Theoretical and Applied Testing Methods in Modern Object-Oriented Systems

Athens University of Economics and Business

Tsirmpas Dimitris, Mandelias Alexios

September 15, 2022

Contents

1 Abstract

Software verification and validation has always been one of the most important aspects of software development. It is currently estimated that more than 50% of software development time is consumed in the testing phase. On the other hand, the prevalence of object-oriented programming languages has radically changed not only the way software is designed and developed, but also the requirements and difficulties in the verification process, resulting in many established verification techniques no longer being useful to developers. For this reason, intensive research has been conducted in recent years to both identify which techniques are still effective and to find new, suitable ones for adressing the challenges of object-oriented systems. In this paper we investigate the difficulties of using the object-oriented model in the testing process, the differences between object-oriented and traditional (procedural) testing, and ways of reconciling these differences. We analyze which techniques have ultimately withstood the test of time and remain in use today, as well as new techniques and tools developed to address modern challenges.

2 Procedural vs Object-Oriented Testing Model

The main differences between the two testing models are summed up on Image 2 (Hayes and Huffman [8] as cited by Gordon and Roggio [7]).

This table refers to 5 of Haeys' 11 axioms, as the rest are common between the development model, according to the author. We can therefore estimate that half of the principles on which the procedural testing model are based do not apply in the object-oriented model.

We describe the core differences in detail below, as well as the difficulties to which they are related.

2.1 State

In the procedural model a test can be viewed as an ordered set of input-output pairs, since the result of a call in this model depends only on the input given to it.

In the object-oriented model, each object normally has an internal state. Methods called on this object not only produce outputs but also change its internal state (i.e. have side effects).

Figure 1: Hayes's most relevant axioms

Axiom	**Description**	**Traditional**	**Object-Oriented**
Complexity	For all n, there is a program that is adequately tested by a test set of size n, but not by a test set of size n-1	There is a minimum set of inputs that must be tested	There is a minimum set of inputs and object states that must be tested
Anti-extensionality	There are programs P and Q that compute the same functions (semantically similar), where T is adequate for P but not for Q	It cannot be assumed that the same test cases can be used for different programs that accomplish the same things	It cannot be assumed that the same test cases can be used for functionally similar programs, this can be extended to mean that just because one state is correct for one program, that does not mean that is correct for a similar program
General Multiple Change	There are programs P and Q that are syntactically similar, where T is adequate for P but not for Q	Syntax does not tell you what needs to be tested	The syntax of two programs does not determine the test sets, this also means that if two programs use the same classes, the test cases should be different because the messages sent between them may be different
Anti-decomposition	There is a program P and component C where T is adequate for P and T' is the subset of T that can be used for Q, but T' is not adequate for Q.	A component of a program (say a method) can be adequately tested for use within one program, but not necessarily on its own.	"When a new subclass is added (or an existing subclass is modified) all the methods inherited from each of its ancestor super classes must be retested."
Anti-composition	There exist programs P and Q and a test set T where T is adequate for P and the subset of T that can be used for Q is adequate for Q, but T is not valid for P;Q (the composition of P and Q).	Two programs (or methods) can be adequately tested on their own, but once combined or used in another class; they may no longer be adequately tested.	"If only one module of a program is changed, it seems intuitive that testing should be able to be limited to just the modified unit. However, [this] states that every dependent unit must be retested as well."

It is therefore obvious that in the object-oriented model the above representation of "test" does not apply, since the result of a call is determined by the state of the object to which it is called, which may change between calls. A further complication is that the change of the object's state is often not perceived by observing the program outputs.

To cope with the new requirements, both the change in the internal state of the objects and all the different series of calls to its methods must be taken into account when checking.

In particular, according to Gordon and Roggio [7], for a correct and well-developed control, the following are required:

- A list of messages and actions which *could* be executed by the test.
- A list of exceptions that may be thrown.
- A stable execution environment.
- All other supplementary information that is needed for the test.

2.2 Polymorphism

Polymorphism introduces the problem that there are no stable assumptions during compilation. Because it is a dynamic operation, it is impossible to know *which* code will be executed at the test.

Classes in the object-oriented model are generally infinitely extensible, and each extension of a class can extend and redefine the functions of the classes above it in the hierarchy. For each test we would therefore be forced to check all possible subclasses of the object, which is impossible, since at any time we can add a new subclass of the tested class.

