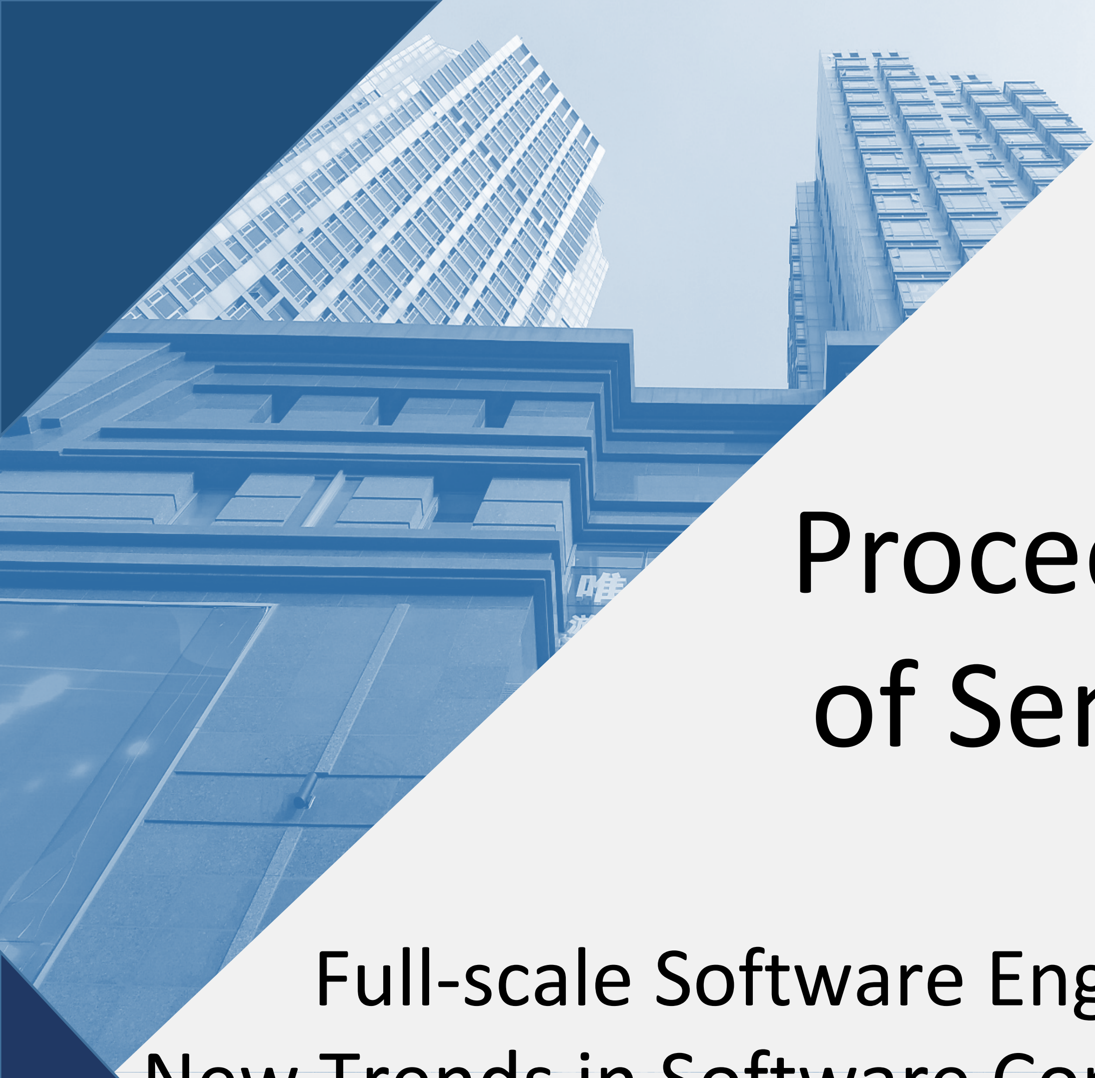


Proceedings of Seminars

Full-scale Software Engineering
New Trends in Software Construction

2020

Editors: Horst Lichter
Peter Alexander
Konrad Fögen
Christian Plewnia
Nils Wild

Software Construction

RWTH AACHEN UNIVERSITY

Table of Contents

Applicability of Test Oracles in Agile Development

Anna Vaassen
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
anna.vaassen@rwth-aachen.de

Adrian Azemi
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
adrian.azemi@rwth-aachen.de

ABSTRACT

While developing complex software, it is nearly unavoidable to cause bugs. Those, however, have to be identified to ensure functional software. One critical activity for finding bugs is structured testing. Thus, methods to generate test inputs and validate test outputs are needed to assure a complete testing. Concerning validation, these methods are provided by test oracles. Since every test oracle strategy has different requirements for the underlying development process, not every test oracle is equally suitable for agile development methodologies. Therefore, choosing an appropriate approach is an essential factor for successful testing. For reaching the optimal decision, an objective analysis based on the right measures has to be conducted. This paper provides suitable criteria and uses these to evaluate three different, commonly used test oracle strategies.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.2.9 [**Software Engineering**]: Testing and Test Verification—*test orcales, agile development*

Keywords

Model-Based Testing, Metamorphic Testing, Fuzzing, Agile Development, Test Oracle

1. INTRODUCTION

In the last decade, most companies migrated from the traditional software development method, also known as the waterfall model, to agile approaches [20]. In the traditional approach the software project is divided into fixed phases that have to be processed sequentially, whereby reaching the next phase requires all prior phases to be completed. Additionally, repeating a finalized phase is prohibited. On the contrary, agile approaches allow simultaneous processing and the repetition of completed phases [5, 16]. As a result of the migration, productivity and software quality increased [35]. Essential for ensuring software quality is extensive testing. Testing consists of creating appropriate test inputs and evaluating the behaviour of the system. Test oracles address the second issue by providing different methods to validate tests. However, formerly used test oracles that were suited for traditional software development methodologies may be insufficient for agile approaches. In traditional software development processes, the environment is considered static and, therefore, predictable. In agile development, the goal is to be flexible to react to changes and therefore dynamically. In contrast to traditional methods, this keeps the process iterative [16]. This is one of many aspects that should be considered when selecting a test oracle in an agile environment. Further differences are listed in Table 1. Thus, criteria need to be defined that regard the most important aspects. These criteria will later help to evaluate the applicability of test oracles. This paper defines applicability as follows: ”the quality of being relevant or appropriate [38]”.

The paper is structured as follows. First, the theoretical background of the concepts of agile development and test oracles are formally delivered to the reader to establish a common terminology. Notably, various categories of test oracles are introduced to the reader. In Chapter 3, the requirements of test oracles in agile environments are discussed. Based on that, specified criteria are used to evaluate whether a test oracle is suitable for the application in an agile environment. Subsequently, in Chapter 4, an evaluation of a representative test oracle of every category introduced in Chapter 2 is conducted. This is done with the defined criteria in section 3. As a basis for that, a definition and illustrating examples of the test oracles precede the evaluation. Afterwards, the results of the evaluation of each criterion for every given test oracle is considered as a whole to discuss their applicability in an agile environment. Finally, this paper is completed by analyzing in which way future work could extend results and a summary.

2. BACKGROUND

2.1 Agile Development

Agility is a broad term meaning that something is ”flexible and responsive” [12]. Due to the broad meaning, multiple agile research areas and methods exist [20]. Since explaining every agile method and its characteristics would go beyond scope, only agility in general will be discussed.

Every agile method is designed in conformity with these four principles of the agile manifest, which state [6]:

1. Individuals and interactions over processes and tools

Table 1: Traditional and agile perspectives on software development [16]

	Traditional view	**Agile perspective**
Design process	Deliberate and formal, linear sequence of steps, separate formulation and implementation, rule-driven	Emergent, iterative and exploratory, knowing and action inseperable, beyond formal rules
Goal	Optimization	Adaption, flexibility, responsiveness
Problem-solving process	Selection of the best means to accomplish a given end through well-planed, formalized activities	Learning through experimentation and introspection, constantly reframing the problem and its solution
View of the environment	Stable, predictable	Turbulent, difficult to predict
Type of learning	Single-loop/adaptive	Double-loop/generative
Key characteristics	Control and direction Avoids conflicts Formalizes innovation Manager is controller Design precedes implementation	Collaboration and communication; integrates different worldviews Embraces conflict and dialectics Encourages exploration and creativity; opportunistic Manager is facilitator Design and implementation are inseparable and evolve iteratively
Rationality	Technical/functional	Substantial
Theoretical and/or philosophical roots	Logical positivism, scientific method	Action learning, John Dewey's pragmatism, phenomenology

2. Working software over comprehensive documentation
3. Customer collaboration over contract negotiation
4. Responding to change over following a plan

Definition 1. *[Agility] Every method that respects the agile manifesto can be called agile [42].*

In contrast to other engineering disciplines, there are no routines that will most likely lead to desirable software [42]. This problem gets addressed by the first and fourth principle which state that no scheme should dominate the engineering process in general. This seems to be reasonable because not all requirements are known at the beginning of a project and they change [42].

Change is also a reason why documentation is less valued in agile methods than working software. Documenting something that changes persistently causes overhead. The lack of documentation is compensated by frequent interaction with the individuals participating in the project [19]. Working software, even in the form of prototypes, is the basis for that as it is a mean for receiving feedback [19].

The necessity of the third principle also results from the lack of knowledge of all requirements. It is illogical to be legally obliged to implement something with regards to outdated requirements. Instead, the idea is that all stakeholders should work in a team merging their knowledge, which allows to identify necessary requirement changes as soon as possible [19].

2.2 Test oracles

2.2.1 Definitions

The definitions below are adopted from the work of *Li et al.* [25], as they are precise enough to establish a common terminology without the overhead of more formal definitions.

Definition 2. *[Output] An output includes everything sent to a screen, written to a file or database, or sent as messages to a separate program or hardware device.*

Definition 3. *[Test Input] Test inputs consist of method calls to a system under test (SUT) and necessary input values.*

Definition 4. *[Expected Results] The results that will be produced when executing the test inputs if the program satisfies its intended behaviour.*

Definition 5. *[Test Oracle] A test oracle provides expected results for some program states as specified by the test oracle strategy (formally defined later). The test oracle determines whether a test passes by comparing expected with actual results.*

2.2.2 Categories of test oracles

Determining whether the output of a SUT corresponds expectedly to the input is a hard problem in general. For solving this problem various approaches have been developed [25].

A *specified test oracle* is designed by specification using mathematical logic if possible. One approach is to model the behaviour of a SUT mathematically using a specification language. The test oracle can then use the specification to verify the output. However, it is not possible to model a system completely, thus less important properties of the system usually aren't captured by the model. The closer the specification of the model is to the application, the more application specific failures can be recognized but the less reusable the test oracle is [33, 3].

Derived Test Oracles decide if an executed test on a SUT behaves expectedly or unexpectedly based on already existing information. These can be derived from documentation, system execution or different software versions [3].

Implicit Test Oracles distinguish correct from incorrect behaviour by using implicit knowledge. This implicit knowledge depends on the domain where the system is used in. In many domains all types of crashes like "buffer overflows" are incorrect behaviour, however not necessarily in all. Typically, only a few classes of failures are covered by this kind of oracles [33, 3].

2.3 Related Work

Lindvall et al. [27, 28] already applied Metamorphic Testing in an agile environment. *Barr et al.* and *Pezzè et al.* created a survey for the test oracle problem [3, 33]. *Barr et al.* also did a trend analysis of solutions to the test oracle problem [3]. *Coutinho et al.* conducted a mapping study to investigate how to develop requirements in the agile context of software testing [14].

Our work, in contrast, focuses on which type of test oracle to choose when agile methods are used in software development. To our knowledge, no related work with a similar aim exist.

3. AGILE CRITERIA

To evaluate the usability of test oracles in agile processes later, it is necessary to identify the essential characteristics of agile methods. To achieve this, the following section analyses and describes characteristics valued in agile development and defines four criteria that will later help to estimate the practicality of oracles in this field.

In Agile development, one of the key factors is to be adaptive [1]. In the early stages of development, most requirements are unknown to the development team and the customers. Instead, they need to adapt over time. This lack of information causes conditions to change rapidly during the development phase. Even very late into the process, changes can and will occur [6]. Therefore, the developers need to be able to react fast to upcoming changes. Accordingly, the team needs the used test oracles to be flexible and constructible from a little number of known requirements (see **C1,C2** below).

Another critical factor in agile development is the ambition to deliver working software to the customer regularly [6]. To achieve this, the development of new functionalities within the actual code is essential. Since time is limited in the process, documentation is considered less crucial for the development process. Thus, executable code is the primary concern of the development team, while exhaustive documentation is not part of agile development [1, 13]. However, many test oracles need complete documentation to be used for an effective result verification [3]. Therefore, one criterion for the test oracle demands the ability to operate with little to no documentation (see **C3** below).

Moreover, due to the ambition to deliver working software periodically, the process is done incrementally. This means the software is released in small portions in a fixed amount of time [1]. In this period, the code needs to be written, changed if new requirements come up and it also needs to be tested [1, 13]. As a result, time is precious and demands a tight schedule. In this scheme, there is little time to learn new and sophisticated tools or strategies. Therefore, methods used in agile development must be straightforward. They need to be easy to learn and quick to modify [1]. Subsequently, test oracles that are complex and hard to adjust are not ideal in agile development (see **C4** below).

For test oracles that are used in agile development, the following criteria can be derived from these characteristics:

C1 How feasible is the construction of the oracle to unknown requirements?

C2 How adaptable is the test oracle to fast-changing requirements?

C3 How much documentation does the oracle need?

C4 How time-consuming is the creation of the test oracle?

4. EVALUATION

In this section, three different test oracle strategies are chosen to be analyzed, namely, Metamorphic Testing, Model Based Testing and Fuzzing. These three were selected to each represent one of the three test oracle categories. For Specified Test Oracles Model Based Testing was elected, since the graph structure makes it intuitive to understand. Metamorphic Testing was chosen for Derived Test Oracle strategies, as it is used often in practice [29, 11, 43]. Lastly, Fuzzing was selected for Implicit Test Oracles due to being the most known of this category [3].

4.1 Metamorphic Testing

The term *Metamorphic Testing* (MT) has first been introduced by *Chen et al.* [9] in 1998. Since then, this strategy has been used often in both the academic field and in industry [29, 11, 43]. One reason for this successful integration is the high fault detection rate of MT. It has found numerous bugs in already established and tested software such as Siemens Suits [21], C Compilers like the GNU Compiler Collection (GCC) and LLVM a compiler framework [26, 34] and in Google Maps [37]. Another reason for the high acceptance of MT is that it can derive information from faulty test cases as well as from successful ones. While in all strategies, faulty test cases indicate that the SUT might be flawed or at least does not behave the expected way. Contrary, some test strategies often consider successful test cases as useless because they only validate one specific configuration [10, 37].

The basis and crucial point of MT is the definition of so-called *Metamorphic Relation(s)* (MR). These are predefined relations between two or more inputs and their corresponding outputs. Thus, MRs describe the fundamental properties of the SUT [10, 37]. By checking against these characteristics, MT goes beyond testing of individual test cases. Often MRs are represented by mathematics formulas.

For example, the algorithm G awaits two nodes a, b and a graph g and should solve the shortest path problem. If this algorithm is supposed to be verified, the following MRs could be derived [40]:

MR1 $G(g, a, b) = G(g, b, a)$
When the shortest path between the nodes a and b is found, the path should not change if we switch the nodes. The start becomes the node b and the destination is changed to a.

MR2 $|G(g, a, b)| = |G(g, a, x)| + |G(g, x, b)|$ for all x on the path between a and b.
If we find the shortest path between the nodes a and b, for any other node x on this path, we can calculate the shortest path between a and x as well as b and x. Then the length of the sum of both paths must be equal to the shortest path from a to b

If these relations are not fulfilled, the algorithm is erroneous.

