**Introduction:-**

**Problem statement & Motivation:** The technology has been advanced in automated test case input generation, but none of them addressed the issue of checking generated inputs with respect to the expected behaviors that is, there is no automated solution to the test oracle problem. The work on test oracle problem in Binder’s Textbook in 2000 and work on automated test oracles for functional properties by Pezze and Zhang in 2014 has provided motivation to this paper.

This paper focuses on how test oracles can be automated along with the understanding of different types of oracles, their evolutions over time and further progress on how we can automate test oracles.

In simple terms, a test oracle distinguishes a test case correct or incorrect behaviour. Extensive research has been done involving the automation of test oracles. As this is a great hindrance to achieve complete test automation. Without this automation, the traditional approach of involving human to verify the correct behaviour of the test case should be employed. Therefore, **we need a formalised procedure to differentiate the actual output from the observed output for System Under Test (SUT)**.

Several approaches to automate testing has been carried out over the past years such as to automate test scripts generation, automatic test scripts runs, detailed bug reports, etc. However, the **area of test oracles is not been looked into greater detail hence posing a road blocker to attain test automation**. To begin with, automatic test input generation can be considered as a stepping stone towards test oracle automation. We have seen that Automated test input generation has been discussed as part of Search-Based testing and Dynamic Symbolic execution however the main intent of finding inputs that can expose faults has not been addressed in this research.

**Partial test oracles:** There can be situations where the exact **detailed specifications are unavailable in the SUT**. In such cases, **the existing relationships of the expected behaviour can be analysed to build partial test oracles**. However, we see in most of the cases, the quality assurance analyst does not have access to the formal specifications or partial test oracles and hence have to manually write the test scripts by going through all the requirements. Hence to avoid this problem, we need a way to automate the test oracles and combine them with the existing testing techniques.

**Definitions required to understand test oracles :-** The following section focuses on the terminologies used to understand the concept of test oracles.

**Test activities:** (Test Activities). For the SUT p, S is the set of stimuli that trigger or constrain p’s computation and R is the set of observable responses to a stimulus of p. S and R are disjoint. Test activities form the set A ¼ S ] R.

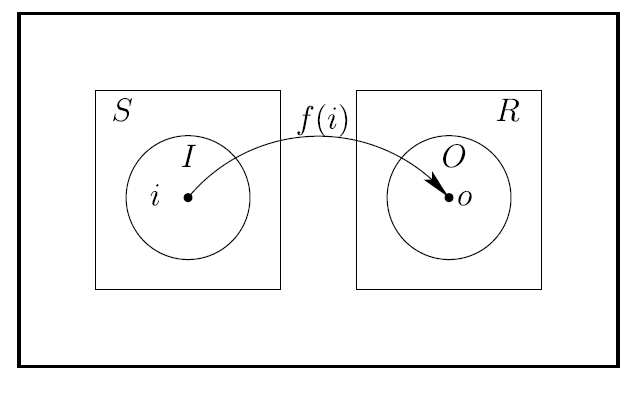


Fig 1:Test activities for Set S under SUT

For a given input, test oracles can either return a positive or negative response for the given test activity under a SUT. Test oracles can be built upon by **analysing the expected outputs(‘responses’)** for **input values also referred to as ‘stimulus’**. Collectively, stimuli and response together focus on the subset of the components.

Common examples of stimulus include **inputted values** on a form, prerequisites, inputs from different system, configuration files, resource constraints, database structures and table data, sensory inputs, involved devices states, etc.

Similarly, responses can also be referred as **observations that is generated as a result of the test case execution**. They can be output values generated from a system, destination device states, memory utilised, bugs reported, number of files executed, power utilisation, etc.

**Test Activity Sequence:** It determines following the constraints such that it can be utilised in the system. The sequence also simplifies the test case execution by directing the testers to satisfy the required constraints.

A test activity sequence is an element of TA = {ω|T→ω} over the grammar

T ::= A’[‘ϕ’]T |AT|ɛ

Where A is test activity alphabet. The testing sequence io : [ o = i2] signifies calling function f on input i and observing the output o.

To check the behavior of a given test input sequence, we need to define test oracle.

**Test oracle:** A test oracle D : TA →Β is a partial function from a test activity sequence to true or false.

**Probabilistic Test Oracle:** A probabilistic test oracle Ď : TA →[0,1]maps a test activity sequence into the interval [0,1] ϵ R. This helps in modeling the case where the test case only offers an acceptable test case probability.

The test oracle D is said to be sound if D(a) => g(a) and said to be complete if g(a) => D(a) where g is total test oracle that gives the right answer. To achieve an arbitrary confidence that to know if the test sequence ω is valid by repeated sampling Ď on ω, the non-negligible advantage ɛ requires Ď to maximize entropy.