A good example of the problem is given by Tauro, Ganesan, and Ghosh [18]:

> “
> As an example, a method used to draw graphics. If three dimensions are passed, the method draw creates a triangle, if four dimensions a rectangle, five pentagons and so on. In this case there are three different methods with the same name but they accept different amounts of variables in the method call. Generally the selection of the variant is determined at run-time(dynamic binding) by the compiler based on, for example, the type or number of arguments passed. Few questions arise that need to be considered before testing:
>
> - Do we only need to test one variant?
> - Do we test all variants?
> - If all, do we need to test all at all levels?
>
> There isn‘t one set answer for these questions. The answers to these questions will depend on the testers, the companies policies etc. In a perfect world we would test everything. However, in reality it is generally impossible to test everything in large scale projects.
> ”

The authors also refer to the function of dynamic binding. For example, C++ mainly uses static binding, which is done at compile time, while Java uses dynamic binding, through the JVM (Java Virtual Machine). In the case of the latter, because the program knows only at runtime the types of parameters, return value and other data used within, the above problems are further complicated.

The authors advise that this problem should be handled if possible by the Integration and System Test layers instead of Unit Tests.

In Sharma, Porwal, and Sharma [16] one solution suggested is to define a hierarchy specification, i.e. all subclasses agree on a common, minimum functionality. Other solutions will be discussed later in the document.

2.3 Inheritance

In the above paragraph a brief mention was made of the problem of overriding the methods of the superclass. This problem is not only limited to the infinite number of possible class types answering a call, but also extends to the scope of the superclass itself.

Suppose we have a fully tested class A and a subclass of B that redefines only a subset of the methods of A. It would be natural to assume that we only need to check these methods. According to Perry and Kaiser [13] as cited by Barbey and Strohmeier [1], this assumption does not hold, because the changed methods may be used by other inherited methods, thus changing the behavior of the latter or changing the internal state of the object in unexpected ways. Another problem that inheritance poses is the violation of encapsulation, as a subclass may have access - indirectly or directly - to the data of its superclasses.

Therefore, we are forced to either re-verify all the properties of the new class or discover the minimal set of properties that are different from the parent class. Algorithms and techniques for finding this set are mentioned later in the paper.

Curiously, multiple inheritance does not pose problems in the verification process, at least as far as modern object-oriented languages are concerned (Barbey and Strohmeier [1], page 11).

2.4 Encapsulation

Encapsulation or ”data hiding” is a fundamental principle of object-oriented programming, according to which the internal data of an object can only be changed and accessed through an interface. Since, as explained above, controlling the internal state of the object is necessary for class testing, this visibility constraint does not allow us to properly evaluate it.

Intuitive solutions require either breaking the encapsulation itself, or deriving classes that provide us with access methods. The first solution violates the reason for the existence of a class, while the second may not even solve the problem, in cases where the class is a subclass and does not have access to the internal data of its superclasses.

Finally, we include a table (Table 2.4) from Kung et al. [10] in which we summarize the main changes to a section of source code that brings changes to the tests’ code.

Figure 2: Possible changes to the test's code

Components			Changes
Data	Component Changes	1	Change data definition/declaration/uses
		2	Change data access scope/mode
		3	Add/delete data
Method	Interface Changes	4	Add/delete external data usage
		5	Add/delete external data updates
		6	Add/delete/change a method call/a message
		7	Change its signature
	Structure Changes	8	Add/delete a sequential segement
		9	Add/delete/change a branch/loop
	Component Changes	10	Change a control sequence
		11	Add/delete/change local data
		12	Change a sequential segement*
Class	Component Changes	13	Change a defined/redefined method
		14	Add/delete a defined/redefined method
		15	Add/delete/change a defined data attribute
		16	Add/delete a virtual abstract method
		17	Change as attribute access mode/scope
	Relationship Changes	18	Add/delete a superclass
		19	Add/delete a subclass
		20	Add/delete as object pointer
		21	Add/delete an aggregated object
		22	Add/delete an object message
Class	Component Changes	22	Change a class (defined attributes)
Library	Relationship Changes	23	Add/delete a relation between classes
		24	Add/delete a class and its relations
		25	Add/delete an independent class

3 Solutions

3.1 Control Units

During Unit Testing, test cases are written so as to cover as many possible object states in an attempt to find errors. As the complexity of the software increases, exhaustive checking of all execution paths and all behaviors of these objects becomes practically impossible. This is further reinforced by the fact that the number of possible states of a single software object is prohibitively large to check manually, and automatic test case construction tools construct an equally prohibitively large tree of possible states. Furthermore, the traditional way of testing with hard-coded data significantly limits the number of different execution paths that are actually tested, making a definitive answer as to whether there are still bugs even more difficult to give.