The general procedure of MT is shown in figure 1 and works as follows: After defining a MR for a system, it has to be checked if the current implementation of the system fulfills this relation. At the beginning the so called *source*

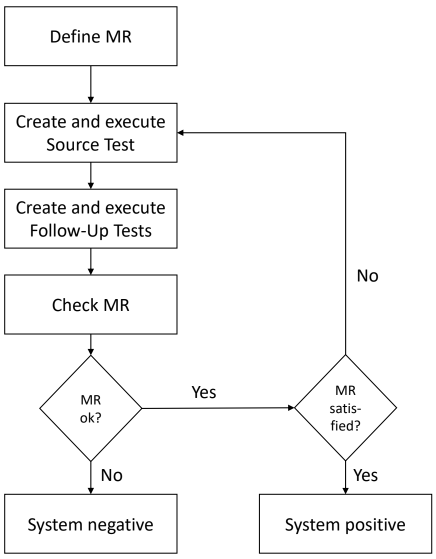


Figure 1: Process of Metamorphic Testing

test is created and executed. The source test is the first part of the MR (e.g. the left side of a mathematical formula). By creating a test scenario and calculation of the source case this first step is done. Next, the *follow-up tests* are performed (the other side of the MR e.g. the right side of a mathematical formula) with the same test scenario as used for the source test. With the results of source and follow-up tests the MR is checked to be true. If the MR is not fulfilled, the SUT does not achieve the desired behaviour and the implementation has to be rechecked. If the MR is tested positive, a new test scenario can be created till a predefined timeout is reached and the MR is considered satisfied [36, 24, 40].

For example, the source test of **MR2** is the generation of a random graph and the calculation of the shortest path between a and b (left side of equation $|G(g, a, b)|$). The follow-up tests are the calculation of the shortest paths between a and x as well as x and b (right side of the equation $|G(g, a, x)|$ and $|G(g, x, b)|$). To check against the MR the sum of the follow-up test cases is compared with the result of the source test case. The two possible results here are that the MR test is positive or negative. If the result is negative, the SUT is faulty and has to be rechecked. However, if the test is positive, either the SUT could be correct, or the erroneous results of source and follow-up tests nullify each other. The latter one is statistically highly unlikely, especially under the condition that multiple executions verify the MR. It is interesting to see that even if the distance between a and b would be calculated correctly the SUT could be erroneous anyway. This is a good example to demonstrate that MT doesn't stick to single testcases but checking against global system characteristics.

4.1.1 How feasible is the construction of the oracle to unknown requirements?

The key factor in the successful execution of MT is the identification and definition of MRs. Without them, not only the generation of follow-up test cases but also the verification of the system is impossible. To define the MRs the essential functions of the system need to be known. Therefore, testers need to know how the system is supposed to behave under specific conditions. However, this is only possible if the requirements are known to the testers since they define the general behaviour of the SUT [10].

Nevertheless, MRs can be especially useful in the early stages of software development, as they can improve communication between developers and customers. MRs are often easy to understand and jet very specific. Hence enabling a great understanding on the customer's side, who will be able to confirm or deny that the MR describes a needed requirement of the system. Since MRs should be described formally they do not have the downside of being misunderstood and can be adapted into the implementation more easily [36, 10].

4.1.2 How adaptable is the test oracle to fast-changing requirements?

If a requirement of a system is changed in mid-development, the behaviour of the system might change as well. This change could lead to already identified and defined MRs to no longer be true for the SUT. For example, the algorithm above can be changed to now not calculate the shortest path in an undirected graph but a directed graph. Due to this change, the first MR (**MR1**) is no longer correct and needs to be reworked. As this example shows, if requirements change, so can the related MRs. Therefore, all MRs have to be checked to decide if they need to be removed or redefined. Moreover, new MRs might be needed. These have to be identified and defined. Nonetheless, as seen in the example, not necessarily all MRs become useless by a change in requirements. In the example, the second MR (**MR2**) can be kept. This property is also likely in bigger systems with more defined MRs so that only small amounts need to be adjusted. Which only leads to a small overhead. Furthermore, the identification of MRs that do need to be adjusted and the definition of new MRs should be an easy task, due to the close contact with the customer who can help to identify those[10].

4.1.3 How much documentation does the oracle need?

Since MT can be used for different tasks, MRs have to be derived depending on the intended task of the MT. If the MT is supposed to verify, the MRs need to be derived based on software specifications. If the task is to validate, testers define MRs based on the expectations of the user and the customers. Lastly, if the MT should help with quality insurance, MRs can be derived by various stakeholders. Thus, limited documentation is needed to derive the necessary MRs and even without complete specification, testers should be able to identify the needed MRs [37, 10].

Yet, effective MRs are essential for the successful execution of MT. Therefore, a systematical method to find good MRs is useful in the process. As it can be difficult to find suitable MRs for testers without expertise and experience. [10] Though there is no specification needed to identify the MRs it is reasonable to still withhold a level of formalism when describing MRs. When MRs are described in natural language, often misunderstandings arise due to misinterpretation. Moreover, it is harder to reuse MRs if they are not defined specifically. Thus, using predicate logic when defining the MRs can help the effectiveness [36].

4.1.4 How time-consuming is the creation of the test oracle?

MT is a simple concept that is still highly effective. Further, the implementation is straightforward and the process can be automated to a certain extent. The generation of different test cases, as well as the execution of MT and the verification, can be automated quickly and with little effort. However, the identification of MRs normally needs to be done manually. Since it demands complete knowledge of the program and its domain. Although alternative ways to generate MRs like combining existing relations [30, 15] and generating MRs automatically [23, 22] have been proposed. Those techniques still need further investigation before they can be entrusted. [36] Another way to ease and fasten up the MR identification process would be guidelines to find good MRs. Though, this as well needs further investigation [36, 10]. As already explained above, the MRs have to be adjusted if the requirements of the system change. This adjustment of the MRs leads to overhead on the tester's side and does need time. Nonetheless, as also explained before, this overhead is relatively small, which means that the time consumption is also limited [10].

4.2 Foundations

Definition 6. *[Graph] A graph G is [25]:*

- *a set N of nodes, where $N \neq \emptyset$*
- *a set N_0 of initial nodes, where $N_0 \subseteq N$ and $N_0 \neq \emptyset$*
- *a set N_f of final nodes, where $N_f \subseteq N$ and $N_f \neq \emptyset$*
- *a set E of edges, where $E \subseteq N \times N$*

Definition 7. *[Mapping] A mapping is a data structure that describes how to translate test inputs from model elements (transitions and state invariants in this research) to the implementation [25].*

Definition 8. *[Test case] A test case is a finite structure of input and expected output [41].*

Definition 9. *[Test suite] A test suite is a finite set of test cases [41].*

Definition 10. *[Test input] The input part of a test case is called test input [41].*

Definition 11. *[Test output] The output part of a test case is called test output [41].*

4.3 Model-Based Testing

The following material including the given example is adopted from the work of *Li et al.* [25]. In model-based testing the SUT gets specified by a model representing the system's behaviour. [25]. However, due to complexity reasons, usually only the most relevant properties get captured by the model [3]. The same principle, already described for specified oracles in general, also holds for model based testing. The closer the specification of the model is to the application, the more application specific failures can be recognized [33, 3].

Often, the abstract model is represented by a graph structure like UML state machine diagrams. Its nodes represent states of the SUT and the edges represent transitions. By taking transitions, which is an abstract term for executing a function on the SUT, the state changes in general. Each state gets a state invariant assigned, which requires a certain condition to be satisfied. Typically, this condition ensures expected behaviour of the system. After that, tools like STALE are used to convert the generated more complex graph structure, for example a state machine diagram, to a plain graph structure specified in definition 6 [2, 25].

After converting, abstract test suites can be created following a specific coverage criterion. If, for example, the coverage criterion is edge coverage, which requires all edges to be covered, every test suite has to cover all transitions from the start of the initial node to the end when a final node is reached.

These abstract test cases have to be mapped to concrete test cases in order to draw conclusions for the behaviour of the SUT. If the specification of the model is powerful enough and includes information about how to transform the abstract tests to concrete tests or it even provides expected output values, the problem becomes trivial, as the testing process would be completely automated [25]. Therefore. we assume in the following the mappings and the expected output values have to be provided manually, as it is usual in practice when such formal specification isn't available [25, 3].

As an example, consider the following extract of the class vending machine as the SUT and a state machine diagram as its model, assuming that only dimes, quarters and dollars are accepted and the price for all products is 90 cent:

```java
public class VendingMachine
{
    //Current credit in the machine
    private int credit;
    ...
    //Constructor: vending machine
    //starts empty.
    public VendingMachine() {}

    // A coin is given to the vendingMachine
    // Must be a dime, quarter or dollar.
    public void coin (int coin) {}

    //Get the current credit value.
    public int getCredit () {}
    ...
}
```

Listing 1: Class VendingMachine (partial) [25]

In that example the abstract model is the state machine diagram. Constraint 1 is the state invariant of state 1, Constraint 2 is the state invariant for state 2 and Constraint 3 is the state invariant of state 3. The node "State 1" is in the set of initial nodes and "State 3" is in the set of final nodes. If edge coverage is chosen as a coverage criterion, every transition of the enumeration: (State 1, coin, State 2), (State 1, coin, State 3), (State 2, coin, State2), (State 2, coin, State 3) has to be covered by at least one test case of the generated test suite. For the model of the vending machine in Figure 2 paths from an initial node to a final node can be interpreted as test cases and ((State 1, coin, State 3)), (State 1, coin, State 2), (State 2, coin State 2),

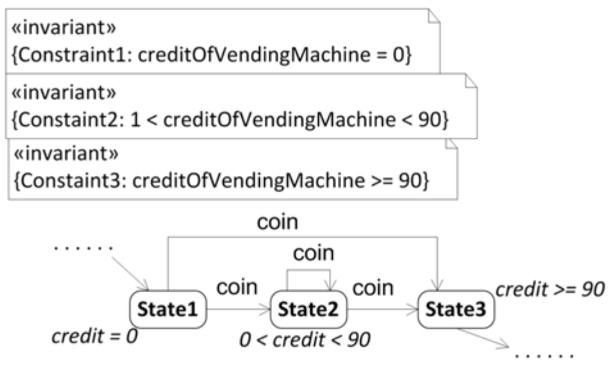

«invariant»
{Constraint1: creditOfVendingMachine = 0}
«invariant»
{Constaint2: 1 < creditOfVendingMachine < 90}
«invariant»
{Constaint3: creditOfVendingMachine >= 90}
coin
coin
coin
coin
State1
State2
State3
credit >= 90
credit = 0
0 < credit < 90

Figure 2: Model for the class VendingMachine [25]

(State 2, coin State 3)) can be considered as a test suite. Using a transition in the abstract model symbolizes the call of the method coin in the concrete system. A mapping for the transition between state1 and state 3 could be for example the test code ”vm.coin(100)” which represents the insert of a dollar, where vm is an object of the class VendingMachine to be tested.

Since STALE selects appropriate mappings for a test automatically, this particular mapping can be used for the transition between State2 and State3 either, but has to be provided only once. Note that it is still possible and even useful to provide more than one mapping for one transition. With regards to the vending machine example, if no further mappings apart from ”vm.coin(100)” are provided, some tests will fail, as for edge coverage the transition (State 1, coin, State 2) needs to be tested either. If not enough mappings are provided to satisfy all tests related to a coverage criterion, a tool like STALE will notify the user to provide more mappings. It is also possible to check properties that differ from the state invariant, for example, by extending the mapping in Listing 2 in the following way to ensure that the credit gets exactly increased by the expected amount:

```
int creditCopy = vm.getCredit;
vm.coin(100);
assertEquals(creditCopy + 100, vm.getCredit());
```

Listing 2: Example Mapping [25]

4.3.1 How feasible is the construction of the oracle to unknown requirements?

When the requirements are completely unknown, the construction of the OS is not robust at all, since model based testing needs a model that specifies the systems behaviour. However, if enough requirements are known so that a rough notion about the future system exist, one approach to deal with that issue is creating an under specified model, which can be refined from time to time [33, 17]. Agile development methods usually use iterative approaches as well, in which systems evolve in every iteration [17]. One advantage of using this approach is that under specified models may be reused for future developments of systems with similar properties [33].

4.3.2 To what degree is the test oracle adaptable to fast-changing requirements?

When requirements change rapidly the model needs to be adjusted to the changed requirements and since the mappings rely on the model, they need to change as well. Nevertheless, the workload of changing a model and its mapping is not constant but depends on the degree of change of the model. If, for example, the requirements change fast but the changes are rather small, then the workload will be lower in comparison to changes of requirements that require a complete restructuring of the model.

4.3.3 How much documentation does the oracle need?

As a model of the system needs to be constructed for the creation of the test oracle, at least enough documentation for a rough model is necessary. However, even if such a minimal amount of documentation doesn’t exist, the model itself can serve as a kind of documentation between all the stakeholders. Using state diagrams, technical details can be abstracted away at first allowing even non technically versed people to be able to participate, which supports the first and third principle of the agile manifesto.

4.3.4 How time-consuming is the creation of the test oracle?

When model-based testing is used for the generation of a test oracle, the time needed for the creation of the test oracle depends on the tools used and the number of assertions that have to be provided by hand [25]. By using tools that offer prefix graph based solution like STALE, the number of tests can be reduced [25]. Considering the graph in figure one, in one scenario the edge between state 1 and state 2 can get checked ten times by ten separate tests, whereas in a more efficient scenario a single test can cover the edges (State1, coin, State2), (State2, coin, State2), (State 2, coin State 3) at once.

Furthermore, not only the number of tests but also the complexity of the tests, consisting of the number of transitions that appear in a single test, can be reduced. Moreover, to create the test oracle it is necessary to transform the abstract test cases to concrete test cases using mappings. For the creation of a mapping assertions have to be provided by hand. That, however depends, on the coverage criterion and the properties of a state that are checked. Counterintuitively, it has been shown that a stronger coverage criterion does not help for finding more errors [25]. Additionally, it turned out that only checking the state invariants of all states after every transition suffices [25]. Considering this aspects, it can be concluded that the time-consumption for the construction of the test oracle can be minimized.

4.4 Fuzzing

Fuzzing dates back to 1989 when Professor Barton Miller proposed his first paper on the matter. His approach was to test software by generating random input and throwing this at the SUT. If the program crashed, this was regarded as a failure. Where when the system did not collapse, the test was viewed as successful [31].

Since then, this method is considered to be an effective way to find real errors quickly. Though the main idea has not changed a lot, there has been room for improvement. The core of Fuzzing is still to generate random input, so-called ’fuzz’ and to test this fuzz against the SUT. Often the input examines the implementation limits and data boundaries. If the system reacts unexpected, both the fuzz and the respected output are monitored with the help of special tools. Allowing the tester to recheck the implementation. Unexcepted behaviour means crashes of the system or internal errors like buffer overflows or unhandled exceptions [4, 7, 39].

Since the general idea is fundamental and comfortable, there is no single method for Fuzzing. Therefore, the efficiency and the error detection highly depend on the creativity of the tester. Since there are no exact rules and no wrong approaches, the creativity of the examiners is not limited [7, 39]. However, three different strategies can be identified. Namely, Blackbox Fuzzing, Whitebox Fuzzing and Greybox Fuzzing [7, 18].

```
void algorithm1(int x)
{
    ...
    if(x == 15){
        f1(x);
    }else if (x == 10){
        f2(x);
    }else{
        f3(x);
    }
    ...
}
```

Listing 3: Example algorthim for execution of Fuzzing

In Blackbox fuzz testing, the idea stays very basic. The input is generated randomly and then applied to the SUT. Therefore, this technique requires no knowledge of the system itself. See for example, the algorithm in Listing 3. It awaits an 32-bit integer. Blackbox tests would generate random integer numbers and would check boundaries like $-1, 0$ and 2^{32} [7, 18]. Whitebox Fuzzing becomes more complicated, however. Here a complete knowledge of the system and its behaviour is needed to generate the test inputs. One method of Whitebox testing is presented by *Godefriod et al.* [18]. The idea is to execute the program symbolically and, while doing so, collecting conditional statements to modify the input. This process is repeated until either a specific limited is reached or all paths possible were checked. For example, the algorithm in Listing 3 could be tested with this technique. The program expects 32-bit integer for the first execution. First, a random input would be generated; for example, this could be $x = 1$. The program is then executed with this input. When the execution reaches the IF-Statement in the algorithm, the Whitebox test recognizes that and saves the constraint $x! = 15$. This statement is then negated and solved to $x == 15$. The new input $x = 15$ is then remembered to be tested as well later. The same procedure would be done with the second IF-Statement, remembering $x = 10$ as an input to be tested. Doing so guarantees the execution of a new path in the system. Since otherwise the chances of executing the functions $f1$ and $f2$ would be $\frac{1}{2^{32}}$. Therefore, in Blackbox Testing, it is likely, that the functions $f1$ and $f2$ would never be tested. Whitebox testing, on the other hand, guarantees the execution of $f1$ and $f2$, since it searches for these specific cases [7, 18].