**Research Trends in Test oracles:-**

This section focuses on the research analysis carried out in the past and further how the oracles are classified into various types based on their behaviour and functionality. By applying the regression models, the high values of determination (R2) for the coefficients confirms that models are good fit to the trend data.

**Volume of publications:** An extensive research has been conducted and researchers have worked on around 694 papers that have been published starting from 1970’s on the topic of software test oracles. The paper claims that the test oracles are broadly categorized into four categories:

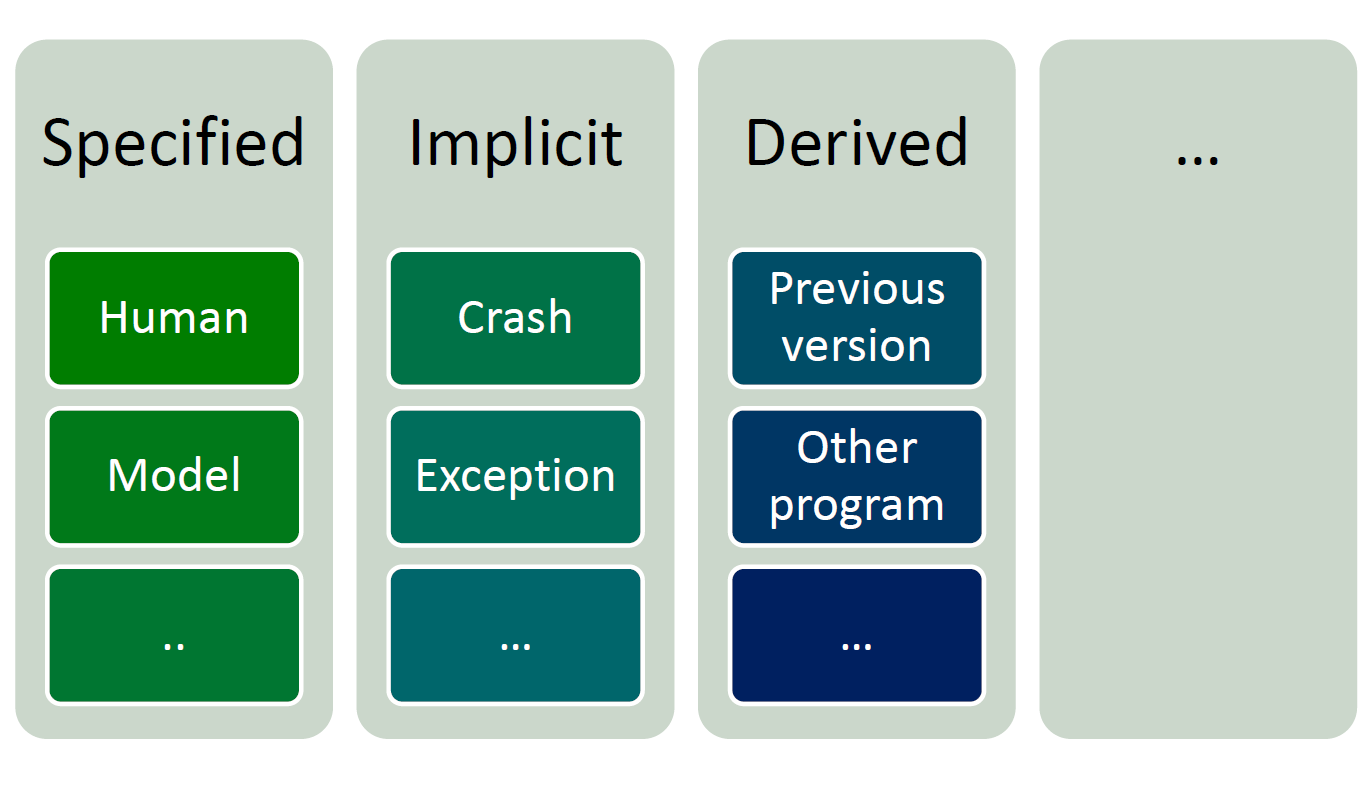
* specified test oracles,
* derived test oracles,
* implicit test oracles and
* no test oracle that deals with scenarios in the absence of test oracles.

**Specified test oracles**: Test the behavioural characteristics for a given set of detailed specifications in a system.

**Derived test oracles:** They consider components from which a test oracle can be extracted such as preceding version of the test system.

**Implicit test oracles:** This type of oracles are very straightforward as they look for evident faults in the test system.

A brief illustration of the different types of oracles is given in the below figure.