3.1.1 YAQC4J

An example of adapting more traditional control techniques to the object-oriented model is the tool **YAQC4J**. The technique used is randomized data generation, which first appeared in associative programming, specifically with the QuickCheck [14] tool. The basic principles underlying the operation of QuickCheck were

adapted and extended for the object-oriented programming model and implemented in the YAQC4J tool, which can be used as a complement to existing checking methods.

The tool undertakes the generation of randomized objects and then executes the test instances a large number of times until either an execution fails or a predefined number of executions succeed. If it fails, we learn that a bug was found in the software, otherwise we learn with a high degree of certainty that the software contains no bugs.

Although based on the same logic as QuickCheck, YAQC4J certainly does not enjoy the same popularity as its predecessor. This, together with the relative lack of popular tools based on random data generation, is an indication that this technique is not preferred by object-oriented programmers.

3.2 Integration Testing

As the software development progresses, more and more attention is invested inthe functionality of the software, so the Integration Test has to ensure that the software behaves correctly while combining different modules together. Object states as well as method overloading and the dynamic binding of objects to their references make interface testing difficult, since at each execution both the internal state of the objects changes, and each method call is likely to correspond to several implementations.

Problems that need to be addressed when checking concatenation are the following [2]:

- We should have the skeleton of the system as early as possible
- During integration we should take care so that the interfaces of the modules join and interact correctly
- The modules should communicate sufficiently well so that system testing can be performed.

Approach Before writing the Integration Tests we must answer the following questions:

1. How many objects do we need to create before we start the checks?
2. In which order should we merge the different units?

3. Do we need more than one skeleton for the integration?

The proposed method that answers these 3 questions is the following and uses a ”virtual integration engine” that is responsible for integrating the individual pieces of a system. The engine has a list of software use cases as input as well as a list of objects to be used. It attempts to execute each use case in turn, where each time a new object is required, one is selected from the list, and is then tested in conjunction with the other previously selected objects for proper functionality and integration.

This engine successfully addresses these three questions as it builds objects when they are needed, the different modules are merged according to the order specified by the software and a skeleton is provided for the integration.

3.3 Differential Testing

One of the greatest difficulties in software testing is evaluating the result of the test. Modern software systems are so complex that it is often a challenge to determine the expected result in advance, let alone for randomized testing. Destructive errors in software are relatively easy to detect, but logical and semantic errors are difficult to detect by traditional methods.

Differential Testing is an attempt to detect those hard-to-find logical and semantic errors in software by giving the same input to many similar applications and observing variations in the software's output or behavior. Using the different implementations as an ”oracle”, deviations in the behavior of the programs are identified which are potential bugs in the software [11].

Below we list a few methods with which this can be accomplished:

Non-guided Input Generation New inputs are generated randomly, ignoring the results from previous executions. Such a strategy is not efficient as inputs are selected from a prohibitively large set of possible inputs. The probability of finding at random those inputs that will exhibit a software error is extremely small, given the latent nature of the errors.

Driven Input Generation By taking into account information about software behavior from previous executions, we can more efficiently test the software with those inputs that are most likely to reveal a software bug.

Field-Independent Evolutionary Guidance Using software metrics we can evaluate the differences between similar programs to better determine the variation that exists between them. Thus, by selecting those programs that exhibit greater variation, differential control techniques are more effectively applied using black-box methods that are always field independent.

Symbolic Execution This white-box control technique executes the program symbolically, computing constraints on each execution path which it then solves in order to produce inputs that satisfy these constraints for each execution path. This technique, although difficult to use at scale due to the exponential complexity resulting from the number of different execution paths in different similar programs, can produce reliable inputs for differential testing.

3.4 Architecture

The ease of writing correct tests depends in part on the consistency of the class to which they are applied (Yeresime et al. [19] as referenced by Gordon and Roggio [7]). In the procedural model, cohesion is related to similar parameters and functionality, while in the object-oriented model it is related to how coherent the individual classes of the program are. A class is considered coherent if all its methods define it well as a self-contained unit.

A class with high coherence will have high similarity in its types and functions, so generating control data is easy. In the case of low coherence, having lots of heterogeneous data and functions means that tests are more costly and more error-prone.

Gordon and Roggio [7] propose a metric that represents the coherence of a class. Here is its definition:

> “
> LCOM (Lack of Cohesion in Methods) is defined as the mathematical difference between the number of methods whose instance variables are completely dissimilar, and the number of methods whose instance variables are shared (Yeresime, et al., 2012)
> […]
> If LCOM is high, it means that a class is not cohesive (and might be a candidate for refactoring into two classes. At the testing stage, a class will need to have different testing sets for the different methods rather than one testing set for the entire class. This leads to confusion and overall complexity of the testing process.
> The authors Badri and Fadel (Badri & Toure, 2012) found that LCOM and lines of code (LOC) were the most predictive testing metrics [over other software metrics].
> ”

A second concept related to the ease of writing checks is coupling. In general there is an inverse relationship between coherence and coupling: the more coherence a class has, the less coupling it has and vice versa (Khatri et al. [9] as cited by Gordon and Roggio [7]).