The last strategy, Greybox Fuzzing, is a mixture of both Blackbox and Whitebox Fuzzing. Therefore, it tries to take advantage of the other two and needs a minimal knowledge of the target behaviour [7]. One commonly used method is the Coverage-Based Gerybox Fuzzing (CGF). Given a so-called seed file, the algorithm tries to generate new files out of the seed, by flipping bits and deleting or copying them. These are then run against the SUT and the behaviour is monitored. If they produce unexpected behaviour, the file is added to the seed list to generate new data for further inspection [8].

In literature and industry, Fuzzing is viewed as an effective method to detected security vulnerabilities in software like memory leaks or denial of service [39, 18].

4.4.1 How feasible is the construction of the oracle to unknown requirements?

Fuzzing relies on the execution of the SUT and the behaviour of the software under different input data. Since these steps can only be done when the software is already implemented, unknown requirements at the beginning of development do not affect the generation process of the Fuzzing data or the execution of the method. Moreover, in Blackbox testing, the input data is generated randomly and therefore needs no knowledge about the application itself, including the requirements [18, 7].

4.4.2 How adaptable is the test oracle to fast-changing requirements?

If requirements change, this could affect the format of the input data. Leading to a needed adjustment in the fuzzed data. However, since Fuzzing is a simple method and the generation of data is dynamic. Data only needs to be generated when the software needs to be tested, leading to only a small to no overhead for the generation of new input data. However, unexpected behaviour of one system might be acceptable or even intended in another system. A change in requirement, therefore, might include such a change in the system. For example, the new requirement could be that the system shuts down whenever getting an invalid input. Leading to test cases that crash because of that reason not being failures of the software at all. To handle these situations, the tools monitoring the failures can be adjusted not further to investigate if the input was invalid [18, 4].

4.4.3 How much documentation does the oracle need?

There is no general answer to this question, since different strategies need different levels of documentation. Something all methods have in common, however, is that they need no domain knowledge. As well as no formal specification to implement the Fuzzing strategy. Blackbox testing also needs only little knowledge about the software itself. Basically, testing blind for the most part. The only required information in Blackbox testing, is the format of the input the SUT excepts. Whitebox Fuzzing, however, needs complete access to the source code and different design specifications. As well as extensive knowledge of the input format. In conclusion, it requires more detailed documentation than Blackbox testing [4, 39, 32].

Another difference in need of documentation erases with the way how the test input is created. So-called fuzzers are used to derive the fuzz for the system. There are two different types of fuzzers the first one being Mutation-based and

Criteria	MT	MBT	Fuzz
C1	Req. needed to derive MRs	few req. needed to construct an (underspecified) model	Just few Req. needed
C2	MRs can be adapted easily	uncertain	Is done automatically
C3	Just few documentation needed	model can serve as documentation	Just few documentation needed
C4	Automated to certain point	mappings have to be provided manually. Tools can minimize time consumption	Highly automated

Table 2: Summarized results from evaluation

the second one Generation-based. Mutation-based fuzzers mutate existing valid and functional input samples to create test cases. Modifications are applied in a few key areas and are then submitted to the target. To use these fuzzers good data is needed. Generation-based fuzzers are based on specifications. These provide information on how the input should look. The specific test cases are then modeled from the system to test the limits and unexpected data. Leading to a higher need for documentation than the mutation-based fuzzers [39, 32].

4.4.4 *How time-consuming is the creation of the test oracle?*

Fuzzing is a testing technique that can be automated very easily. From the generation of the test input with different fuzzers to the motoring of the failed test cases in multiple tools, everything can be done automated. This automation enables the testers to test large applications with a lot of test data in a limited time. Therefore, Fuzzing is considered to be a cost-effective and fast method for detecting security vulnerability [18, 39, 32].

5. DISCUSSION

Up to this point, three different test oracles are presented as representatives of one of the previously mentioned test oracle category. In this scope, Metamorphic Testing (MT) belongs to Derived Test Oracles, Model Based Testing (MBT) to Specified Test Oracles and Fuzzing (Fuzz) to Implicit Test Oracles. This chapter aims to evaluate if the presented testing strategies are suitable for agile development. The results of the previous section are summarized in table 2.

Metamorphic Testing fits quite right with agile development. Testing is easy to adapt to changing requirements and the amount of required system documentation is manageable. Thus, it supports the agile principle of working code over excessive documentation. Just the MRs have to be modified manually and the other steps are performed automatically. Therefore, testing can be adopted quickly and flexible to the changing SUT. One difficulty is the identification of qualitative MRs with fewer requirements. To overcome this issue, the development team needs to have great expertise and frictionless collaboration with the customer is mandatory [10, 36]. Since it is used in agile development processes the potential of MT is further underlined [27, 28].

The advantage of a Model Based Test Oracle is that it can be integrated into an agile environment with minor adoptions and its construction is robust to unknown requirements at the beginning, using underspecification [17]. However, the amount of workload when requirements change is difficult to predict. Moreover, the used graph structure for the specification of the model, which represents the system's behaviour, can serve as a type of documentation, which allows stakeholders without much technical knowledge to participate. Additionally, the time consumption of the creation of a Model Based Test Oracle can be kept to a minimum when appropriate tools and an inexpensive coverage criterion are used and only state invariants are checked at the end of each transition. Ultimately, Model Based Test Oracles are only with restrictions suitable for an application in an agile environment, since it provides advantages. Still, the possible accompanying workload of a change of requirements can be critical.

Fuzzing fulfills all the derived criteria. As the evaluation indicates, the strategy is easy to adapt and highly automated. Moreover, it does not need extensive documentation which suits the agile development process. However, there are limitations to Fuzzing that need to be taken into consideration. One issue is the code coverage as Blackbox testing, where input is generated randomly can not grantee a complete code coverage. While Whitebox testing theoretically achieves comprehensive code coverage, in complex systems, this may not be the case. Due to long lines of code, checking every possible path can take very long. Therefore in practice, the Testing is stopped before a complete coverage is reached to keep the testing time reasonable [18]. Another issue of Fuzzing is the failure assumption. Since this strategy considers failures only as system crashes, many bugs, like wrong calculation that do not lead to crashes, go unnoticed. Consequently, Fuzzing can not be used as a standalone test strategy but should be used in combination with other methods. However, if the software needs testing for security vulnerabilities, Fuzzing should be applied, since the approach has proven to perform effectively in this field [18].

6. CONCLUSION

This paper aimed to evaluate the applicability of test oracles in agile development. To achieve this, four different criteria for test oracles in agile processes were derived and defined. These were used to analyze three test oracle strategies, namely Metamorphic Testing, Model Based Testing and Fuzzing. This breakdown came to the result that Metamorphic Testing suits agile development well and should, therefore, be considered in agile testing. Model Based Testing, however, is not so easy to adapt and needs more work to change into an agile process. Though the advantages arising from the use might overcome the obstacles, therefore before using Model Based Testing, cost-effectiveness analyses should be applied. While Fuzzing fitted best in the analyze the paper still concluded that the use of Fuzzing on its one is not efficient enough. Though there are fields in which Fuzzing is a very effective method.

To further extend the work of this paper, more test oracles form the categories Specified, Derived and Implicit Oracles could be evaluated. Therefore, allowing a more detailed and specific analysis of suitable oracle strategies in agile development. Moreover, different tests could be applied to validate the results. For example, a cost-effectiveness study could be done to examine the usability of Metamorphic Testing in this field further. Costs here mean the time needed to adjust the MRs to changing requirements. Another work that could be done in Metamorphic Testing is to establish guidelines for finding reasonable MRs. Identifying those can be a challenging and time-consuming task if never done before. For Model Based Testing, an analysis of the question: How to make Model Based Oracles more usable?; would be of great interest for the further applicability of these kinds of test oracles.

7. REFERENCES

[1] P. Abrahamsson, O. Salo, J. Ronkainen, and J. Warsta. Agile software development methods: Review and analysis. *arXiv preprint arXiv:1709.08439*, 2017.

[2] P. Ammann and J. Offutt. *Introduction to software testing*. Cambridge University Press, 2008.

[3] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The oracle problem in software testing: A survey. *IEEE transactions on software engineering*, 41(5):507–525, 2014.

[4] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The oracle problem in software testing: A survey. *IEEE Trans. Software Eng.*, 41(5):507–525, 2015.

[5] Y. Bassil. A simulation model for the waterfall software development life cycle. *CoRR*, abs/1205.6904, 2012.

[6] K. Beck, M. Beedle, A. Van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, et al. Manifesto for agile software development. 2001.

[7] S. Bekrar, C. Bekrar, R. Groz, and L. Mounier. Finding software vulnerabilities by smart fuzzing. In *Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2011, Berlin, Germany, March 21-25, 2011*, pages 427–430, 2011.

[8] M. Böhme, V. Pham, and A. Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 45(5):489–506, May 2019.

[9] T. Y. Chen, S. C. Cheung, and S. M. Yiu. Metamorphic testing: a new approach for generating next test cases. Technical report, Technical Report HKUST-CS98-01, Department of Computer Science, Hong Kong ..., 1998.

[10] T. Y. Chen, F.-C. Kuo, H. Liu, P.-L. Poon, D. Towey, T. Tse, and Z. Q. Zhou. Metamorphic testing: A review of challenges and opportunities. *ACM Computing Surveys (CSUR)*, 51(1):4, 2018.

[11] T. Y. Chen, T. H. Tse, and Z. Q. Zhou. Fault-based testing without the need of oracles. *Information and Software Technology*, 45(1):1–9, 2003.

[12] T. Chow and D.-B. Cao. A survey study of critical success factors in agile software projects. *Journal of systems and software*, 81(6):961–971, 2008.

[13] E. Collins, A. Dias-Neto, and V. F. de Lucena Jr. Strategies for agile software testing automation: An industrial experience. In *2012 IEEE 36th Annual Computer Software and Applications Conference Workshops*, pages 440–445. IEEE, 2012.

[14] J. C. S. Coutinho, W. L. Andrade, and P. D. L. Machado. Requirements engineering and software testing in agile methodologies: A systematic mapping. In *Proceedings of the XXXIII Brazilian Symposium on Software Engineering*, SBES 2019, pages 322–331, New York, NY, USA, 2019. ACM.

[15] G.-W. Dong, B.-W. Xu, L. Chen, C.-H. Nie, and L.-L. Wang. Case studies on testing with compositional metamorphic relations. *Journal of Southeast University (English Edition)*, 24(4):437–443, 2008.

[16] T. Dyba and T. Dingsoyr. What do we know about agile software development? *IEEE software*, 26(5):6–9, 2009.

[17] D. Faragó. Model-based testing in agile software development. *30. Treffen der GI-Fachgruppe Test, Analyse & Verifikation von Software (TAV), Testing meets Agility*, 2010.

[18] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California, USA, 10th February - 13th February 2008*, 2008.

[19] J. Highsmith and A. Cockburn. Agile software development: The business of innovation. *Computer*, 34(9):120–127, 2001.

[20] R. Hoda, N. Salleh, and J. Grundy. The rise and evolution of agile software development. *IEEE Software*, 35(5):58–63, 2018.

[21] M. Hutchins, H. Foster, and T. Goradia. omas ostrand. 1994. experiments on the effectiveness of data owand control ow-based test adequacy criteria. In *Proceedings of the 16th International Conference on So ware Engineering (ICSE'94). IEEE Computer Society, Los Alamitos, CA*, pages 191–200.

[22] U. Kanewala. Techniques for automatic detection of metamorphic relations. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*, pages 237–238. IEEE, 2014.

[23] U. Kanewala and J. M. Bieman. Using machine learning techniques to detect metamorphic relations for programs without test oracles. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, pages 1–10. IEEE, 2013.

[24] U. Kanewala and T. Y. Chen. Metamorphic testing: A simple yet effective approach for testing scientific software. *Computing in Science & Engineering*, 21(1):66–72, 2018.

[25] N. Li and J. Offutt. Test oracle strategies for model-based testing. *IEEE Transactions on Software Engineering*, 43(4):372–395, 2016.

[26] C. Lidbury, A. Lascu, N. Chong, and A. F. Donaldson. Many-core compiler fuzzing. In *ACM SIGPLAN*

Notices, volume 50, pages 65–76. ACM, 2015.

[27] M. Lindvall, D. Ganesan, R. Árdal, and R. E. Wiegand. Metamorphic model-based testing applied on nasa dat: an experience report. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*, pages 129–138. IEEE Press, 2015.

[28] M. Lindvall, D. Ganesan, S. Bjorgvinsson, K. Jonsson, H. S. Logason, F. Dietrich, and R. E. Wiegand. Agile metamorphic model-based testing. In *2016 IEEE/ACM 1st International Workshop on Metamorphic Testing (MET)*, pages 26–32. IEEE, 2016.

[29] H. Liu, F.-C. Kuo, D. Towey, and T. Y. Chen. How effectively does metamorphic testing alleviate the oracle problem? *IEEE Transactions on Software Engineering*, 40(1):4–22, 2013.

[30] H. Liu, X. Liu, and T. Y. Chen. A new method for constructing metamorphic relations. In *2012 12th International Conference on Quality Software*, pages 59–68. IEEE, 2012.

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

[32] P. Oehlert. Violating assumptions with fuzzing. *IEEE Security & Privacy*, 3(2):58–62, 2005.

[33] M. Pezzè and C. Zhang. Chapter one - automated test oracles: A survey. volume 95 of *Advances in Computers*, pages 1 – 48. Elsevier, 2014.

[34] J. Regehr. Finding compiler bugs by removing dead code. *blog*, 2014.

[35] D. J. Reifer. How good are agile methods? *IEEE software*, 19(4):16–18, 2002.

[36] S. Segura, G. Fraser, A. B. Sanchez, and A. Ruiz-Cortés. A survey on metamorphic testing. *IEEE Transactions on software engineering*, 42(9):805–824, 2016.

[37] S. Segura and Z. Q. Zhou. Metamorphic testing 20 years later: A hands-on introduction. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pages 538–539. ACM, 2018.

[38] A. Stevenson. *Oxford Dictionary of English*. Oxford University Press, 2010.

[39] M. Sutton, A. Greene, and P. Amini. *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[40] D. Towey, T. Y. Chen, F.-C. Kuo, H. Liu, and Z. Q. Zhou. Metamorphic testing: A new student engagement approach for a new software testing paradigm. In *2016 IEEE International Conference on Teaching, Assessment, and Learning for Engineering (TALE)*, pages 218–225. IEEE, 2016.

[41] M. Utting, A. Pretschner, and B. Legeard. A taxonomy of model-based testing approaches. *Softw. Test., Verif. Reliab.*, 22(5):297–312, 2012.

[42] L. Williams and A. Cockburn. Agile software development: it's about feedback and change. *IEEE Computer*, 36(6):39–43, 2003.

[43] Z. Q. Zhou, S. Xiang, and T. Y. Chen. Metamorphic testing for software quality assessment: A study of search engines. *IEEE Transactions on Software Engineering*, 42(3):264–284, 2015.

Current State of Equivalent Mutant Reduction Methods in Mutation Testing

Lukas Malcher
RWTH Aachen University
lukas.malcher@rwth-aachen.de

Kristian Lebold
RWTH Aachen University
kristian.lebold@rwth-aachen.de

ABSTRACT

Mutation testing presents an approach to evaluate a test suite by mutating the source code while checking if the mutations are detected by the test suite or not. This method is considered to be a more reliable metric in terms of indicating the quality of test suites than code- or branch coverage. The main drawback is the generation of equivalent mutants. There is no solution to the Equivalent Mutant Problem (EMP) yet, but approaches to minimize their occurrence do exist. This paper evaluates Weak Mutation Testing, Higher Order Mutants, Compiler Optimization, and Control-Flow analysis techniques. The results indicate that Weak Mutation Testing and Higher Order Mutant methods which avoid equivalent mutants perform better than Compiler Optimization and Control-Flow analysis which try to detect equivalent mutants. We observed that current mutation testing tools preferably use methods that avoid equivalent mutations.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.2.9 [**Software Engineering**]: Management—*productivity, programming teams, software configuration management*

Keywords

Mutation Testing, Equivalent Mutation Problem, Equivalent Mutant Reduction, Mutation Testing Tools

1. INTRODUCTION

Testing software is a tedious act, which requires human involvement to write tests. Tests may be incomplete, covering only a fraction of potential execution paths and thus, missing on potential bugs. In general, writing good tests is considered not to be a trivial task.