**Evolution of test oracle techniques:**

The below figures illustrate the advent of research papers for each specific category. We see that progressive amount of work has been carried out from 1978 to 2010 on test oracles with more focus on Specified and Implicit test oracles.

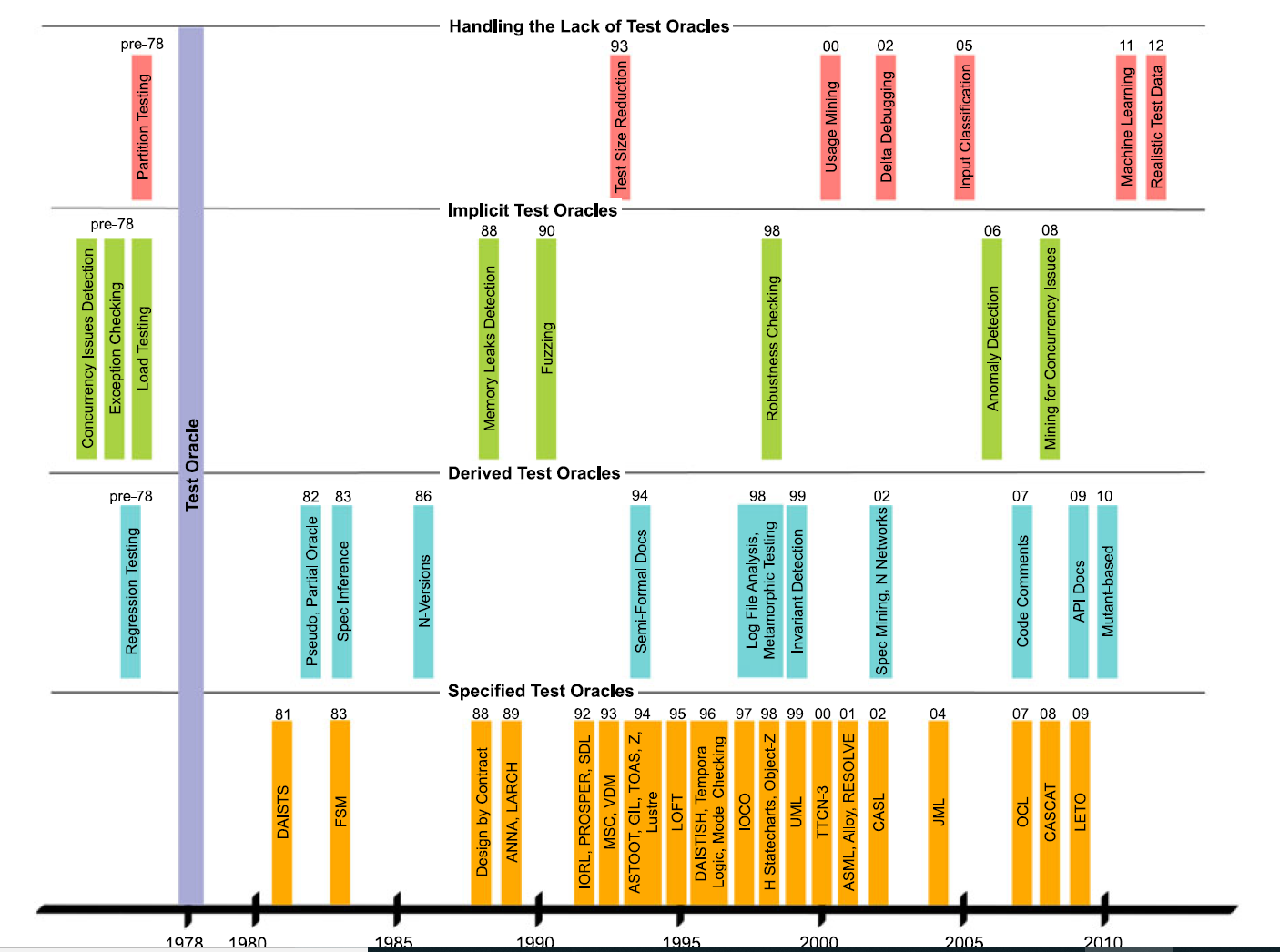
**Every vertical bar represents the test oracle technique along with the concept used labelled against the year it was published**. For instance, the study on Finite State Machines was carried out as early as in 1950s however the actual use of FSM was made in the papers written by ‘Jard and Bochmann’ and ‘Howden’.

This figure only focuses on the publications that were published year wise along with the progress reports documented in conference and workshops. It does not cover any other documentation such as technical reports and manuals.

