David Notkin. Checking inside the black box: Regression testing by comparing value spectra. *IEEE Transactions on Software Engineering*, 31(10):869–883, October 2005.

[201] Zhihong Xu, Yunho Kim, Moonzoo Kim, Gregg Rothermel, and Myra B. Cohen. Directed test suite augmentation: techniques and tradeoffs. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, FSE '10, pages 257–266, New York, NY, USA, 2010. ACM.

[202] Shin Yoo. Metamorphic testing of stochastic optimisation. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, ICSTW '10, pages 192–201. IEEE Computer Society, 2010.

[203] Shin Yoo and Mark Harman. Regression testing minimisation, selection and prioritisation: A survey. *Software Testing, Verification, and Reliability*, 22(2):67–120, March 2012.

[204] Bo Yu, Liang Kong, Yufeng Zhang, and Hong Zhu. Testing Java components based on algebraic specifications. *Software Testing, Verification, and Validation, 2008 International Conference on*, 0:190–199, 2008.

[205] A. Zeller and R Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.

[206] S. Zhang, D. Saff, Y. Bu, and M.D. Ernst. Combined static and dynamic automated test generation. In *ISSTA*, volume 11, pages 353–363, 2011.

[207] Wujie Zheng, Hao Ma, Michael R. Lyu, Tao Xie, and Irwin King. Mining test oracles of web search engines. In *Proc. 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Short Paper*, ASE 2011, pages 408–411, 2011.

[208] Zhi Quan Zhou, ShuJia Zhang, Markus Hagenbuchner, T. H. Tse, Fei-Ching Kuo, and T. Y. Chen. Automated functional testing of online search services. *Software Testing, Verification and Reliability*, 22(4):221–243, 2012.

[209] Hong Zhu. A note on test oracles and semantics of algebraic specifications. In *Proceedings of the 3rd International Conference on Quality Software*, QSIC 2003, pages 91–98, 2003.

[210] B. Zorn and P. Hilfinger. A memory allocation profiler for C and Lisp programs. In *Proceedings of the Summer USENIX Conference*, pages 223–237, 1988.