Test suites are written to ensure that a program behaves in a desired manner. The output of the program is described in a test case of a given test suite. The quality of such test suites are commonly evaluated by characteristics like branch or code coverage. Whether those metrics really correlate with the effectiveness of a program is still not clear. Namin and Andrews [38] and Inozemtseva and Holmes [19] suggested that there exists a correlation between branch coverage and fault detection, although the correlation varies between different programs and test suites. While achieving similar results, Inozemtseva and Holmes [19] argued that the variance in correlation and the fact that there is no strong correlation between different programs and their test suites does not indicate a correlation between high levels of coverage and the effectiveness of test suites. However, Kochhar et al. [28] suggested that real faults in large systems are indeed moderately or strongly correlated to code coverage. Chekam et al. [9] examined the effect of applying test suites to the faulty and fixed version of the program. The results show that mutation testing seems to be coupled to faults revelation, while branch- and statement coverage do not.

Mutation testing presents an approach to evaluate a test suite by mutating the source code based on a collection of fitting mutation operators. These operators inject small bugs into the code which should be detected by the test suite if it is covering sufficient test cases. Thus, if the original code passes the test but the mutated version fails instead, then the mutant is considered to be killed by the test suite. In the other case, both versions pass the test; the injected fault is not detected which may indicate an incomplete testing suite.

By generating a vast variety of different mutated versions, the relation between killed and survived mutants, also called the Mutation Score (MS), presents a metric to evaluate the coverage and quality of the test suite and investigating survived mutants can also help identifying bugs. The MS is defined as follows:

$$\mathrm{MS} = \frac{\#\text{Killed Mutants}}{\#\text{Mutants} - \#\text{Equivalent Mutants}} \cdot 100[\%]$$

Although in larger programs mutation testing can impose significant overhead. Each mutation has to be compiled and executed which is a time consuming and potentially error prone process. In order to make mutation testing applicable, Offutt and Untch proposed three main approaches, categorized into doing fewer, faster, and smarter mutations [44] (see Section 2.2).

Another issue of mutation testing is the creation of equivalent mutants, i.e. mutants that are semantically equivalent to the original version despite being syntactically different. This paper evaluates and compares the techniques that focuses on reducing the number of mutants.

To our knowledge, three systematic literature reviews about mutation testing were published which also address the Equivalent Mutant Problem (EMP). Published in 2011 [22] (updated by [48] in 2017), a first general overview was presented. The authors of [34] extended it with the current EMP research state focusing on the higher order mutation approach (see Section 2.4.2).

While the existing literature reviews like [22, 34, 48] already provide a detailed overview, they lack in practical implementation details. Therefore, reviewed listed papers and relevant testing tools are systematically evaluated, compared, and summarized.

We contribute an overview and comparison of the most promising approaches to reduce the number of equivalent mutants, selected based on research volume and improvement indicators. Current active implementations are evaluated and the state of mutation testing in practice is investigated. For this, promising equivalence reduction methods are compared, implementation issues are reviewed and the effectiveness in real world testing scenarios is evaluated.

The paper is organized as follows: Section 2 concerns with the underlying mutation testing hypotheses (Section 2.1) and the challenges it faces (Section 2.2). The EMP in particular is introduced in more depth in Section 2.3, motivating why a reduction of equivalent mutants is inevitable for practical usage. Section 2.4 elaborates on approaches that try to reduce the number of equivalent mutants. Section 3 presents related work. Then the effectiveness of these approaches is compared in Section 4. Section 5 investigates existing mutation testing tools (Section 5.1) and their usage of equivalence reduction methods (Section 5.2). Finally, Section 6 concludes why mutation testing still suffers in practical applicability.

2. BACKGROUND

This chapter introduces the underlying hypotheses which explains why mutation testing is considered to be effective in improving a testing suite. An overview of the challenges that mutation testing is facing leads to the Equivalent Mutant Problem which is explained in more detail. Approaches to reduce the number of equivalent mutants (primarily focusing on the **Do Smarter** category) are introduced.

2.1 Fundamental Hypotheses

Generated mutants are expected to simulate real faults in the software. The mutation operators which introduce little faults into the program are based on the *Competent Programmer Hypothesis.* This hypothesis states that programmers usually write programs that are already close to the theoretical ideal bug-free version [13]. Faults in the program are most likely small syntactical mistakes.

The coupling effect states that such small errors can be reliably detected by a test suite which provides sufficient test data. The test suite can detect these simple faults because they are said to be coupled to more complex errors. The coupling effect is an empirical principle and cannot be proven to hold. However it seems to hold in "real world" programs [13].

In the context of mutation testing the Competent Programmer Hypothesis and the Coupling Effect are questioned critically. Jia and Harman [21] criticized the idea of the competent programmer hypothesis. They exemplarily stated that "[...] arbitrarily replacing a plus with a minus symbol [...] is likely to create a large number of faults which no competent programmer would commit". Many generated mutants would thus not represent real faults in the program.

In case of the Coupling Effect, Jia and Harman [21] underlined that a good testing suite covering all simple mutants will also cover a large percentage of complex faults. It should be noted that emphasis is on a "large percentage of complex faults" and not on "all faults". Nevertheless, the Higher-Order Mutation approach is based on this hypothesis and is discussed in Section 2.4.2.

2.2 Mutation Testing Optimizations

Mutation testing was first introduced in the 1970s and proved to be a powerful tool for unit testing. Unfortunately it is still not widely adopted in the industry most likely due to its computationally expensive requirements and its manual involvement in detecting equivalent mutants. In order to reduce the overhead generated by mutation testing, three general categories of approaches are proposed [44].

Do Fewer Research done by Papadakis et al. [46] indicate that only about 5 % of all generated mutants are *subsuming.* A subsuming mutant contributes to the mutation score and all mutants that are subsumed by it are killed by the same tests, and thus, are redundant. Hence, only a small amount of mutants are relevant for the testing evaluation, whereas the rest is redundant, unfortunately falsifying the mutation score. The challenge lies in selecting a subset of mutants such that it represents the whole set as best as possible. Ma and Kim [32] proposed to cluster mutants that are expected to produce the same result. Only one representative of each cluster is then executed. If this mutant is killed, all mutants in the cluster are killed as well. Compiler optimization can also be used to determine the equality of mutants. If the compiler optimizes two mutants to the same output, then the mutants are equal. Of multiple equivalent mutants (do not confuse that *equivalent mutants* have a different meaning here) only one has to be executed to determine if these mutants are killed.

Do Faster Being an *embarrassingly parallel workload* [7], mutation testing can be parallelized or distributed in a larger cluster. There are multiple aspects that can be evaluated, e.g. compilation, execution, and generation of mutants [52]. All these tasks can be divided and distributed among multiple cores or nodes [1].

Do Smarter A more significant problem is equivalent mutants. Due to the halting problem, detecting that two programs compute the same function is inherently undecidable [5]. This leads to the question how to generate mutants while avoiding semantically equivalent ones despite being syntactically different. The problem is often referred to as the EMP which is introduced in more detail in Section 2.3.

2.3 The Equivalent Mutant Problem (EMP)

Mutants which are not killed by the test suite are called *alive.* A mutant which is alive indicates either a weakness in the test suite or is an equivalent mutant. An equivalent mutant is a mutant which is not distinguishable by its derived function from the original version, although being syntactically different. Such mutants do not represent a fault in

```
int index = 0;
while(...) {
  ...;
  index++;
-   if(index == 10) {
+   if(index >= 10) {
    break;
  }
}
```

Listing 1: A simple example for an equivalent mutant. In this case the condition is mutated from "==" to ">=".

the software. Due to the halting problem it is generally not decidable if the mutant computes the same function as the original program. An equivalent mutation can be applied in code that is never executed, can alter the state without being semantically different from the original program, or can just suppress speed improvements (e.g. undo optimizations back to an inefficient version) [17].

Listing 1 shows the original program and the mutated version in a content diff. The mutated version obviously computes the same function as the original version under the assumption that `index` is not changed anywhere else than indicated. This mutant will not be distinguishable from the original by any test suite and will be reported as alive. Human interaction is necessary to distinguish between a mutant which is not recognized by the test suite or an equivalent mutant.

Unfortunately this turned out to be a huge barrier for the practicality of mutation testing. For each mutant alive, Schuler et al. [55] reported that it took them in average 30 minutes to decide if the single mutation is related to a real fault or is an equivalent mutant instead. Other papers reported an average access time of about 15 minutes [56, 17], which coincides with the observations by Madeyski and Radyk [34]. It took them between 2 and 26 minutes, whereas the time decreased for second order mutants (mutants of a higher order are introduced in section 2.4.2).

However, these numbers should be taken with caution due to high variance and the fact that the manual inspection was not done by the developers of the programs under test themselves. A higher efficiency can be expected if the inspectors are familiar with the source code which is tested.

Nevertheless, the sheer amount of equivalent mutants that is usually generated further underlines the importance of solving the EMP. Using different mutation operators, testing tools, and programs under test, a clear trend is given:

Considering the possible amount of mutants even for small projects, processing all equivalent mutants becomes infeasible. Out of all mutants it is reported that 6.24% [41], 7.39% [56], and 9% [43] are equivalent. A more recent and non-approximating study found out that the percentage is even higher. By manually checking 1230 alive mutants for equivalence (out of 4181 mutants in total) it was observed that about 23% of all mutants were equivalent.

2.4 Equivalent Mutants Reduction Methods

In the following, some approaches that aim to reduce the number of equivalent mutants are introduced. They are carefully selected from a set of approaches based on research

```
private int getDepth(Node n){
-   int depth = 0;
+   int depth = 1;
  Object parent = o;
  while((parent = parent.getParent())
    != null) {
    ++depth;
  }
  return depth;
}
```

Listing 2: An equivalent mutation that alters state by changing the value of a variable. Comparing the depth of different nodes would return the same result before and after the mutation. Adapted from [17].

volume and success indicators, gathered from multiple literature reviews (see section 3).

2.4.1 *Weak Mutation Testing*

Weak Mutation Testing (WMT) is a more weak definition of traditional mutation testing, which, in this context, will be referred to as strong mutation testing. Strong mutation testing requires the following three conditions to kill a mutant:

1. The mutated statement has to be reached.
2. The mutated statement has to infect the state of the program.
3. The infection has to propagate to the programs' output.

On the other hand, WMT only requires the first two conditions to hold. Once the state is infected, the mutant will be killed [18].

Listing 2 shows an example of a mutation where the internal state of the program changes, but where it does not affect the computed function. The mutation of `int depth = 0;` to `int depth = 1;` shifts the depth of every node in the program by one. E.g. comparing the depth of two nodes would thus yield the same result as before the mutation, effectively resulting in an equivalent mutant.

WMT has the advantage that a mutant will be killed prematurely, although the output in the test case could have the same value. In Listing 2 the mutant would be killed after executing the modified instruction as it infects the programs state. If the output program computes the same function, even though the program was mutated, WMT effectively kills an equivalent mutant.

A huge advantage is that it is computationally significantly less expensive than strong mutation testing. A disadvantage of WMT is that it does not guarantee the exposure of all errors associated with the mutation transformation [18]. E.g. a function, in which the state would be infected but the returned result would not change for the provided test data. Such function would present a potential case where WMT would kill a mutant, which would otherwise be alive, and thus missing a possible error.

2.4.2 *Higher Order Mutants (HOMs)*

The main idea of Higher Order Mutants (HOMs) is based on the *coupling effect* hypothesis which states that complex faults are coupled to simple faults [13]. Therefore, in contrast to simple faults, complex ones cannot be fixed by a single change within the source code.

In the context of mutation testing this can be extended to the *Mutation Coupling Effect* as follows: "Complex mutants are coupled to simple mutants in such a way that a test data set that detects all simple mutants in a program will detect a large percentage [i.e. not all!] of the complex mutants [40]." Strong confirming evidence for this hypothesis is given by an empirical study done by Offutt [40] and the statement of Purushothaman and Perry [54], that the probability of introducing a new error by making a single line change is less than 4%, further arguments for HOMs instead of First Order Mutants (FOMs).

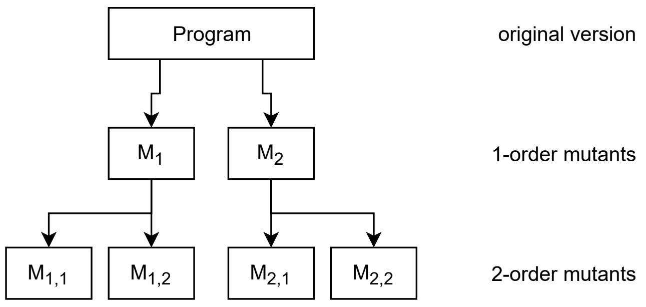


Figure 1: Exponential growth of mutants in higher order mutation. Adapted from [53].

Complex mutants, usually referred to as HOMs, are generated by combining multiple simple mutants or by mutating a mutant multiple times. As illustrated in Figure 1 there are exponentially more HOMs than FOMs, which is why only subsuming HOMs which replace more than one FOM are targeted.

Multiple algorithms to combine them in a meaningful and valuable way are developed for this purpose [53, 50]. Further approaches were proposed by e.g. Jia and Harman [21] where the selection of FOMs is based on a fitness function, effectively taking into account indications for equivalence.

The main goal of HOMs is the general cost reduction, as significant fewer test cases are required [50]. However, it also can be leveraged to reduce the amount of equivalent mutants. This is due to the fact that equivalent mutants *hide* behind killable ones as they are more likely to be combined with at least one non-equivalent mutant. Eventually, this makes HOMs harder to kill [53, 34].

Despite the possibility that two first-order non-equivalent mutants compensate into a second-order mutant which is equivalent, the probability for such a case is considered to be very low and can be neglected [53].

One of the great advantages of higher order mutations are the low implementation efforts, resulting in less mutations in total (and in equivalent ones) while not losing any effectiveness [21].

2.4.3 Compiler Optimization

While compiling a program the source code usually faces several transformation steps from higher level languages to machine code. During these steps the code is compiled into an optimized version. Redundant, unreachable, and non-impacting statements are removed indirectly. As mutants apply only very small syntactic changes, the corresponding source codes are very similar in their nature as well. Thus, all mutants which differ syntactically but have identical compiled code can be declared as equivalent. This is based on the idea that a compiler might optimize two similar programs into syntactically equal programs.

This technique together with 6 optimization rules were suggested by Baldwin and Sayward [4]. But instead of implementing specialized optimization rules, Papadakis et al. [47] suggested to leverage existing compiler tools and called their approach Trivial Compiler Optimization (TCE). In TCE mutations are applied on the source code level, then compiled and compared. According to them, this makes avoiding equivalent mutants through compiler optimization practical and scalable.

Also often referred to as compiler optimization are mutations performed on an intermediate representation like Java bytecode, LLVM bitcode, or assembly [33, 14, 15]. This is done because mutating the source code means to invoke the compiler for each single mutant. Thus, shifting the mutation generation process to an intermediate representation means that all mutations can be done in a single pass increasing performance drastically. On the other hand mutations in the intermediate layer can cause unwanted behavior. Compiling `C/C++` code into an intermediate representation can cause functions to be inlined. Mutating this code might be unwanted, e.g. it may be part of a library. Additionally, mutating the intermediate representation can mutate a program more fine grained. Typical statements in a higher level language translate to multiple statements in the intermediate representation. Mutating one of those statements can introduce faults that do not correspond to typical human errors.

This is why TCE generates mutants on the source code level. As existing compilers (or optimization tools) can be leveraged, TCE is very easy to implement. On the downside of that, mutants are generated on the source code, then compiled and compared. This introduces huge overhead, as compiling is considered to be computational expensive. Therefore, many tools first compile and then mutate, as this prevents recompiling multiple times.