An interesting observation shows that the works carried out before 1978 as shown in the figure was not necessarily on test oracles but the **concepts and technical developments that lead to the creation on test oracles**. For instance, work on **detecting deadlocks, live locks** as part of ‘Concurrency detection’ and conducting performance type of testing such as Regression/Load testing. This kind of testing also involves checking its former versions of software to check its performance and hence **lays a roadmap** for ‘**Derived test oracles**’.

Similarly, partition testing requires human intervention to select test case and hence this can be considered under human oracles. Contrastingly, it also covers the lack of automatic test oracles, however this field of research also included explicit factors such as **human oracle cost problem**.

Lastly, we can see that the **volume of publications** published for ‘**Specified test oracles**’ was **very high,** but it has considerably reduced in the early 2000’s. Hence, this figure covers the various concepts that were worked on in a chronological sequence from pre-1978 till 2010 to give us a comprehensive overview of **history of test oracles**.



**4. Specified test oracles:**

Requirements are detailed into formal specifications and can be considered as a source of test oracle information. Therefore, **a specified test oracle D** can be used to represent a specification language which can distinguish whether **the behaviour of a system can confirm to its formal specification**. Over the past years, extensive research and study has been carried out on formal specification and this has led to the development of four broad categories, namely:

1. Model based specification languages
2. State transition systems
3. Assertions and Contracts
4. Algebraic specification languages

A close up of a sign

Description automatically generated

**4.1 Specification languages**

**4.1.1 Model based specification languages:** This type of formal specifications model a system by interpreting them as **a collection of state and transitions**, therefore they can also be referred as **“state-based specifications”**. To model this kind of a system, we must consider the **preconditions and postconditions that constrain the systems actions**. Preconditions define constraints that apply to the input states of the system operation while postconditions focuses on the impact the operation has on the program state. Hence, we can observe that postconditions are stronger than preconditions.

The direct applications of these **specification languages** can be used by developers to solve problems. These languages help to **build relationships** between specification and implementation that can be used **to visualise iterative incremental versions of software**.

Popular examples of languages developed and used in the industry are: **Z, UML/OCL, VDM/VDM-SL, Alloy**, etc. Languages such as VDM, Z and B can be used to identify unexpected behaviour that can be used for testing. If a test case causes voilation for an invariant can be used to discover unexpected behaviour and hence these invariants can be treated as partial test oracles.

Parnas has introduced the concept of Test Oracle Generator (TOG) from program documentation. This kind of documentation is formatted in tabular expressions including the method signature, the external variables and by specifying the relation between the start and end states. Therefore, test oracles can be automatically generated to verify the outputs against the given states of a system or software.

**4.1.2 State Transition Systems**

**State Transition Systems provide a clean and visual representation of the system by presenting a graphical syntax**. It is made up of input and output states along with the transitions leading to different states in the system. A state can be defined as the abstract representation of concrete state of the previously specified modeled system.

We see that various types of state transition systems have been defined such as **Finite State Machines (FSM), PROMELA, Mealy/Moore machines, SDL, I/O Automata, Harel statecharts, Labeled transition systems, UML State Machines**, etc.

State transition systems can be used for system automated reasoning as they consist of only a finite set of states and their behaviour can be abstracted into states defined by a definite set of values. These systems capture the behaviour of a SUT, the **transitions represent stimuli** that can cause system to change and the **output of the system can be modeled as property of states** such as in **Moore machines** or transitions traversed as in Mealy machines.

**Model based testing** and **Protocol conformance testing** were the two significant motivation for applying state transitions systems to testing. We notice that models and SUT are mostly similar to each other and there is a fine line of difference among the two. Hence, identifying behavioural difference among the two can be quite a challenging task.

**For a given specification F acting as a state transition system, a test case can be obtained from sequence of transitions in F**. The input states can be taken from the transition labels. In this scenario, a test oracle can be constructed from F as : if F takes the test sequence and gives an output, then the SUT should also behave in a similar way. On the contrary, if F doesn’t accept the input, then the SUT shouldn’t take it as well.

**4.2 Assertions and Contracts:**

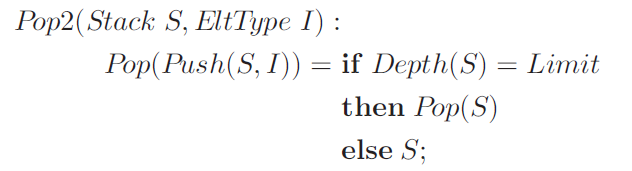
Assertions are commonly used in programming languages to verify the set of properties at a certain point in a program. **Assertion is a simple Boolean expression that is marked at a certain point in a piece of code/program**. The main purpose of an assertion to check the behaviour of the code at that line in **runtime**. If the assertion returns true, it means the program is working as ‘expected’ and if the assertion returns false, it means there is some bug that needs to be fixed.