Coupling is a metric that assesses the degree of dependency between classes in the program. So while coherence deals with the relationships internally of a class, coupling deals with the relationships *between* classes.

High coupling means that all or most methods should be understood as a closely related set, rather than independent functions. This obviously negatively affects verification since Unit Checks are much harder to write. Low coupling is one of the goals of good software architecture, particularly for large, complex systems in which the delegation of tests to the System and Integration layers should be avoided as much as possible.

Feature testing should be done in Unit and Class Checks. Other tests include methods with return values that are polymorphic, as well as polymorphic parameters. This is more easily done in Integration or System Testing.

3.5 Inheritance

Class implementation is regarded to be simpler when it is done from the top down in the hierarchy. In the same way, controlling classes in an inheritance hierarchy is generally simpler when approached from the top down. By first looking at the top, we can deal with the common interface and code, and then the test code for each subclass. Implementing bottom-up inheritance hierarchies may require significant reconfiguration of the common code in a new superclass. The possibilities of dealing with different inheritance structures and the additional possibility of potentially dealing with multiple forms of inheritance can add another layer of complexity to the testing process. These issues raise a number of questions:

- Do we fully test all core classes and their subclasses, and at what levels should we test?
- Do we fully test all base classes or only changes or modifications to subclasses, and if so at what levels?
- In what order do we test the hierarchy, top-down or bottom-up?

To answer question 3, the generally accepted answer is top-down for the reasons explained above. The other questions are answered mainly in the Regression Testing section.

3.6 Artificial Intelligence

We study the contribution of AI to software testing by examining how it has been used in different parts of the software testing lifecycle.

3.6.1 Impact of AI in Software Testing

Subsubscription of Controls At the beginning of the software control lifecycle test instances are written according to the software requirements. The test cases are organized in a specification document to ensure that all requirements have been tested.

In this process Info-Fuzzy Networks (IFN) can be used to learn functional requirements from the execution data in order to recover missing or incomplete specifications, determine a minimum set of checks and evaluate the correctness of the software outputs.

Test Case Mapping The process in which only the most efficient control cases are selected for execution, resulting in a reduction of the overall test's cost.

Using Info-Fuzzy Networks (INF) correlations between inputs and corresponding software outputs are automatically identified. INF then learns from these correlations in order to create a categorization tree for control cases from which the most important ones are then selected. This significantly reduces the number of combinatorial black box checks.

Deduction of Control Cases The identification of control suitability criteria is followed by the generation of a set of controls that conform to these criteria. Such a process is obviously too complex to be performed manually, especially for complex systems, so there is a shift towards artificial intelligence regarding the automatic generation of control cases.

For black-box testing, a model of the software's behaviour is constructed by means of input-output examples, which is used to generate new inputs.

Test cases can be generated by directly analysing the Software Requirements Definition Document using Natural Language Processing (NLP) techniques.

Testing Data Generation The execution of the Test Cases requires the generation of the data that will be used to test the software. The better the quality of this data, the greater the coverage of the code by the test cases will be.

With the help of genetic algorithms, a search is performed in the software input space to find data which covers each execution path.

Using deep learning, using the "monkey" method, learning the inputs that the software controller would put in and statistically correlating them with the context is done. The "monkey" then predicts new inputs based on the observed context in each case.

Construction of an "Oracle" In order for software testing to perform its function properly, a mechanism is needed which has the ability, given the software as input, to distinguish between correct and false software behavior. This mechanism is the oracle problem.

Machine learning algorithms can be used to automatically construct oracles, even in cases where the software specification is incomplete. Each execution of the control produces results which are fed to the algorithm. The resulting model is used as the oracle.

When a reference software model exists, then supervised machine learning can also be used through which the resulting model learns to separate control instances as successful and unsuccessful.

Supervised learning is also applied to neural networks, where the model learns to separate execution patterns for successful and unsuccessful executions for a given software. A small subset of the execution traces are categorised as successful or unsuccessful and the model then learns from them.

Test Case Hierarchy The various test cases can be performed in many different ways. We try to find the optimal order of execution so that priority is given to those test cases that are most likely to reveal software defects, or to those that are associated with greater risk depending on, for example, the severity of the object under test or the impact of the risk if it occurs.