2.4.4 Control-Flow Analysis

Control-flow analysis applies reasoning about the flow of the program to generate additional test cases that can kill more mutants.

One example is mutation testing in combination with Dynamic Symbolic Execution (DSE) which is also known as *Concolic Execution*. The core concept is to generate additional inputs to reach the mutated statements with the goal that the program state gets infected and ideally propagates the error to the output [49].

The input generation is done by executing the program while collecting the constrains imposed by the particular execution path as an Satisfiability Modulo Theory (SMT) formula. By negating the constraint of a branching point, another input for the execution of the program can be generated that covers a different flow through the program. If the collective constraint cannot be satisfied, the current execution path is infeasible and thus, no input can be generated. The selection of which constraint is to be negated next is de-

```
void test(){
  int x;
  assertEquals(some_function(x, true), x);
}
int some_function(int x, bool condition){
    if(condition){
        return x;
    } else {
-        return x/2;
+        return x*2;
    }
}
```

Listing 3: A mutant which is not discovered by the test suite, as only one branch where the condition holds true is tested. The other branch containing the mutation is not reached. Thus, the semantics are indeed different between the original and the mutated version, but the test suite handles it as either a fault or an equivalent mutant which is wrong. Therefore, DSE creates additional test inputs in such a way that the corresponding mutated branch is reached as well.

cided by a predefined heuristic. For instance, Papadakis and Malevris [49] use a heuristic that approximates the distance between the branch and the mutated statement.

Listing 3 shows a mutant where the test set does not kill the mutant. DSE would find example inputs that satisfy the constraints to execute `return x*2;`. After executing the mutated statement, the error would propagate to the output of the function. This would generate a new input for the test case, which would kill the generated mutant. By generating additional test inputs DSE is able to kill mutants which would be still alive with the original test suite. If DSE cannot generate an input to infect the state, there exists no feasible path to execute the mutated statement. In the other case, if DSE can reach the infected state but does not manage to propagate the error to the output of the program, then there exists no feasible execution path that propagates the error. In both cases the mutant has to be an equivalent mutant.

DSE uses SMT solving which imposes heavy computational constraints with a worst case exponential runtime. SMT solving has to be performed every time an execution path is chosen to check if it is feasible and to synthesize an input for that execution path.

The number of execution paths increases exponentially with every branch point in the code. This means that this approach might not be able to reach every reachable infected statement if a finite amount of time is given. Additionally DSE struggles with loops, especially with `while` loops, as they increase the search-able execution paths drastically, potentially creating infinitely many execution paths.

3. RELATED WORK

Apart from our work, there exist other papers which concentrate on mutation testing and try to provide an overview over the topic.

Jia and Harman (2011) [22] provide a first overview over different equivalent mutant detection techniques. But their focus is primarily on mutation testing in general and not on the EMP, mainly surveying research achievements up to 2009. They also show different implementations of tools available at the time, but do not discuss the selection of reduction methods in those tools.

Papadakis et al. [48] generally speaking provide an update to Jia and Harmans work and extend it. They include advances to the point of the release of their paper.

Madeyski et al. [34] build on top of Jia and Harman [22] results and classify the existing methods on EMP reduction. Unfortunately they miss to evaluate existing implementations and categorize them.

Pizzoleto et al. [52] build upon all the predecessor reviews. To our knowledge it is the newest systematic literature review and the most complete one. However, their research questions focus on cost reduction methods. They found out that dealing with equivalent mutants has not been addressed as much as the other problems and they recommend more research into it.

Compared to the listed papers, we present an evaluation of different techniques used for the EMP. We gather and compare results from different papers. Additionally, an overview of active existing mutation testing tools is presented and their use of different techniques is evaluated.

4. COMPARISON OF APPROACHES

In terms of the reduction of equivalent mutants it is important to note that research papers differ in their evaluation setup. Comparing these results should be viewed with caution, as they might be not as comparable as desired. Some relevant differences that impede the comparison of the results are as follows:

- Different mutation testing tools
- Different mutation operators
- Different test programs in various sizes
- Different combining algorithms in case of HOM strategies
- Different evaluation methods, e.g. manual check on equivalence or approximation methods
- Different metrics: ratio between equivalent mutants and all other mutants instead of the ratio between the equivalent ones in the set of all alive ones

However, considering that all evaluations done by other papers form consensus, a clear trend can be observed, indicating that the introduced approaches indeed can significantly reduce the number of equivalent mutants.

4.1 Weak Mutation Testing Effectiveness

WMT unlike strong mutation testing does not require the error to be propagated. Thus, the equivalent mutants that are not killed by WMT are a subset of the equivalent mutants of strong mutation testing.

In comparison to strong mutation testing, Kintis et al. [27] observed a reduction of about 73 %. Additionally, WMT is computationally less expensive than strong mutation testing, as the error does not have to be propagated. An infection of the program state is sufficient for WMT to kill the mutant.

4.2 Higher Order Mutation Testing Effectiveness

Papadakis and Malevris [50] concentrated their evaluation on second order strategies which allowed them to avoid 80-90% of equivalent mutants. Using a random selection approach, the probability of selecting two equivalent FOMs is reduced to about 5% ($\approx (22.5\%)^2$) which coincides with the results from Polo et al. [53], achieving a reduction from 18.66% to about 5% equivalents.

The authors of [50] and [34] also observed promising numbers. Evaluating the same combination algorithms, even though implemented individually and tested on different programs, yielded in reductions varying from 72.73% to 81.82% (85.65%-87.77% respectively).

Earlier benchmarks such as one by Offutt [40] yielded even better results. The evaluation resulted in a reduction from about 10% equivalent mutants down to 1%. However, the programs under test were significantly smaller and the reduction improvement seems to decrease with the increase of the program size under test [34].

In contrast to the good reduction results above, the combination algorithm called *NeighPair* which matches FOMs that are as close to each other as possible yields an incrementation instead of a reduction for most of the tested programs [34]. Reasons for this exception are not known, but it proves that the choice of the combination algorithm does matter.

Different approaches for the optimization problem of the selection process such as the subsuming search-based method significantly reduces the amount of HOMs, but unfortunately no benchmark was done regarding the reduction of equivalent mutants [21].

Most benchmarks focus on second-order mutants. While third-order or even higher HOM indicate further improvements, the amount of mutants would grow exponentially with the mutation order, which increases the difficulty to combine the FOMs in a valuable way [40].

4.3 Compiler Optimization Effectiveness

Offutt and Craft [41] studied the 6 optimization rules suggested by Baldwin and Sayward [4] and found out that 45% of equivalent mutants can be detected. However, at the time of writing those papers, computational power did not allow them to test larger projects. The programs under test are small, ranging from 5 to 52 executable statements, which is why the results might be impacted by noise [41].

Evaluating the more recently proposed TCE method yielded 30% less equivalent mutants. This is a reduction of about 7% from all mutants, whereas this technique also allowed to remove another 21% of mutants, as they were redundant (i.e. duplicates). While Papadakis et al. [47] focused on the C programming language, Kintis et al. [26] further studied the TCE technique extending it with Java optimizations. In case of Java the empirical results are promising as well: on average, a reduction of about 50% of equivalent mutants was achieved. This corresponds to 6% of all mutants. Similar improvements were observed on programs written in C [26]. Interestingly, in Java, 99% (57% in C) of removed equivalent mutants were detected due to a failed propagation of the mutated statement to the output [26].

Important to note is that for both languages the reduction results faced large variations between the projects under test, indicating that program characteristics heavily impact the performance of TCE [26].

4.4 Control-Flow Analysis Effectiveness

DSE can manage to kill all non-equivalent mutants. Papadakis and Malevris [49] state that most killable mutants are killed after a few iterations. More persistent mutants can be killed with a drastic increase in iterations. In the programs tested by Papadakis and Malevris [49] the number of killed mutants is always well above 85 %. Zhang et al. [58] finds similar results of over 80 % of killed mutants with their tool PexMutator.

Mutated statements that cannot be reached are assumed to be on unfeasible paths. All such mutants are therefore labeled equivalent. Mutants which fail to propagate the fault to the output are also labeled as equivalent. Offutt and Pan [43, 39] show that the detection rate for equivalent mutants is at about 45 %. In 7 out of 11 cases the detection rate was well over 60 %.

Unfortunately the acquired results are over 20 years old. SMT solving has seen some progress over the last decades. Symbolic execution has only become practical due to the recently achieved progresses [6]. Imprecision in the modeling of data types may caused Offutt and Pan [43, 39] to miss paths which could have identified more equivalent mutants.

4.5 Comparison

Comparing the reduction of equivalent mutants, Higher Order Mutation testing seems to outperform the other approaches. It manages to avoid 80 to 90 % of all equivalent mutants. But the HOM approach is a reduction method. It does not identify equivalent mutants, but rather makes equivalent mutants less likely.

The WMT method, like HOMs, also avoids equivalent mutants by killing mutants after the infection of the state. It shows a reduction of equivalent mutants of about 73 %. On the basis of the available data WMT performs worse than HOM.

Compiler optimization actively detects equivalent mutants by checking the optimized code for equivalence. Thus it is fundamentally different from WMT and HOM. TCE is heavily language and project dependent. It is able to detect around 30-50 % of equivalent mutants.

DSE also detects equivalent mutants by searching for an execution path, such that the error introduced by the mutation can be propagated. In average 45 % of all equivalent mutant could be detected by DSE.

Comparing the different approaches directly seems rather speculative. Each method has to be applied on a shared set of test suites in order to compare the presented methods. Unfortunately every approach uses a different set of programs for its evaluation.

The used mutation operators varied, as some papers use only a subset of available mutation operators [21, 55, 45]. Thus, the total number of generated mutants, killable mutants, and equivalent mutants might depend on the used mutation operators.

Moreover, the used software under test varied in size and maturity. Some papers only tested very small programs [42]. Others used matured programs like OpenSSL to evaluate their implementation [14]. Not only size but also designation of the software differs. E.g. applying mutations in mathematical operators will heavily impact a math library. Other types of software may not be impacted in the same scope. Thus the ratio of equivalent mutants vary in different types of software [22].

In conclusion, based on the data available, methods that try to avoid equivalent mutants seem to be more effective than methods that actively try to detect them.

5. CURRENT IMPLEMENTATION STATE

This chapter links the theoretical approaches to the current state of practical usage of mutation testing. Therefore, current available testing tools beyond prototype implementations are evaluated and compared.

5.1 Actively Developed Mutation Testing Tools

At the time of writing this paper, various mutation testing tools were published for all common programming languages. In order to gather a better overview this section is limited to Java, C/C++, and Python tools as most development was contributed within the scope of these languages.

An overview is provided in Table 1, which shows current actively developed projects. We consider active projects to be tools that were either updated recently or are well matured and are used in a significant amount of research papers. The development period is considered to be the commit period in the corresponding git repository. If the project was not open source, we used the available papers to narrow down the development period.

5.2 Implementations of Reduction Methods

As Table 1 indicates, most tools do only support one or at most two of the introduced approaches. Interestingly, currently active projects do not seem to focus on the EMP problem but on the overall execution speed instead. In the following our observations grouped by the reduction method are listed.

WMT The weak mutation approach is found in two testing tools. Although it is implemented in Mart, it is only used to measure the mutants infection [8]. The Major mutation framework is the only one that supports full WMT analysis [23].

HOM We suggest that due to the low implementation effort Higher Order Mutants are used more frequently in testing tools. The most prominent tool in research papers is MiLu (C language), which is explicitly designed for HOM strategies [20]. MiLu is used in [21, 26, 47], but unfortunately development seems to have stopped and not much documentation is available. More recent and actively developed tools are LittleDarwin (Java [51], unfortunately not providing much information about its support for HOM), PIT-HOM (Java; based on PIT, extended by [30]), and mutpy (Python [3]). However, not much information about their adoption is available.

Choosing the right combination of FOMs can be considered to be a separate problem. This is why the authors of [53], for instance, wrapped the tool muJava with their combination algorithms in such a way that muJava generates FOMs which are then combined in an intermediate step. Thus, extending existing tools with HOM functionality can be considered to be a rather simple task if the process of generating mutants is separated from the actual execution phase.

TCE Although not noted explicitly, TCE seems to be used directly or indirectly more often than other methods. In every case an already existing compiler is used, which significantly lowers the implementation effort of this method. However, while TCE proposes mutant generation on code level, some tools such as muJava [33], Javalanche [56], PIT [10], Mart [8], MuVM [57], and Mull [14] work directly on the bytecode/LLVM to prevent re-compiling for performance reasons. Unfortunately, this introduces further issues, as mutants are generated on a "*desugared*" and more complex version of the sourcecode. Thus, mutants might be applied on statements which are not included in the original version and which cannot be mapped back for manual inspection [23]. Unfortunately, it is not clear if bytecode level mutant generation is as effective as TCE, unfortunately no benchmark was done. Nevertheless, compiling the source code to a minimized and optimized version will remove all unnecessary statements such as debug statements that have no impact on the output.

To our knowledge TCE, as proposed by Kintis et al. and Papadakis et al. [26, 47], is only implemented in MiLu (v3.2) and was successfully incorporated in Proteum, Mart, and muJava.

DSE Dynamic Symbolic Execution is not implemented in any tool we found. DSE has usages outside of mutation testing and is mainly used for test generation and automatic testing of software. This could be the reason this approach is rarely considered for mutation testing.

Most theoretical papers that incorporated DSE rarely mention any tool they used. These tools are most likely closed source and were only prototypes.

6. CONCLUSION

The results indicate that Weak Mutation Testing and Higher Order Mutants which avoid equivalent mutants perform better than Compiler Optimization and Control-Flow analysis which try to detect equivalent mutants.

Higher Order Mutants avoid equivalent mutants significantly better than Weak Mutation Testing. HOM is an probabilistic approach which lowers the frequency at which equivalent mutants occur, whereas WMT prematurely kills mutants which manage to infect the state. WMT does not change the amount of generated equivalent mutants.

Comparing the equivalent mutant detection methods, ControlFlow analysis is stated to be better than Compiler Optimization. But the computational demand for tools like DSE is greater than for TCE, which favors the practical use of TCE. The stated results are reflected in the usage of equivalent mutant reduction methods used by mutation tools.

We observed that the main focus of current tools for mutation testing is in usability. Relatively little work is done to reduce the number of equivalent mutants. Table 1 shows that mainly HOM is used to make the generation of equivalent mutants less likely. WMT also sees some use-cases, although it is significantly less used in the evaluated tools. Methods to detect equivalent mutants are observed to be only accomplished with TCE. We assume TCE is often im-

Table 1: Overview of available active testing tools, sorted by latest development news. Reduction methods refer explicitly to the reduction of equivalent mutants.

Name	Lang.	Development	References	Reduction Methods
PIT-HOM	Java	2019 – present	[30] extension of PIT	HOM
LittleDarwin	Java	2017 – present	[51]	HOM
Mull	C++	2016 – present	[14]	
Mart	C++	2016 – present	[8]	WMT; mutant generation on optimized intermediate representation
Cosmic Ray	Python	2015 – present	[1]	
mutpy	Python	2014 – present	[3]	HOM
mutmut	Python	2010 – present	[2]	
PIT	Java	2010 – present	[10]	
Major	Java	2014 – 2018	[23]	WMT
GiGAn	C++	2017	[12, 11] extension of MuCpp	
MiLu	C	2008 – 2016	[20, 47, 26, 21]	HOM; TCE
muJava	Java	2003 – 2016	[33, 31, 25, 53, 24, 26, 27]	mutant generation on optimized intermediate representation
MuVM	C	2016	[57]	HOM
Proteum	C	1996 – 2015	[37]	HOM; mutant generation on optimized intermediate representation
Judy	Java	2008 – 2014	[35, 34]	HOM
HOMAJ	Java/AspectJ	2014	[45] extension of muJava	HOM
CCMUTATOR	C++	2013	[29]	HOM
Javalanche	Java	2009 – 2012	[16, 56, 55]	
Paraμ	Java	2011	[36]	HOM

plemented due to the usage of an already existing compiler, which lowers the implementation effort.