Since assertions have been widely used by programmers and testers **to check the functionality of the program and identify faults**, it makes an obvious choice to **use assertions as test oracles**. Popular programming languages such as C, C++, Java, C#,etc provide assertion constructs/variables to enable developers to utilise them during programming.

However, assertions have limited functionality to check only limited set of properties at a particular point. Hence, Cheon and Leavens came up with an idea of how assertions can be extended to **Java** **Modelling languages** that work on the principle of **design by contract**. This contract provides a means to verify contracts for client and supplier objects represented as pre and post condition constraints and program invariants.

**4.3 Algebraic Specification languages**

These type of specification languages are used to represent a software module or component in terms of sorts and operation symbols that together combines its interface. An ideal example for this type of language is an **Abstract Data type that combines data and operations in a suitable format of representation**. Data structures such as graphs, tree, stacks, queues have been integral parts of any programming language and one of the earliest languages that implemented this was DAISTS. For example, the below figure represents an axiom used in DAISTS:



This axiom can be applied to a test oracle against the specification to verify its correctness.

Given specifications:

**Top : Returns the top element of stack without changing the contents of the stack.**

**Pop : Returns a new stack by removing the previous top element.**

For the above specifications, **the test oracle runs the above axiom against a set of implemented components for a test suite**. If the outcome is true, the behaviour is as expected and we gain some confidence in the system. If the outcome is false, we can conclude that a fault has been identified either in the axiom or in the program that needs attention.

Several theories have been proposed on algebraic specifications. The important ones are Gaudel’s theory followed by Frankl and Doong. Their works focus on **observations equivalence and inequivalence**. Therefore, **an algebraic specification language LOBAS and a test harness tool called ASTOOT** was proposed by Frankl and Doong. To implement this concept, **the code should implement a testing function ‘EQN’ that is used by ASTOOT to verify whether the class instances are equivalent or inequivalent.**

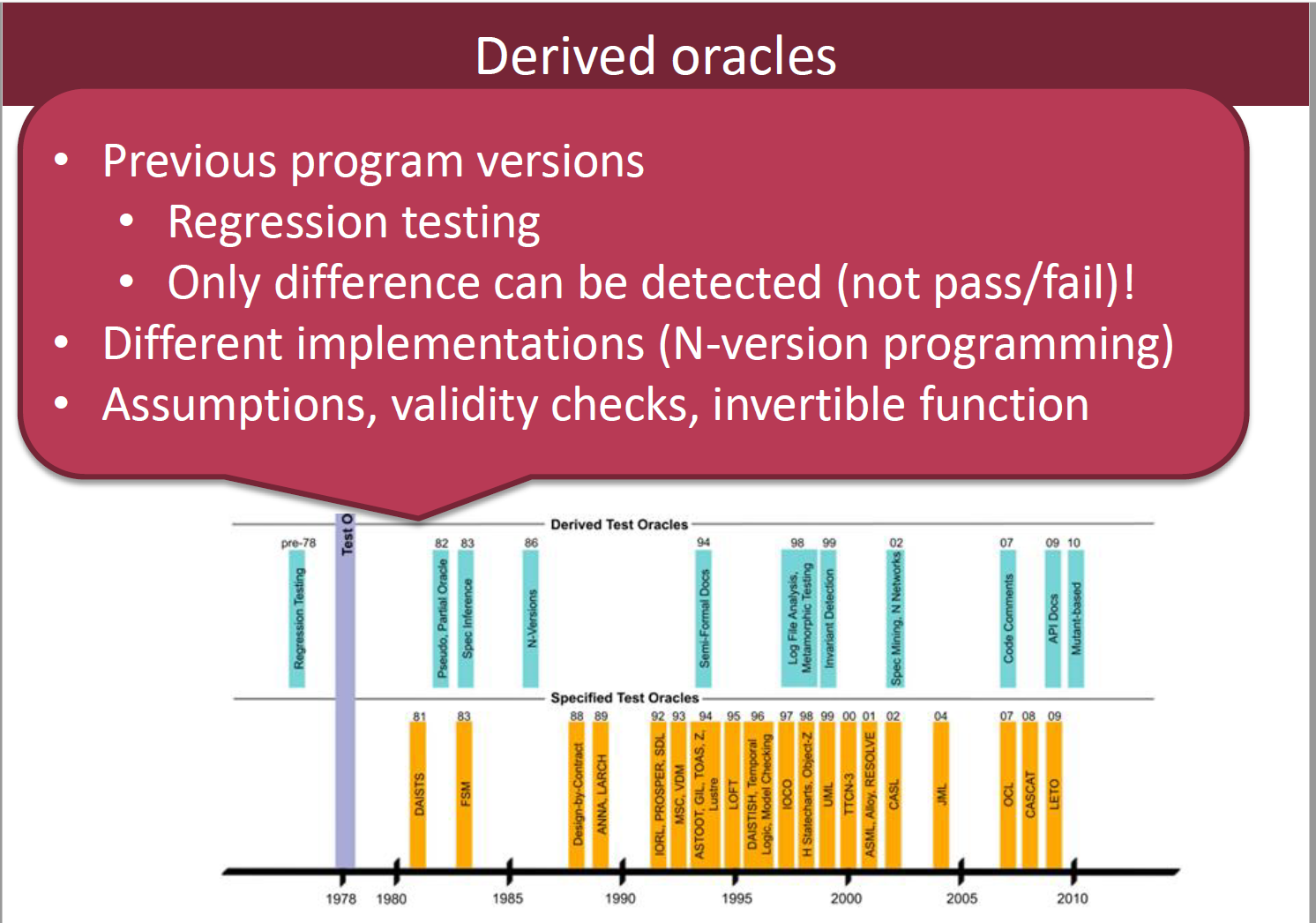
Therefore, **a specified test oracle can be conceived when LOBAS checks the specifications against the equivalence of an object in ASTOOT**.