A set of existing techniques is brought together to act as machine learning to prioritise test cases using information such as code coverage, the age of the test, failure history and similarity of textual content, in order to construct a model that efficiently prioritises test cases

For SVM (support-vector machines, also known as support-vector networks) ranking, black-box metadata such as test case history and also natural language descriptions of the test cases are used to prioritize use cases.

Estimating the cost of test cases Cost estimation is the process in which the effort required to develop the software is predicted. Generally there should be no shortage of estimation calculations, and they should be available as early as possible.

Machine learning can predict the size of the audit code, i.e. the software metric ”audit code lines”, which is an important indicator of the effort that will be expended during the audit. Using techniques such as linear regression, K-nearest-neighbors, naive Bayes classifier, random forest and multi-level Perceptron, a model was constructed that accurately predicts this metric.

Because underestimating the test cases’ cost has serious consequences for software quality, models that estimate cost include some bias toward overestimation. Furthermore, we identify those characteristics that are most important for predicting software testing costs, specifically testing time:

Software Testing Activity	**AI Technique Applied**
Test Case Generation	Inductive Learning - Active Learning - Ant colony Optimization - Markov Model - AI Planner -GA - Tabu Search - NLP - Re-enforcement Learning C4.5 - Goal Based - Decision Tree - K-Nearest Neighbour - Logistic Regression - Random Forest - Multi-Layer Perceptron - K star - LSTM - Heuristic Search
Test Data Generation	GA - Simulated Annealing - Hill Climbing - Generative Model - LSTM - Deep Re-enforcement Learning - Ant Colony Optimization - Heuristic Methods
Test Oracle Construction	ANN - SVM - Decision Trees - AdaBoostM1 - Incremental Reduced Error Pruning (IREP) - Info Fuzzy Network
Test Case Prioritization	K-Means - Expectation-Maximization - c4.5 - Cob Web - Reinforcement Learning - CBR - ANN - Markov Model - K-NN - Logistic Regression - SVM Rank
Test Case Specification	IFN - C4.5
Test Case Refinement	IFN - Classification Tree Method
Test Cost Estimation	SVM - linear regression - k-NN - Naïve Bayes - C4.5 - Random Forest - Multilayer Perceptron

Table 1: Methods used by test type

3.6.2 Genetic Algorithms

As in the procedural model, in the object-oriented model, large-scale test case checking could make it impractical to run all possible test cases to verify the program. One solution to this problem is random testing.

Ciupa et al. [5] provide evidence that the relative number of bugs detected by random testing over time is predictable and indeed that most are found relatively quickly. The first failure is likely to occur within 30 seconds. Evolutionary algorithms, such as genetic algorithms, have been widely used for procedural software testing. In some places, approaches based on genetic algorithms have been proposed for testing object-oriented software.

An intuitive use of genetic programming is to construct tests that consist of a sequence of method calls. This could be implemented by encoding methods as identifiers that the algorithm would evaluate using a fitness function. However, it has been shown that this implementation produces so many invalid call sequences that the algorithm is inefficient.

In Meziane and Vadera [12], reference is made to S. and J. [15]), who also used genetic programming, not for randomized testing, but to produce automatic tests by generating and evolving trees representing functions or methods. The exact implementation is given below:

> “
> Instead they propose a novel use of Genetic Programming (GP), which aims to learn functions or programs by evolution. The underlying representation with most GP systems is a tree instead of a numeric list. In general, a tree represents a function, where leaf nodes represent arguments and non-terminal nodes denote functions. In context of testing, such trees can represent the dependencies between method calls which can then be linearised to produce tests. Use of mutation and crossover on these trees can result in invalid functions and inappropriate arguments in the context of testing object oriented programs. Hence, Wappler and Wegener(2006), suggest the LCOM use of strongly typed GP (Montana, 1995), where the types of nodes are utilized to ensure that only trees with appropriate arguments are evolved. This still leaves the issue of how such trees are obtained in the first place. The approach they adopt is to first obtain a method call dependency graph that has links between class nodes and method nodes. The links specify the methods that can be used to create instances of a class and which instances are needed by a particular method. This graph can then be traversed to generate trees to provide the initial population. The required arguments (objects) for the trees are obtained by a second level process that first involves generating linear method call sequences from the trees and then utilizes a GA to find the instantiations that are the fittest. Once this is achieved, the trees are optimized by use of recombination and mutation operators with respect to goals, such as method coverage.
> ”

A significant improvement on the work of Wappler and Wegener was brought by Ribeiro (2008), who ran the above algorithm ignoring methods that had no side effect (i.e., did not change the internal state of the object). A test on the Stack class of the Java Standard Library proved that this improvement reduced the execution time to full coverage by two-thirds.