7. FUTURE WORK

Despite being in research since the 1970s, mutation testing is far from matured. Many theoretical proposals have never been implemented. Even though computational power has increased significantly over the past decades, mutation testing is still not adapted in the industry.

Another metric could be used by ignoring equivalent mutants, which results in the *mutation score indicator* (MSI) [34]:

$$\text{MSI} = \frac{\#\text{Killed Mutants}}{\#\text{Mutants}} \cdot 100[\%]$$

The MSI is still a valuable metric providing a lower bound of the actual MS. The MSI could be integrated into a fully automated CI environment, but we did not find any evident in doing so. The meaningfulness of the MSI needs to be investigated.

Another question for future work would be how good these approaches can be combined. For instance, HOM and TCE could collaborate with each other, whereas the combination of HOM and WMT is not yet tested.

Besides the approaches introduced in this paper there are many more to investigate, e.g. [34]:

- Leveraging mathematical constrains
- Program Slicing
- Observing differences in the running profile (CPU/memory usage, running time, ...)
- Margrave's change-impact analysis
- Lesar model-checker
- Co-evolutionary search techniques
- Semantic exception hierarchy
- Using the impact of dynamic invariants
- Examining changes in coverage

However, most approaches can only be applied to a smaller subset of programs or are only theoretical proposals. Due to the research volume of certain approaches this paper did not elaborate on these methods in more detail.

8. REFERENCES

[1] Cosmic Ray: Mutation testing for Python — Cosmic Ray documentation. https://cosmic-ray.readthedocs.io/en/latest/.

[2] Mutmut - python mutation tester — mutmut documentation. https://mutmut.readthedocs.io/en/latest/.

[3] Mutpy/mutpy. mutpy, Dec. 2019.

[4] D. Baldwin and F. Sayward. Heuristics for Determining Equivalence of Program Mutations.:. Technical report, Defense Technical Information Center, Fort Belvoir, VA, Apr. 1979.

[5] T. A. Budd and D. Angluin. Two notions of correctness and their relation to testing. *Acta Informatica*, 18(1):31–45, Mar. 1982.

[6] C. Cadar and K. Sen. Symbolic execution for software testing: Three decades later. *Communications of the ACM*, 56(2):82, Feb. 2013.

[7] P. C. Cañizares, M. G. Merayo, and A. Núñez. EMINENT: Embarrassingly Parallel Mutation Testing. *Procedia Computer Science*, 80:63–73, 2016.

[8] T. T. Chekam, M. Papadakis, and Y. Le Traon. Mart: A mutant generation tool for LLVM. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2019*, pages 1080–1084, Tallinn, Estonia, 2019. ACM Press.

[9] T. T. Chekam, M. Papadakis, Y. Le Traon, and M. Harman. An Empirical Study on Mutation, Statement and Branch Coverage Fault Revelation That Avoids the Unreliable Clean Program Assumption. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 597–608, Buenos Aires, May 2017. IEEE.

[10] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque. PIT: A Practical Mutation Testing Tool for Java (Demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 449–452, New York, NY, USA, 2016. ACM.

[11] P. Delgado-Perez, I. Medina-Bulo, and M. Nunez. Using Evolutionary Mutation Testing to improve the quality of test suites. In *2017 IEEE Congress on Evolutionary Computation (CEC)*, pages 596–603, Donostia, San Sebastián, Spain, June 2017. IEEE.

[12] P. Delgado-Pérez, I. Medina-Bulo, S. Segura, A. García-Domínguez, and J. José. GiGAn: Evolutionary mutation testing for C++ object-oriented systems. In *Proceedings of the Symposium on Applied Computing - SAC '17*, pages 1387–1392, Marrakech, Morocco, 2017. ACM Press.

[13] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, Apr. 1978.

[14] A. Denisov and S. Pankevich. Mull It Over: Mutation Testing Based on LLVM. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 25–31, Vasteras, Apr. 2018. IEEE.

[15] A. Derezinska and K. Kowalski. Object-Oriented Mutation Applied in Common Intermediate Language Programs Originated from C#. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 342–350, Berlin, Germany, Mar. 2011. IEEE.

[16] B. Grun, D. Schuler, and A. Zeller. The impact of equivalent mutants. IEEE International Conference on Software Testing, Verification, and Validation Workshops, ICSTW 2009, pages 192 – 199, May 2009.

[17] B. J. M. Grün, D. Schuler, and A. Zeller. The Impact of Equivalent Mutants. In *2009 International Conference on Software Testing, Verification, and Validation Workshops*, pages 192–199, Apr. 2009.

[18] W. Howden. Weak Mutation Testing and Completeness of Test Sets. *IEEE Transactions on Software Engineering*, SE-8(4):371–379, July 1982.

[19] L. Inozemtseva and R. Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*, pages 435–445, Hyderabad, India, 2014. ACM Press.

[20] Y. Jia and M. Harman. MILU: A customizable, runtime-optimized higher order mutation testing tool for the full C language. In *Testing: Academic Industrial Conference - Practice and Research Techniques (Taic Part 2008)*, pages 94–98, Aug. 2008.

[21] Y. Jia and M. Harman. Higher order mutation testing. *Information and Software Technology*, 51(10):1379 – 1393, 2009.

[22] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, Sept. 2011.

[23] R. Just. The major mutation framework: Efficient and scalable mutation analysis for java. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 433–436, New York, NY, USA, 2014. ACM.

[24] M. Kintis and N. Malevris. Identifying More Equivalent Mutants via Code Similarity. In *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*, volume 1, pages 180–188, Dec. 2013.

[25] M. Kintis and N. Malevris. MEDIC: A static analysis framework for equivalent mutant identification. *Information and Software Technology*, 68:1 – 17, 2015.

[26] M. Kintis, M. Papadakis, Y. Jia, N. Malevris, Y. Le Traon, and M. Harman. Detecting trivial mutant equivalences via compiler optimisations. *IEEE Transactions on Software Engineering*, PP:1–1, Mar. 2017.

[27] M. Kintis, M. Papadakis, and N. Malevris. Evaluating Mutation Testing Alternatives: A Collateral Experiment. In *2010 Asia Pacific Software Engineering Conference*, pages 300–309, Nov. 2010.

[28] P. S. Kochhar, F. Thung, and D. Lo. Code coverage and test suite effectiveness: Empirical study with real bugs in large systems. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 560–564, Montreal, QC, Canada, Mar. 2015. IEEE.

[29] M. Kusano and Chao Wang. CCmutator: A mutation generator for concurrency constructs in multithreaded C/C++ applications. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 722–725, Silicon Valley, CA, USA, Nov. 2013. IEEE.

[30] T. Laurent and A. Ventresque. PIT-HOM: An extension of pitest for higher order mutation analysis. *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2019.

[31] O. Ma and Kwon. *muJava Home Page.* 2016.

[32] Y.-S. Ma and S.-W. Kim. Mutation testing cost reduction by clustering overlapped mutants. *Journal of Systems and Software*, 115:18–30, May 2016.

[33] Y.-S. Ma, J. Offutt, and Y. R. Kwon. MuJava: An Automated Class Mutation System: Research Articles. *Softw. Test. Verif. Reliab.*, 15(2):97–133, June 2005.

[34] L. Madeyski, W. Orzeszyna, R. Torkar, and M. Józala. Overcoming the Equivalent Mutant Problem: A Systematic Literature Review and a Comparative

Experiment of Second Order Mutation. *IEEE Transactions on Software Engineering*, 40(1):23–42, Jan. 2014.

[35] L. Madeyski and N. Radyk. Judy - a mutation testing tool for java. *IET Software*, 4(1):32–42, Feb. 2010.

[36] P. Madiraju and A. S. Namin. Paraμ – A partial and higher-order mutation tool with concurrency operators. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 351–356, Mar. 2011.

[37] J. Maldonado, M. Delamaro, S. Fabbri, A. Simão, T. Sugeta, A. Vincenzi, and P. Masiero. Proteum: A family of tools to support specification and program testing based on mutation. pages 113–116, May 2001.

[38] A. S. Namin and J. H. Andrews. The influence of size and coverage on test suite effectiveness. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis - ISSTA '09*, page 57, Chicago, IL, USA, 2009. ACM Press.

[39] A. Offutt and Jie Pan. Detecting equivalent mutants and the feasible path problem. In *Proceedings of 11th Annual Conference on Computer Assurance. COMPASS '96*, pages 224–236, Gaithersburg, MD, USA, 1996. IEEE.

[40] A. J. Offutt. Investigations of the software testing coupling effect. *ACM Trans. Softw. Eng. Methodol.*, 1(1):5–20, Jan. 1992.

[41] A. J. Offutt and W. M. Craft. Using Compiler Optimization Techniques to Detect Equivalent Mutants. *The Journal of Software Testing, Verification, and Reliability*, 4:131–154, 1994.

[42] A. J. Offutt and S. D. Lee. An empirical evaluation of weak mutation. *IEEE Trans. Softw. Eng.*, 20(5):337–344, May 1994.

[43] A. J. Offutt and J. Pan. Automatically Detecting Equivalent Mutants and Infeasible Paths. *Softw. Test., Verif. Reliab.*, 7:165–192, 1997.

[44] J. Offutt and R. Untch. Mutation 2000: Uniting the orthogonal. Mutation Testing for the New Century, pages 34–44, May 2001.

[45] E. Omar, S. Ghosh, and D. Whitley. HOMAJ: A tool for higher order mutation testing in AspectJ and java. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*, pages 165–170, Mar. 2014.

[46] M. Papadakis, C. Henard, M. Harman, Y. Jia, and Y. Le Traon. Threats to the Validity of Mutation-based Test Assessment. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 354–365, New York, NY, USA, 2016. ACM.

[47] M. Papadakis, Y. Jia, M. Harman, and Y. Le Traon. Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 936–946, May 2015.

[48] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. L. Traon, and M. Harman. Chapter Six - Mutation Testing Advances: An Analysis and Survey. volume 112 of *Advances in Computers*, pages 275 – 378. Elsevier, 2019.

[49] M. Papadakis and N. Malevris. Automatic Mutation Test Case Generation via Dynamic Symbolic Execution. In *2010 IEEE 21st International Symposium on Software Reliability Engineering*, pages 121–130, Nov. 2010.

[50] M. Papadakis and N. Malevris. An Empirical Evaluation of the First and Second Order Mutation Testing Strategies. In *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, pages 90–99, Apr. 2010.

[51] A. Parsai, A. Murgia, and S. Demeyer. LittleDarwin: A feature-rich and extensible mutation testing framework for large and complex java systems. *CoRR*, abs/1707.01123, 2017.

[52] A. V. Pizzoleto, F. C. Ferrari, J. Offutt, L. Fernandes, and M. Ribeiro. A systematic literature review of techniques and metrics to reduce the cost of mutation testing. *Journal of Systems and Software*, 157:110388, 2019.

[53] M. Polo, M. Piattini, and I. García-Rodríguez. Decreasing the cost of mutation testing with second-order mutants. *Software Testing, Verification and Reliability*, 19(2):111–131, 2009.

[54] R. Purushothaman and D. E. Perry. Toward understanding the rhetoric of small source code changes. *IEEE Transactions on Software Engineering*, 31(6):511–526, June 2005.

[55] D. Schuler, V. Dallmeier, and A. Zeller. Efficient mutation testing by checking invariant violations. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 69–80, New York, NY, USA, 2009. ACM.

[56] D. Schuler and A. Zeller. Covering and uncovering equivalent mutants. *Software: Testing, Verification and Reliability*, 23(5):353–374, Nov. 2013.

[57] S. Tokumoto, H. Yoshida, K. Sakamoto, and S. Honiden. MuVM: Higher Order Mutation Analysis Virtual Machine for C. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 320–329, Chicago, IL, USA, Apr. 2016. IEEE.

[58] L. Zhang, T. Xie, L. Zhang, N. Tillmann, J. de Halleux, and H. Mei. Test generation via Dynamic Symbolic Execution for mutation testing. In *2010 IEEE International Conference on Software Maintenance*, pages 1–10, Timi oara, Romania, Sept. 2010. IEEE.

On Factors Contributing to the Qualitative Measurement of Test Suite Effectiveness

Marian Assenmacher
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
marian.assenmacher@rwth-aachen.de

Ajay Pandi
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
ajay.pandi@rwth-aachen.de

ABSTRACT

In software development, it is of utmost importance to ensure the quality of software since flawed software can potentially lead to severe problems like security risks, loss of sensitive data or loosing business. While the quality of software can be ensured through different measures, software testing plays a crucial role: software testing helps to corroborate that applications are running as intended, helps to find misbehaving code and reduce the risk of regressions in subsequent versions. However, the results of such testing depends on the effectiveness of test cases and test suites. Furthermore, in order to measure its effectiveness, one should have a complete understanding of factors influencing the effectiveness of test cases and test suites like coverage criteria, suite size and other similar notions. Also, it is vital to understand the relation between test cases, test suite and test effectiveness.

In this paper, we study the literature regarding test case and test suite improvement to identify factors that contribute to the effectiveness of test cases, test suites and the overall testing. Finally, taking all the findings into consideration we briefly describe techniques for both, building effective test suites and also to measure and improve test suite effectiveness of existing frameworks, which then reflects in improving overall quality of the software.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*software tests, test effectiveness, test suites, test evaluation metrics*

Keywords

Software Tests, Test Case, Test Suite, Test Analysis, Test Effectiveness

1. INTRODUCTION

A software product can never be considered as fully developed, as requirements change frequently. To ensure the quality of existing and added functionalities across multiple versions of the software, automated testing is needed. Also excessive testing is not feasible due to limited time and resources, hence testing has to be effective. An effective testing refers to maximum paths being tested for the SUT. Test suite that drives such testing is then considered an effective test suite. In order to create an effective test suite one has to know what contributes to the effectiveness, how it can be measured and how that knowledge can be used to improve the test suite quality.

Some of the prominent test levels are unit testing, integration testing, system testing and acceptance testing. Each of the aforementioned test levels has its own significance, and attributes directly to the quality of the software product. Testing is performed with test suites which are dependent on the testing technique. A test suite is nothing but a collection of test cases pertaining to the system/functionality under test. Oftentimes, it is assumed that the quality of test cases correlates directly with quality of the overall testing and that the quality of testing correlates with the quality of the finished product. So it is of paramount importance to have an effective test suite for a quality software product.

Extensive research is conducted in studying and improving the effectiveness of testing. Researchers have performed experiments in order to understand the significance as well as relationship between test suite effectiveness and various plausible contributing factors like code coverage, test suite size and other similar factors [30, 2, 7, 16]. There is a large body of empirical studies deriving metrics or, threshold for existing metrics through research experiments to determine the effectiveness of the test cases and test suites [6, 24]. In parallel, due to the introduction of unprecedented software development processes like Behaviour-Driven Development (BDD) and Scaled Agile Framework (SAFe), there is also extensive research performed in order to understand their testing strategies and proposing methods to measure and improve test suite effectiveness in such scenarios [26, 11].

There are many different techniques to test software for potential bugs and deviations from business requirements. Each technique might perform differently depending on the adequacy criteria being used, type of software under test and other similar notions. Furthermore, it is unclear which metric to choose as adequacy criterion to denote the progress and success of testing. This paper captures a comprehensive overview of research done in the field of software testing. Specifically in understanding test cases, test suites, their effectiveness and plausible ways to build such effective test suites. We initially present the techniques and methods for assessing and improving test case effectiveness leading to assessment of test suite effectiveness and finally the motivation and technical details about the ways to build effective test suite.

Our initial observations can be broadly classified into the following categories according to which we will also structure our paper:

1. Measuring effectiveness of test cases — Strategies to measure and improve the effectiveness of test cases. Prominently, a metric would be proposed to measure the effectiveness.

2. Measuring effectiveness of test suites — Likewise, strategies or metric to measure and improve test suite effectiveness. Also, there is extensive research being conducted in studying relationship between various properties: test suite size, coverage criterion and other similar properties.

3. Techniques to improve test effectiveness — In the end, test frameworks which improved test effectiveness by taking into consideration the findings from the previous two subsections (3.1 and 3.2) will be presented. Furthermore, recent test frameworks due to wide acceptance of particular algorithm or data structure will also be mentioned.