Similarly, other examples of **algebraic languages that have been used as test oracles are LOTOS and Zhu**.

**Specified Test Oracle Challenges**

Three challenges must be overcome to build specified test oracles.

1. **Absence of formal specifications** that affects test oracle generations for a few cases but not all.
2. **Imprecision** in capturing the required behaviour while checking for a specification.
3. Difficulties in **interpreting model output as concrete program output**.



* **Derived test oracles**

A derived test oracle separates correct and incorrect behaviour by using **information deriv**

**ed from artefacts of the system**. This may include documentation and characteristics of versions of the SUT. **Regression test suites (or reports)** are an example of a derived test oracle - they are **built on the assumption that the result from a previous system version can be used as aid (oracle) for a future system version**. Textual documentation from previous system versions may be used as a basis to guide expectations in future system versions.

1). Pseudo-Oracles:-

An alternate version of the program as an oracle.

● Pseudo Oracle because we know the two don’t agree, but don’t know which is wrong or why. Also called N-Version programming, where multiple designs are implemented, or same design is implemented by independent teams.

Pseudo oracles are utilized to test programming for which no other is accessible, that it is too hard to even consider determining what the right yield ought to be, or on the other hand in light of the fact that oracle is impossible to produce.

2). Metamorphic Relations:-

A particular change to the input “changes” the output . To test programs without the involvement of an oracle. It employs properties of the target function, known as metamorphic relations, to generate follow-up test cases and verify the outputs automatically.

If you have test cases, you can generate partial oracles for follow-up tests by deriving metamorphic relations between tests. If these relationships are violated, then there is a bug.

3). Regression Test Suites:-

A regression test suite is the set of test situations that are intended to guarantee that product is exact. Each level of testing ought to have its very own regression test suite. When changes are made to a system, rerun your tests. Any existing tests that passed previously should still pass. An older version of your program can be the oracle.

● Do new features break working features?

● Do bug fixes break working features?

● If requirements have changed, we do NOT want the output to match for features related to the requirement.

4). System Executions:-

Two main techniques for deriving test oracles from traces—invariant detection and specification mining.