Meziane and Vadera [12] also report another interesting use of genetic algorithms (GA) in the field of control development:

C., Feng, and Y. [4] explore the use of Genetic Algorithms (GAs) to determine the optimal order for class integration and control. A major problem in determining the appropriate ordering arises because the dependency cycles of classes must be broken, necessitating the use of stubs. The complexity of the required stubs varies depending on the level of coupling that exists between the classes, therefore different layouts require different levels of effort to create the stubs.

C., Feng, and Y. [4] exploit previous work on programming, for example in using GAs for the hawker problem, and use an encoding of classes with offsets along with a fitness function that measures the degree of coupling between classes. The fitness measure is defined to reduce the number of features, the methods that would need to be changed if a dependency is cut off, and the number of strains created. In addition, they do not allow the cropping of inheritance and composition dependencies, as they lead to costly strains. After experimenting with this approach in an ATM case study using the GA system "Evolver" (Palisade, 1998), the results are compared with those obtained using a graph-based approach and it is concluded that the use of GA can provide a better approach for generating embedding and class testing arrangements.

3.6.3 Neural Networks

In Sharma, Porwal, and Sharma [16], references are made to the work of Gong and Li [6], who proposed the use of Event-Driven Petri Nets (EDPN) so that changes in the internal state and behaviour of objects can be modelled.

> "The faults are detected by analyzing the differences of test scenario in the dynamic behaviors of EDPNs and the method can select a test case that detects errors described in the fault models. Ghang et al. [5] proposed a method using relative reliability test and operation paths' reliability prediction to adjust the test allocation of software reliability test in object-oriented programming. In their approach, software reliability test is based on the operational profiles and the relative reliability prediction results of operation paths."

So far this paper has focused exclusively on software verification, since this is the most time-consuming and most directly relevant to the object-oriented model. However, software validation, the verification that the software functionality meets the customer's requirements, is an equally important part of the development process. Here, neural networks can help us by analysing the customer's requirements, which are written in natural language, and converting them into a form suitable for designers and developers, thus avoiding human errors in this process.

Batarseh et al. [3] describe such a system:

> “
> The use cases obtained were crisp and consistent, irrespective of the size of requirements text. RTool, NL-OOPS, CM-BUILDER are few other NLP based computer aided software engineering tools [37]. Such tools produce class diagrams from the user requirement document (although it still requires user intervention).
> Michl et al. [37] proposed the NL-OOPS (Natural Language – Object-Oriented Production System) project to develop a tool supporting object oriented analysis. Requirements documents are analyzed with LOLITA (Large-scale Object-based Linguistic Interactor, Translator and Analyzer), a large-scale NL processing system. Both, the knowledge in the documents and ones that are already stored in the Knowledge-Base of LOLITA are then proposed to be used for producing requirements models. The approach was based on the consideration that requirements are often written in unrestricted NL and in many cases, it is impossible to impose customer restrictions on the language used. The object-oriented modelling module implemented an algorithm that filters entity and event nodes in its knowledge base to identify classes and associations.
> In parallel, Ninaus et al. [38] proposed a method to reducing the risk of low-quality requirements through improving the support of stakeholders in the development of RE models. The authors introduced the INTELLIREQ environment. This environment is based on different recommendation approaches that support stakeholders in requirements-related activities such as definition, quality assurance, reuse, and release planning.
> ”

3.6.4 Regression tests

According to Kung et al. [10], regression testing must address four fundamental problems:

- How to automatically identify those items that have been affected due to a change in some other items?
- What strategy should be used to retest these affected items?
- What are the coverage criteria for retesting these items?
- How to select, reuse and modify existing controls (and create new ones)?

Solutions to these problems for traditional programs have been proposed over the past two decades. However, not much attention has been given to the control of object-oriented programs, and almost none to regression. This is for three main reasons, according to the author:

First, traditional approaches do not address the complex relationships and dependencies, such as inheritance, aggregation and correlation, that exist between classes. Second, most traditional approaches are based on the control flow model, but class objects have state-dependent behavior that can change in various ways. Therefore, traditional approaches cannot be applied to class control. Third, traditional approaches use stubs to simulate the units being called, but in object-oriented programs this is difficult and expensive because it requires understanding many related classes, their methods, and how those methods are called.

In Tanejaa et al. [17], linear regression on java source code files is used to produce the minimum set of checks to be performed to deal with the vast majority of possible errors.