We provide an overview of such research outcomes pertaining to the research question of improving test suite effectiveness. We start with a brief introduction to software testing, explaining the base concepts of test effectiveness and measures to quantify effectiveness in section 2. In continuation, then we will give a summary of the findings in main chapter in section 3. In section 4, we then discuss the results and giving a brief conclusion of our work. Lastly, we focus on a look into possible future work in section 5.

2. BACKGROUND

It is important to have sturdy understanding of testing, and its significance followed by the relationship with test cases and test suites. Hence, first we will recapitulate software testing.

As per the book "Guide to the Software Engineering Body of Knowledge" by IEEE computer society, the definition of software testing is as follows:

Software testing consists of the dynamic verification that a program provides expected behaviors on a finite set of test cases, suitably selected from the usually infinite execution domain. [9, p. 4-1]

Or as Myers puts it more simply in his book:

Testing is the process of executing a program with the intent of finding errors. [20, p. 11]

Software testing plays a pivotal role in the software development life cycle (SDLC)[1] by being a major factor influencing the quality of the product. The primary goal of software testing is to validate a given program or piece of software to find potential errors, improve system performance and ensure compliance with the expected functionality while being robust, reliable, secure and stable. The earlier the discrepancies or defects are identified, the lower is the cost, and overhead to take corrective measures is reduced. In order to identify potential discrepancies such as program errors, deviation from expected functionality, adherence to service level agreements (SLAs) or misbehavior of parts of the software, the test suite should contain effective test cases that evaluate source code to the maximum potential.

Test Cases

At the end of test design phase, test case specifications are decided by the project team to assess the conformance of functional requirements. However, such test cases do not ascertain to uncover all the defects since there are also factors like misinterpretation of requirement by the developer and poor code quality leading to potential defects. Unit test cases are those test cases that check for compliance of actual behavior of the smallest unit of the system under test (SUT) with the expected behavior. Additionally, these test cases are often written taking into consideration the business needs, expected functionality and key performance indicators (KPI) of the SUT. Quality of any testing is largely dependent on the effectiveness of the test cases.

Test Suites

A test suite is nothing but a collection of test cases. An effective test suite should be able to evaluate the source code to the maximum potential for compliance with business requirement. If the test suite size is not accounted for, then there is liberty in accommodating more test cases pertaining to every possible data and control flow of the SUT. However, the size of test suites should also be taken into consideration since the larger the test suite is, the longer the tests run. And also, an extensive test suite gets harder to maintain, hurting the longevity and effectiveness when new changes are introduced. Therefore, there should be a balance between the size of test suite and its ability to find potential defects in the SUT. Additionally, the process of developing test cases can help finding problems in the design of the product as well.

Test Coverage

Test coverage describes the amount of code which is covered by the test cases. Generally, there are multiple coverage metrics measuring different aspects of the SUT. Most often used are *statement cov.* measuring the amount of actual statements that are tested, *function cov.* indicating the amount of functions under test and *branch cov.* checking the number of branches included in test cases.

Mutation Testing

When using mutation testing, the original source code is *mutated* slightly according to predefined patters. Tests then check these mutations of the original code. If a mutation is detected by a test, it is *killed*, otherwise the information of the mutation can be used to generate new tests.

3. OVERVIEW OF FINDINGS

The research done by the team of Vahabzadeh et. al. [29] emphasizes the importance of software testing and its effectiveness. They investigated test code for existence of bugs and other flaws. For that, 211 projects in total of the Apache

[1]SDLC generally describes a number of phases which model the development process such as the waterfall modell or agile development.

Software Foundation and 5.5k test related bugs were extensively examined. It was found that around 50 % of the projects had buggy test code, whereas most of those were false alarms, more commonly known as false positives, i. e. detecting bugs where there were none, some others were silent horrors, i. e. not detecting actual bugs in production code, more often referred to as false negatives. As root causes, mostly missing or incorrect assertions were identified regarding the silent horrors. Whereas semantic, flaky or environment related bugs mostly caused false positives.

From our point of view, while software testing is essential to have, it does not intrinsically find all misbehavior. Therefore, it is important to use the appropriate testing methodology and technique to increase the effectiveness[2] of the test suites.

During our research, we found that extensive research is conducted related to evaluating and improving effectiveness of test cases, test suites and the entire testing technique. To give an overview we will structure the findings of our literature review into three categories: firstly, we will start at the detailed level and present the findings about improvement of test case quality. We then will broaden our view by examining research on improvement of test suite quality. Conventional testing techniques did not improve test effectiveness in the case of specific unprecedented algorithms, data structures and other development frameworks. Hence, unconventional testing techniques were applied in such scenarios. This led to another branch of research in understanding such framework's testing procedure and focus on improving test suite effectiveness. So finally, findings about such unconventional testing techniques together with approaches to build effective test suites are presented.

3.1 Measuring Effectiveness of Test Cases

In current research about software test case effectiveness, most of the papers conduct analyses about assessing code, meaning about the testing code itself. This is done in the interest of improving quality of test cases for which predominantly multiple coverage criteria like statement, branch and block coverage are considered as adequacy measures. It was later discovered that, these criteria themselves would only have very low if at all negligible significance, when applied to a single test case. In general, these coverage criteria apply to a set of test cases or test suites but nevertheless can also be considered for individual test cases. The measures were useful in determining test case significance leading to refactoring and prioritization in the case of regression testing.

For instance, Park et. al. [24] and Kapfhammer and Soffa [15] proposed approaches for test case prioritization by determining the effectiveness of test cases. Park et. al. suggested to use historical value-based test case prioritization that used fault-based metric *average percentage of faults detected* (APFD). Where in faults are seeded automatically or manually into the SUT and test cases are prioritized depending on its bug detecting ability. While this approach evidently improved the effectiveness of the resulting set of test cases, Kapfhammer and Soffa argued to prefer coverage-based metrics over fault-based metrics like APFD. The motivation was the fact that high coverage test cases are more likely to reveal increased program faults [7]. Hence they used coverage measures as a metric to determine the effectiveness of test cases and prioritize them. So that, in the next testing run for the modified set of test cases, potential bugs are identified earlier. Previously existing methods of this type did not take into consideration the test case execution times before prioritizing. Taking coverage and execution times into account, the authors introduce a metric called CE_T whose value is ranged between 0 and 1. A test case is awarded a high CE value when it quickly covers the test requirement. This approach is open to a broad, wider spectrum of all the coverage criteria.

The above mentioned approach for test case prioritization also aims to reduce the test suite size after prioritizing. Since increased test suite size will lead to increased execution time for the overall testing procedure and effort for maintaining the suite. Additionally, increased run time is considered to be an indicator for sub-optimal test cases. Recent studies suggest that test suite size highly correlates with test effectiveness [21], [16]. However, Yucheng Zhang and Ali Mesbah [31] conducted experiments and suggested that assertion quantities are far more correlating than test suite size. An assertion is a Boolean expression written in the program to validate the results of specific variables, to determine successful execution of certain code blocks and for many such scenarios. Assertion quantity refers to the number of assertions written in the program. Furthermore, assertion coverage is a measure that determines the percentage of assertions covered during testing. The experiments of Zhang and Mesbah identified that the number of assertions and assertion coverage have high correlation with the effectiveness of test cases. These correlations are even grater than those of test suite size and statement coverage. The findings were based on 24,000 assertions collected across 5 cross-domain real world Java programs.

Yuri Chernak [6], studied the significance of testing. In the interest of reducing or removing software defects prior to product release, he proposed an approach to validate and improve quality of existing test cases. To measure the effectiveness, the author proposed the so called *TCE* (test case effectiveness) metric to measure the effectiveness of overall test cases. It is defined as the ratio of defects found by test cases (N_{tc}) to the total number of defects (N_{tot}) reported during the function test cycle: $TCE = N_{tc}/N_{tot} \cdot 100$.

The author's test case effectiveness improvement framework was based on *test case escapes* which refers to 'software defects that a given suite of test cases failed to find but that were found as a side effect in the same test cycle' [6, p. 3 (83)]. The identified test case escapes depicts deficiencies in test cases. Hence, going forward the emphasis can be on such potentially deficit areas in order to improve the quality.

Palomba et. al. [23] also performed an extensive research on test code quality, driven by the motivation to find non-coverage related criteria which were able to improve the quality of automatically generated test code. For measuring the test effectiveness and maintainability, the authors investigated on two metrics: *test cohesion* and *test coupling*. With cohesion, the team tested how much the tested code belongs together i. e. is cohesive; this should not be a too low number as each test should only validate a very spe-

[2]In the following, we will use the term of effectiveness frequently. Therefore, we want to emphasize that effectiveness may not be mistaken with efficiency: *effectiveness* indicates whether a process is successful in determining the correct ore expected result whereas *efficiency* indicates in how good something can be achieved when considering the effort needed to do so.

cific functionality of the SUTin order to receive as precise results as possible. *Test coupling*, on the other hand, measures the overlap of different test cases; here, tests should overlap as little as possible to ensure maintainability under future changes. For that, the team derived two metrics based on core concepts: *Coupling between test methods* and *Lack of cohesion of a test method*. For evaluation of said metrics, the researchers used the EvoSuite[3] for test generation on the SF110[4] dataset, a dataset containing code samples of 110 representative and/or popular projects on SourceForge[5] provided by EvoSuite. After evaluation of the results, the team found out that there were some problems in automatic test generation using the already built in methods, although these already incorporated some quality control measures. Especially, it was found that the tests suffered from too low cohesion and to high coupling. Using their developed methods and implementing them in the test generation process, the authors were able to improve results in both fields. Furthermore, it was noted, that the inclusion of those metrics in the test generation had also a positive effect on branch coverage and test suite size.

3.2 Measuring Effectiveness of Test Suites

While the approaches mentioned in section 3.1 were developed with test cases in mind, most of them can be considered for measuring test suite effectiveness as well. This is because a test suite is a collection of test cases. Improving test case quality directly reflects in overall suite effectiveness. With test suites, a broader set of potential factors needs to be considered which usually do not arise when individual test cases are examined: size of test suite and relevance to the SUT. On the positive side, it provides a wider scope for research in terms of understanding relationship between various factors like statement coverage, branch coverage, block coverage and other similar notions. They determine the percentage of statements, branches and blocks of the source code of a program that are covered by a particular test suite during testing respectively. For an instance, `if .. else` block comprises of 2 branches, whereas `if .. elseif .. else` comprises of 3 branches. Furthermore, a block refers to the atomic keywords used in building a program statement. Unreachable program locations can be identified if no test case could be written so that the specific part of the code is accessed.

Laura Inozemtseva and Reid Holmes,[10] from University of Waterloo studied the relationship between statement coverage, decision coverage and modified condition coverage against suite effectiveness. They claim it to be the largest study with 31,000 test suites for five systems with 724,000 lines of source code in Java. The suite's mutant detection ability was considered as the characteristic to measure the effectiveness. As a major finding, the study showed low to moderate correlation between coverage of a test suite and its effectiveness, when the suite size is controlled. Also, the authors claim that the type of coverage used had little impact on the strength of the correlation. However, though branch and decision coverage are very similar, branch coverage takes into account unconditional branches as well, which wasn't accounted in this study.

Experiments by Gupta and Jalote [8] determined the significance of branch coverage over block and predicate coverage. The team studied the effectiveness and efficiency of block coverage, branch coverage, and predicate coverage with mutation kill of test suites as adequacy measure to evaluate. In their tests, effectiveness refers to the fault detection ability of a test suite, and efficiency refers to average testing cost incurred for detecting fault in a program. Their results suggested the existence of a trade-off between effectiveness and efficiency. Predicate coverage was most effective but least efficient in fault detection. And block coverage was most efficient in terms of testing cost but least effective in fault detection. But branch coverage performed consistently with good effectiveness and efficiency. Wei et. al. [30] explored in this direction to study the effects of branch coverage in suite effectiveness.

Wei et. al. [30] conducted studies to understand the relationship between branch coverage and suite effectiveness in random testing. Random testing is an useful strategy to test a software when there is no clear or sufficient information to perform the testing. The testing was carried out for 2520 hours on 14 Eiffel classes using fully automated random testing tool AutoTest[6]. AutoTest is a open-source framework for performing automated testing widely used by Google, IBM, Red Hat, and many others. A major finding from the study was, branch coverage was not a good indicator for suite effectiveness. Branch coverage was used as adequacy criterion to stop the testing once testing reaches specific coverage levels. In the initial 10 minutes of the test most of the branches were exercised, while the uncovered faults increased gradually. Almost 50% of the uncovered faults were when the branch coverage was almost stagnant. However, branch coverage was widely considered as stopping criterion in random testing. The authors evidently proved with statistical results that branch coverage increases rapidly but the uncovered faults only increase gradually. Hence, using branch coverage to measure test suite effectiveness would end up with inaccurate values. As an extension, Namin and Andrews studied the influence of size on test suite together with structural coverage.

Namin and Andrews [21] proposed that correlation between coverage and effectiveness can differ for different SUT, but the combination of coverage and suite size yields more accurate results in determining the effectiveness of a test suite. The authors initially performed an analysis of covariance (ANCOVA) to verify if either of both influenced effectiveness. The p values were always less than 0.001 indicating high correlation between coverage, suite size and suite effectiveness. Test suite size is determined by the number of test cases in a test suite. In general, suite size grows in proportion to the functionalities offered by the product, due to increase in test cases. The linear models built by ANCOVA were then used for building multiple regression models to determine the effectiveness. Among the models, the one with $AM = B1 \cdot log(size) + B2 \cdot coverage$ had higher adjusted R^2 values indicating it best explains the effectiveness where *size* refers to the test suite size, *coverage* refers to the coverage percentage, AM refers to *mutants adequacy ratio*, B1 and B2 are coefficients.

As it seems that coverage can, in some cases, be used as an indicator for effectiveness, we also investigated on code coverage in order to understand its influence better.

A research group of Antinyan et. al. [1] in collaboration

[3]`http://www.evosuite.org`

[4]`http://www.evosuite.org/experimental-data/sf110/`

[5]`https://sourceforge.net/`

[6]`https://autotest.github.io/`

with Ericsson constructed a study on the quality of code and its related tests with special regard to the code coverage. The goal was to find, if a high code coverage also indicates a correlation with defects in code. For that, an analysis on a relatively large in-house telecommunication project of Ericsson with approximately 2 million lines of code (LOC) was conducted. For this analysis, the team gathered information about all encountered bugs per file in the course of one year. As it was found, the defect data was relatively stable so that the actual evaluation was conducted only on a smaller sample set. After the evaluation, the researchers found the following: only a small correlation between the (statement, function or decision) coverage and the number of defects could be noted (approx. -0.20 for Pearson Correlation Coefficient or approx. -0.15 as for the Spearman Coefficient, respectively). With this result, it could be argued that coverage, regardless whether it is statement, function or decision coverage, does only help to a very small extent, if at all, in finding defects in the examined code. However, not only coverage was inspected but other metrics as well. Some of these yielded a much higher correlation with the number of defects found. For example, the change rate of certain files in terms of the number of versions (0.79/0.62)[7], the LOC of the inspected files (0.67/0.53), the age of the file (0.31/0.27), or lastly the block depth (0.42/0.42). Interestingly, the amount of comments added to a file does not correlate at all with the number of defects. Other metrics, such as the amount of changed code (0.61/0.55) and the number of developers of a certain file (0.76/0.63) also indicated a high potential for introducing defects. In summary this means that long and deeply interleaved files disclose a high tendency for defects. But also files “under intensive development” or from a large team of different programmers introduce a high risk for defects. And ultimately, the pure amount of test coverage does not necessarily guarantee successful detection and/or prevention of defects in the system under test.