Invariant detection:-

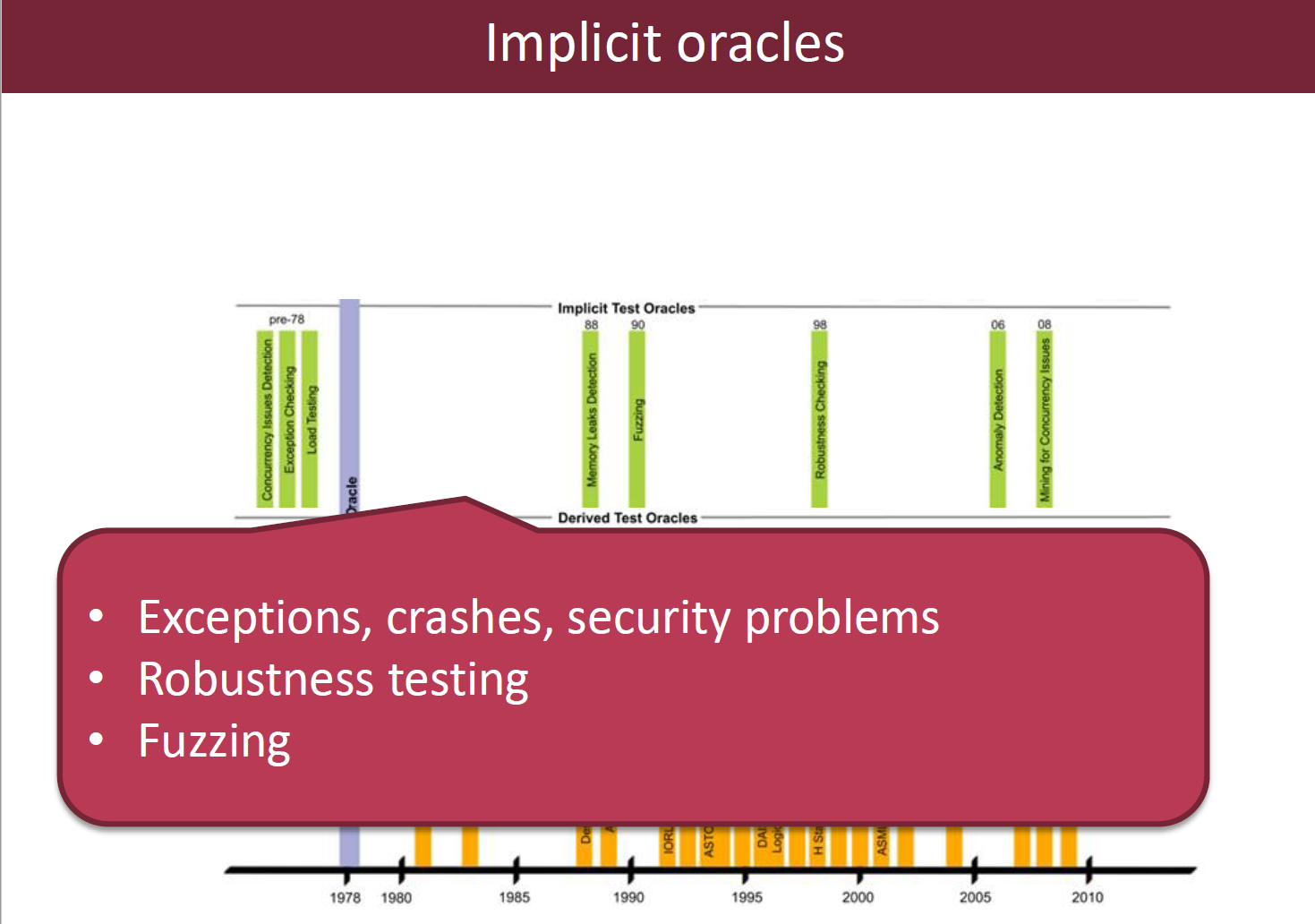
Program behaviours can be checked against invariants. Thus, invariants can serve as test oracles. The invariants are checked by binding their variables to the program’s variables. The variants notices the program`s behaviour and can be used to check program`s correctness.

Specification Mining:-

Specification mining deduces a formal model of program behaviour. A test oracle can enforce these formal models over test activities. The testing searches for I/O matches that portray each part of both the expected and genuine practices, while surmising begins with a lot of I/O sets, and infers a program to fit the given conduct.

5). Textual Documentation

Documentation also serves as an oracle in a way that it describes the functionalities expected from SUT. These are actually informal. They are also partial, unambiguous in contrast to specification languages. There are two approaches :- one is to build a technique to make formal specification from artefacts and second one makes a natural language to semi formal which can be amended to automatic processing.



**An implicit oracle :-**

Implicit oracle can be applied to check the specification that can be expected from any runnable program. It don`t require any domain knowledge. Implicit oracles often detect network irregularities or deadlock. Implicit oracles can be built to detect Concurrency Issues - Deadlock, live lock, and violations related to non-functional attributes of the system like:

* Performance properties
* Robustness
* Memory access and leaks.

Implicit test oracle are not universal means behaviour for one system could be normal for particular and it could be abnormal for another system. Fuzzing is an effective way to find a crashes. The main idea is to generate a “fuzz” is to feed them to the system to find anomalies. **Fuzzing is used to detect buffer overflows, denial of service etc**.

**THE HUMAN ORACLE PROBLEM**

The researchers say that no such artefact exists like humans to verify whether the behaviour of software is correct or not for a given stimuli. Human oracle cost reduces human involvement in two dimensions such as **writing test oracles** and **evaluating test outcomes**.

**Handling the lack of Oracles**

Even if there is no oracle, there are techniques that can reduce the human oracle cost through:

● Quantitative reduction in the amount of work the tester has to do for the same amount of fault-detection potential.

● Qualitative reduction increase in the ease of evaluating and understanding test cases.