The algorithm looks at many software evaluation metrics, some of which we have already mentioned (like LCOM) and evaluates how much each unit ”contributes” to the probability of a bug. The WAKA software produces a primary equation in which each metric is a parameter. Replacing each parameter with the metric of each software produces a number that evaluates the suitability of a control to find bugs.

Finally, we rank each sequence of tests by the above value and construct a table (Table 3.6.4) of the most effective tests. In the example below these would be TS3, TS4, TS5, TS6, TS7, TS10.

The authors conclude that ”it can easily be seen that almost 80% of the errors have been covered in 60% of the total time. Therefore, it can be concluded that the number of errors detected per unit time is better with the proposed methodology.”

To assess the advantages of this method and how AI enhances them, we can compare it with a more traditional method.

Figure 3: The execution plan of the Tanejaa et al. [17]

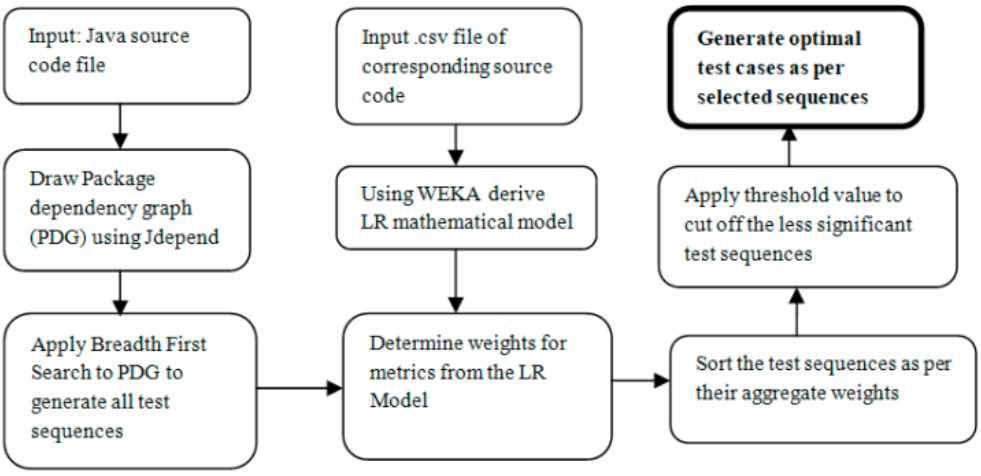

Figure 4: The program's results for each test sequence

Test Sequences ID	Nodes Covered	Weight
TS_1	1,6	4.7438
TS_2	1,5	4.4612
TS_3	1,4	10.0074
TS_4	2,1	10.6780
TS_5	2,1,4	**16.3882**
TS_6	2,1,5	10.8420
TS_7	2,1,6	12.2246
TS_8	3,5	1.4982
TS_9	4,5	5.8742
TS_{10}	4,1,6	10.5540

Kung et al. [10] use mathematical solutions, mainly based on graph theory, to construct algorithms that identify dependency relationships between program elements and thus which controls should be changed. The theory and algorithms used and the methods applied are beyond the scope of this paper and so we refer the reader to read the report for themselves.

It becomes clear that this approach, although theoretically more reliable, lags behind the automation capability (and ease of automation) compared to the alternative solution above. This is probably the reason why this research ultimately failed to produce a practical result in the object-oriented field.

3.7 Automation

By far the most common and often most practical way to facilitate verification of object-oriented programs remains the automation of tests. This is essential for delivering a good quality product at the right time.

Test automation is software that automates any aspect of testing an application system with the ability to produce test inputs and expected results. It reduces the repetitive tasks that programmers otherwise do manually and also provides them with a clear result(”success” or ”failure”).

According to Tauro, Ganesan, and Ghosh [18], the appropriate extent of automated testing depends on:

> “
> [...] testing goals, budget, software process, kind of application under development, and particulars of the development and target environment. Automated tests ensure a low defect rate and continuous progress, whereas manual tests would very rapidly lead to exhausted testers.
> ”

The author goes on to sepcify guidelines which tests ought to follow in order to be considered ”automated”:

> “
> To summarize the characteristics of tests we are aiming at:
>
> - Tests run the system – in contrast to static analyses.
> - Tests are automatic to prevent project members to get bored with tests (or alternatively to prevent a system that isn‘t tested enough).
> - Automated tests build the test data, run the test and examine the result automatically.
> - Success resp. failures of the test are automatically observed during the test run.
> - A test suite also defines a system that is running together with the tested production system. The purpose of this extended system is to run the tests in an automated form.
> - A test is exemplar. A test uses particular values for the input data, the test data.
> - A test is repeatable and determined. For the same setup the same results are produced.
>
> ”

But where automation truly stands out, and the reason why it is widely (almost universally) used in software verification, is that using it almost all the techniques mentioned in this paper can be applied - as long as they are not computationally difficult. Traditional unit tests, random data tests and regression checks can easily be run at regular intervals without programmer intervention.

4 Conclusion

The modern era, in which software development is guided by the object-oriented model, has led to a change in both the techniques and tools used to verify its software. Techniques based on data generation or mathematical verification of tests are rarely chosen, unlike other development models such as in fucntional programming, while simple techniques such as unit testing are used extensively with the help of automation tools. Research in this area also points towards producing correct tests through proper design and architecture, rather than finding some optimal way for testing. In any case, due to the rapid evolution of the field, the relatively nascent research around it and the continuous development of other tools (e.g. AI), new ways and tools leading to more efficient and easy-to-use check generation may be discovered in the near future.

References

[1] S. Barbey and A. Strohmeier. "The problematics of testing object-oriented software". In: *Swiss Federal Institute of Technology, Computer Science Department, Software Engineering Laboratory, EPFL-DI-LGL* (1994).

[2] Paul Bashir I. "Object-oriented integration testing". In: *Annals of Software Engineering* (1999). DOI: https://doi.org/10.1023/A:1018975313718.

[3] Feras A. Batarseh et al. "Chapter 10: The Application of Artificial Intelligence in Software Engineering – A Review Challenging Conventional Wisdom". In: *Nexus of Artificial Intelligence, Software Development, and Knowledge Engineering* (2021). URL: https://arxiv.org/abs/2108.01591.

[4] Briand L. C., J. Feng, and Labiche Y. "Using genetic algorithms and coupling measures to devise optimal integration test orders". In: *Proceedings of the Fourteenth International Conference on Software Engineering and Knowledge Engineering* (2002).

[5] I. Ciupa et al. "On the number and nature of faults found by random testing." In: *Special Issue:ICST 2008, the First IEEE International Conference on Software Testing, Verification and Validation* 21 (1 2011). DOI: https://doi.org/10.1002/stvr.415.

[6] H. Gong and J. Li. ""Generating Test Cases of ObjectOriented Software Based on EDPN and Its Mutant". In: *9th International Conference for Young Computer Scientists* (2008).

[7] Jamie S. Gordon and Robert F. Roggio. "A Comparison of Software Testing Using the Object-Oriented Paradigm and Traditional Testing". In: *Proceedings of the Conference for Information Systems Applied Research San Antonio, Texas, USA* 67 (2013).

[8] Hayes and Jane Huffman. "Testing of Object Oriented Programming Systems (OOPS): A Fault-Based Approach". In: *Notes in Computer Science* 858 (1994).

[9] Khatri et al. "Analysis Of Factors Affecting Testing In Object-oriented Systems". In: *International Journal On Computer Science And Engineering 3* (2011).

[10] David C. Kung et al. "On Regression Testing of Object-Oriented Programs". In: *Journal of Systems and Software* 32 (1 1996).

[11] William M. McKeeman. "Differential Testing for Software". In: *DIGITAL TECHNICAL JOURNAL* 10.1 (1998), pp. 100–107.

[12] Farid Meziane and Sunil Vadera. *Artificial Intelligence Applications for Improved Software Engineering Development: New Prospects*. 2009. DOI: 10.4018/978-1-60566-758-4.ch014. URL: http://usir.salford.ac.uk/id/eprint/2208/.

[13] Dewayne E. Perry and Gail E. Kaiser. "Adequate testing and object-oriented programming." In: *Journal of Object-Oriented Programming* (1990).

[14] *QuickCheck official page*. https://hackage.haskell.org/package/QuickCheck.

[15] Wappler S. and Wegener J. "Evolutionary unit testing of object-oriented software using strongly-typed genetic programming." In: *Proceedings of the Eighth Annual Conference on Genetic and Evolutionary Computation* (2006).

[16] Chandra Mani Sharma, Rabins Porwal, and Deepika Sharma. "Testing Object Oriented Software: Issues, State-of-the-art and Future". In: *Journal of Systems and Software* 32 (2013).

[17] Divya Tanejaa et al. “A Novel technique for test case minimization in object oriented testing.” In: *International Conference on Computational Intelligence and Data Science* (2019).

[18] Clarence Tauro, Nagesswary Ganesan, and Anupam Ghosh. “Testing Object-Oriented Software Systems: A Survey of Steps and Challenges”. In: *International Journal of Computer Applications* 42 (2012).

[19] Yeresime et al. “Effectiveness of Software Metrics For Object-Oriented System”. In: *Procedia Technology, 2nd International Conference on Communication, Computing and Security* 6 (n.d).