**1).** Quantitative Human Oracle Cost:-

Researchers claim that test cases may be unnecessarily large, for containing redundant method calls even though they doesn’t contribute to the overall test case. Test suite reduction techniques maximize coverage and minimize test suite size to alleviate human oracle cost problem. Test case reduction techniques attempt to remove unnecessary test steps.

2). Qualitative Human Oracle Cost

Not all test cases are equally understandable by human testers. Automated test generation often produces test inputs that do not match the expected usage of a program, and humans have trouble judging the results of such tests. Some test generation approaches such as augmenting test suites, computing realistic inputs and mining usage patterns to alleviate the human oracle cost problem qualitatively.

3). Crowdsourcing the Test Oracle:

Crowdsourcing is performed by submitting tasks to generic platform such as Amazon Mechanical Turk and also crowd should be able to provide sufficient documentation and should determine the correct output from the incorrect ones. Researchers found Mob4Hire, MobTest, uTest as some of the crowdsourcing services for testing mobile applications.

**FUTURE DIRECTIONS AND CONCLUSION:**

Researchers have done sufficient comprehensive survey of test oracles and managed to find that test oracle reuse and test oracle metrics although the test oracle problem is open to new research directions. **Building test oracles based on older versions of the system and sharing of oracular data are two of the promising approaches for test oracle reuse**. Measure of how well the program has been tested and the measurement of assessing the quality of assertions comes under oracle metrics which is a challenged opportunity.

**New idea:**

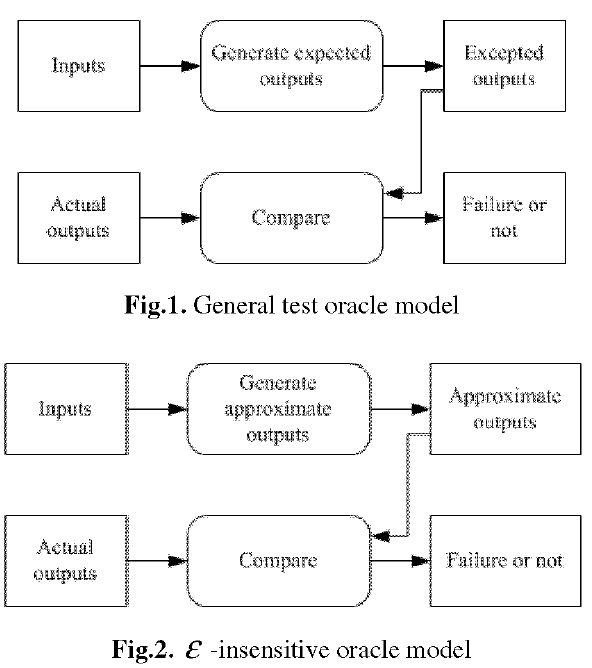
To develop an automated test oracle, we must make sure the expected output is generated and compared with actual output automatically. As part of this paper, we propose an automated test oracle that generates approximate outputs which is very close to the expected output. The approximate outputs are estimated by using neural networks working with supervised learning.

Several techniques have been proposed to document the expected output to automate test oracles. We can **store the results in a database and perform a simple lookup, create state transition system such as FSM or simply evaluate the results against Boolean expressions**. These approaches can be used to determine whether the behavior is expected or not. Research has shown that **neural networks can be used to test software and act test oracles with acceptable degree of accuracy**.

The **significance of neural networks** lies in its approach to behave as a test oracle and predict the outputs of a system without having prior knowledge of the system. Research experiments have been carried out on continuous function for single and multiple input value to validate the responses.

**Approximate output generation:**

The approximate outputs can differ from expected outputs by a certain degree of precision. This is clearly explained in the below figure:



**Automated test oracle generations procedure:**

Test oracle can be automated by dividing the procedure into the main two steps:

1. To **obtain approximate outputs** by using neural networks approach.
2. **Compare approximate output with expected output**.

The following steps detail the procedure to **automate the process using neural networks approach**:

1)Generate neural networks function based on the acceptable degree of accuracy.

2)Train the function using supervised learning by providing various training sets.

3)Provide single or multiple inputs to the function and record the outputs from the system.

4)Obtain expected output from the system and compare with the output in Step 3.

5)The steps 3 & 4 are repeated until the entire application under SUT is tested or all test suites are generated.

6) Determine if there is a fault in the system or not.

The **main advantage of using neural networks** is that the **precision achieved in the experiments conducted so far is acceptable** but however it requires training to determine the input, output relationships and an extensive research is more required towards this direction.

To conclude, this approach can **save huge time and efforts** employed in software testing and can be considered as **‘one step closer to achieve test automation’**.