While the above mentioned research investigated on the code coverage in general, Holmes and Inozemtseva both also worked together with René Just and Gordon Fraser on an analysis of mutation testing for detecting real faults in software considering code coverage in 2014. The results are found in [13] as a summary and more detailed in [14]. They conducted a study in which they examined 350 defects from publicly available software repositories[8] with more than 320k LOC in total and to which they introduced over 200k defects by using mutation. The goal was to identify if mutant testing is suitable for actually detecting bugs or other real-life defects. In order to answer that question, the team examined the bugs found in the aforementioned projects and compared the original version with the one containing the fix. After that, automatic test suites were generated basing on the rectified versions which then tested a mutated part of the software. The researchers found out that an increase in mutation score (i.e. the percentage of mutants detected) shows significant correlation to an increase in statement coverage. Furthermore, the researchers investigated the two ends of the spectrum where a test contributed to an increase in test coverage and when it did not. In particular, the researchers noted that in the first case, it detected more than 10 additional mutations in 35% of the tests, whereas on the opposite side over 40% of the tests were not able to detect any additional mutations. Concluding, the team found that generated mutants generally are coupled to real-life defects and can be detected relatively well. However, considering the defects which were not detectable in mutation testing can, according to the authors, be roughly categorized in three sections: “faults requiring stronger mutation operators”, “requiring new mutation operators” or lastly “faults not coupled to mutants”[14, p. 7 (660)]. Whereas most of these undiscovered defects might be detectable with the addition of further mutation operators, especially the faults from the last category are probably not detectable by mutation analysis in general as these require algorithmic or other changes. Lastly, the team stated that it was able to show a significant positive correlation between mutant detection and the detection of real defects.

Based on these findings, Holmes and Just continued the research together with Gordon Fraser from the University of Sheffield with regard to the efficiency and effectiveness of mutation testing. The results of their study are also summarized in [13] and explained more extensively in [12]. The two major constraints of mutation testing, at least according to the authors are, scalability and equivalency of mutants. For that, the team proposed three main optimizations in mutant generation: firstly, the researchers proposed a dynamic analysis approach in order to detect whether a certain mutant is able to infect a program state, meaning that it actually changes the outcome and/or behavior of the current test. The second goal is to check whether a mutant is not only able to infect the current scope but is also able to propagate and infect other tests. Thirdly, it is considered, if two mutations result in the same valuation. In order to evaluate how these optimizations effect testing, the team examined an extended list of open-source software projects from above. They implemented their optimizations as extension of the Major Mutation Framework[9] and additionally created 20 random tests for each class supplementing the already existing test suites from the projects themselves. Ultimately, their results show that with these optimizations, the researchers were able to increase the effectiveness and efficiency of the tests. It can be noted that the partitioning and propagation tests yielded in increased reduction of test code. Furthermore, the researchers achieved a speedup of mutation analysis by up to 20% by only using state infections as reduction measure. When considering all optimizations working hand in hand, the analysis time could be reduced by around 40% compared to plain coverage optimization.

While code coverage is one possible measure to get an indication of effectiveness of the tests, there are also others to think about. For that, we now want to shed some light on such in the upcoming paragraphs.

Regarding the topic of software testing metrics, Lazic and Mastorakis [17] presented an extensive study in 2008 covering the entire software development life cycle (SDLC). In their paper, the authors described where and when to use metrics, and gave proposals on what metrics to deploy. For example, in every state of the software project, metric data should be gathered serving two main purposes: on the one hand, track progress and on the other hand give other

[7]first value is Pearson correlation coefficient, the second one is Spearman

[8]*JFreeChart*, *Closure Compiler*, *Apache Commons Math & Lang* and *Joda-Time*

[9]`http://mutation-testing.org/`

team members (e.g. the project manager) information on advancements or regressions. Furthermore, tips and guidelines are given on how to construct useful metrics. In particular, metrics should be kept simple and ultimately follow a project goal. Metrics not following a goal of the project do not contribute to the advancement and are a waste of time. Going hand in hand with simplicity in definition, metrics should also be constructed unequivocally, meaning that it is irrelevant who is performing the test with regard to the outcome. As minimal test metrics, which should be applied in every software project, the researchers proposed the following: *number of test cases, their executions as well as passed and failed cases, test cases under investigation, cases that are blocked or re-executed, first run failures, total executions, total passes and total failures and lastly execution time of both test cases and individual tests.* Based on these, the team further proposed some derivative metrics, which help to improve effectiveness of testing in the SDLC. After introducing these metrics, the authors also give advice on how to interpret the gathered information, not only in terms of actual software quality or progress but also from a business perspective, in particular issues like resources, cost, growth and stability. Furthermore, it is also suggested to keep detailed information about defects, which were located in the project. With this valuable information, further progress can be tracked and problematic components or other abnormal phenomenons can be gleaned. Lastly, various proposals on test management, test execution and incorporating tests as central part of the SDLC were proposed.

Similarly, the research team of NIEDERMAYR ET. AL. [22] compared 14 different open source projects with regard to their code coverage and general test quality. During testing with a mutation approach to selectively remove program logic to introduce a defect, they found problems with pseudo-tested methods. In this case, a pseudo-tested method indicates that there is a test for a specific method, however it either does not get validated or in a false/erroneous way. This indicates, that a method is tested incorrectly and thus might not correctly recognize faults. This is especially the case for system tests, which tests a certain system in its whole in comparison to unit tests. During their testing, it was found that, around 10 % to 19 % of methods tested in the unit tests showed pseudo-testing characteristics with an average of 11 % and a deviation of 6 %. This led the researchers to conclude that, method coverage may be a good indicator of unit test effectiveness, however this does not hold for system tests. Here the team found a higher ratio of pseudo-tested methods, ranging from only 6 % to almost 72 % with strong deviation (21 %) from the mean of 34 %. Furthermore, the researchers noted, that "more than half of the pseudo-tested methods were of medium or high severity" [22, p. 6 (28)].

3.3 Techniques to Improve Test Effectiveness

In general, prominent research in this direction aims to measure and improve the effectiveness of test cases, test suite and studying different properties. However, these have a causal implication on improving test suite quality and directly contribute to the problem of qualitative evaluation and comparison among test suite's effectiveness. There is also considerable amount of research conducted in order to formulate an entirely new and effective frameworks to keep abreast with emerging trends. We initially present such new development strategies, algorithms, data structures and other emerging trends demanding a new testing technique, followed by methods to scrutinize existing techniques to build effective test suites.

For an instance, in recent years multiple forms of agile software development processes like Scrum, Extreme Programming (XP) and Feature Driven Development (FPP) are extensively in practice. RAHMAN ET. AL. [26] proposed an approach for *Behavior-Driven Development (BDD)*, which is an agile software development methodology used frequently in micro-services development, according to the authors. In BDD, the expected behavior of a system and its acceptance criteria are described in the form of scenarios using a simple business readable syntax using *Given, When and Then* keywords as follows:

Given some initial context,
When an event occurs,
Then ensure some outcomes

These keywords are called *steps*. Each of these steps are parsed and executed by the BDD test framework to verify the software's expectation. If the modules after development complies with the scenarios, an implicit acceptance testing is already performed. This alleviates the need, or in some cases reduces the overhead of additional acceptance testing. The entire range of test cases pertaining to acceptance testing is then eliminated from the test suite, eventually bringing down the test suite size. Size of test suite is among the factors contributing to the effectiveness of a test suite. The authors further go on to propose a *Reusable Automated Acceptance Testing Architecture* (RAATA). The core problem discussed is not directly relevant to test suite effectiveness, but although as a repercussion effect, the quality of test cases would be improved.

Likewise, an ubiquitous use of *Deep Neural Networks (DNN)* in machine learning based software is witnessed off late. Traditional software testing cannot be applied to DNNs due to its complex structure. Formal coverage criteria are not applicable since a specific test case might have varying coverage in different times due to backward/forward learning mechanisms. The complications in testing such software is exponential with complicated process of developing test cases. SUN ET. AL. [27] describe a testing strategy together with *Sign-Sign (SS)*, *Sign-Value (SV)*, *Value-Value (VV)* and *Value-Sign (VS)* metrics tailored to structural features of DNNs with ReLU activation function to thoroughly test and evaluate the quality of the test suite. Sign and value changes for each activation function input are considered as the different cases to be tested. The core idea is to have individual test cases pertaining to every pair of features with all possible combinations of sign and value changes. Authors claim the test results to be promising, with the covering methods identifying a significant portion of adversarial examples. Thus, the aforementioned metrics are to examine the unique properties of DNNs. In this context, considering these additional metrics for DNNs evidently improves the test suite effectiveness.

The researchers JABBARVAND ET. AL. [11] from Department of Computer Science at the University of California came up with energy-aware test suite minimization technique. They considered energy as a property of interest and an energy-aware coverage criterion that indicates the degree to which energy-greedy segments of a program are tested. The core idea was to divide the program P into p segments

$S = s_1, s_2, ..., s_p$. Test cases in a test suite $T = t_1, t_2, ..., t_n$ are represented as vectors $V_{ti} =< v_{i,1}, ..., v_{i,mi} >$ where $v_{i,j}$ with value 1 indicates that t_i covers an energy greedy segment t_j and 0 if it covers a normal segment. Moreover, the metric *eCoverage* is calculated for every test case that determines the degree to which the energy-greedy segment is covered. The main components of this approach are coverage calculator to calculate eCoverage and the actual test suite minimization. Then the researchers investigate to find the smallest subset $T' \subset T$ without hampering the effectiveness of the original test suite. This final task was modeled in two different approaches: by integer linear programming and greedy algorithm (based on heuristics) for quicker results. The team demonstrated this technique on real world application from *F-Droid*[10]. The conducted experiments provided statistical evidence supporting the ability to reduce the test suite size on an average to 84% of its original in the case of IP and 81% in the case of greedy, while maintaining the suite quality.

Regression test suites often become as large and complex as the software itself due to continuous software evolution. If the test suites are not constantly maintained then ultimately the effectiveness is hampered. The researchers TENGERI ET. AL. [28] proposed a novel approach *TAIME (Test Suite Assessment and Improvement Method)* for test suite assessment and improvement that utilizes code coverage information, but on a more detailed level and adds further value aspects derived from the coverage. They plotted a binary coverage matrix with test cases and program elements such as statements or functions according to the chosen level of granularity. Test cases and the program elements are decomposed further into coherent logical groups called *functional units.* The subsets of test cases are called *test groups* and subset of program elements *code groups.* The numbers in the cells of the matrix represent the code coverage ratios that a test group attains with respect to the given code group. By visualizing it as heat-map they were able to pin point potential problems where the overall coverage is too low or which functional units were less coherent. This makes it easy to identify the potential test cases for removal or refactoring. Further, they proposed following metrics:

- *Coverage Metric (COV)* to determine the percentage of test cases satisfying the program elements, *Partition Metric (PART)* characterizes how well a set of test cases can differentiate between the program elements based on their coverage information
- *Tests per Program elements (TPP)* to determine the number of test cases created on average to test a set of program elements (procedures, statements, etc.)
- *Specialization metric (SPEC)* depicts how specialized a test group is to a code group
- *Uniqueness metric (UNIQ)* to measure the portion of the covered elements that are covered uniquely by a particular test group.

Together with these metrics, they propose an iterative approach through multiple phases which measure the coverage effectiveness. Based on the evaluation, test and code groups are updated after every iteration until cohesive test cases are clustered under the same test group, ultimately improving the effectiveness of the test suite. The authors demonstrated the approach with the application of TAIME on SoDA (Software Development and Analysis framework) library for improving its regression test suite. SoDA is an open source library and tool set that aims to provide researchers and practitioners a framework with which various code coverage-based analyses can be performed.

Test case minimization is another famous approach that helps in reducing the test suite size by eliminating less significant, redundant cases. These cases are identified through heuristics-based criteria, code and data-flow information, execution costs and many other similar criteria. For instance, the greedy approach (GRD), Harrold Gupta Soffa (HGS), Delayed greedy (DGR), 2-optimal greedy (2OPT) are all approaches that target test case minimization. Along the same lines, researchers MARCHETTO ET. AL. [19] conducted comprehensive experiments taking into consideration the requirements coverage and test suite execution cost, together with code coverage. They proposed a three dimensional approach called *Multi-Objective test suites reduction (MORE+).* The first dimension uses information on how test cases exercise the under-test application. The second dimension concerns how test cases exercise business application requirements while the third dimension considers the time taken to execute test cases. It is more of a refactoring approach to determine a subset of test cases S_{red} from the original test suite S. This reduced subset S_{red} comprises of those test cases that cover the mentioned three dimensions with a higher likelihood. An extension of this high priority reduced test case set S_{red} with new cases substantially improved the quality of the suite. About the effectiveness, statistically significant improvements were shown by MORE+ approach against the traditional strategies in terms of test suite's execution time and fault detection capability, but not the overall size.

On same lines, as a replacement to using greedy approaches for the same problem of test suite minimization/reduction, LIU ET. AL. [18] propose to apply k-medoids clustering. Previously, researchers conducted the study using k-means clustering. There were shortcomings caused due to the instability of the algorithm and its inability to take into consideration the actual code coverage criteria. Hence LIU ET. AL. enhanced it with k-medoids clustering algorithm and actual code coverage criteria. The idea is to consider cyclomatic complexity and code coverage rate as the axis dimensions. Every test case will have values generated for both metrics in the initial run. The original test suite is called R, and the test cases in R are called the data objects D. Randomly selected D acts as the center of the cluster and over the iterations, the focal points encoding the test case with equal distances are removed from the cluster. This ensures that there is no redundancy and every test case is unique within a cluster. The paper uses greedy approach in forming the cluster representing the set of test cases. The claim was quantitatively proven by demonstrating the application of this approach and comparing the results against traditional HGS and reduction using k-means. The authors claim that the results showed improvement not just in test suite size, but also in terms of efficient coverage and error detection rate.

To assess the effectiveness of testing in real world development teams, the researchers of CHEN ET. AL. [5] conducted a

[10]`https://f-droid.org/en/`

study with IBM to evaluate the effectiveness metrics proposed for the research and development team for the "IBM Electronic Commerce Development (ECD)". They proposed three different metrics which together form an evaluation framework. These three metrics shall – according to the authors – only be used as a whole and not stand-alone, as these metrics only tell a picture as a whole. The metric categories are (1) Quality-of-Code, (2) Time-To-Market, and (3) Cost-To-Market metrics. There are many sub criteria such as (a) quality of code, measured by weighted defects in tests plus weighted defects in production divided by the lines of new or changed code. A lower number indicates fewer and/or less severe defects. Or, (b) quality of product considers the weighted errors in the final product divided by the new/changed lines of code (LOC). This metric is similar to (a), however takes into consideration the code of the already shipped version. Last but not least, (c) test improvement is measured by the weighted errors found by the test team divided by the changed/new LOC. This measures the relationship between the errors found in testing and the size of the finished and shipped product. A high score indicates a higher number or more severe defects and ultimately indicates also the effectiveness of the test team. With their analysis, the team discovered that the quality of the tested software between the two releases had gotten worse, took more time to test and that the ratio of testing cost to development cost was lower. These results indicate that, on the one hand, the testing team got more efficient and, on the other hand, the second version tested seemed to have a significant increase in complexity. From these results, the researchers concluded, that the proposed metrics helped in identifying shortcomings as well as improvements in software testing effectiveness of the IBM teams and proposed to include the metrics in further development processes. However, it was also noted, that the proposed metrics do not seem to be sufficient enough in giving suggestions for the process improvement.

While these proposed metrics can form one possible method in assessing test effectiveness, also other testing methodologies can play an interesting part in doing so. In the following, we want to go through some select interesting methodologies to present.

The research team of Bures and Ahmed [3] investigated the effectiveness of combinatorial interaction testing (CIT) in software testing. Here the two researchers proposed the use of CIT methodology and investigated in how far the number of specific test variable combinations is necessary in order to reliably predict defects in software. Therefore, the authors examined different folding techniques and came to the conclusion that ultimately a three-way combinatorial test offered a high recognition value of mutants introduced into the system under test, whereas a two-way combination was insufficient.

While the testing concept of symbolic execution was first introduced in the late seventies, and therefore is far from new, it only became technically viable and popular in recent years. With symbolic execution, a system under test is not executed with actual variables but with symbolic placeholders instead. Once a diverging point, e.g. an `if... else...` statement, is reached, the divergence parameters are added to a set of constraints exactly describing the path taken. When the symbolic execution then has reached all possible paths or consumes all its computational resources, the path constrains then are evaluated with a solver to receive actual valuations of the variables. With that the paths can be reconstructed and defects in the code can easily be found. This is especially helpful for developers in order to reproduce defects. However, while symbolic execution can be advantageous to use, there are also some limitations to it. Mostly, it is hard, if not impossible, to evaluate certain types or functions in a program. This leads to untestable code, as the states can simply not be modeled. Additionally, symbolic execution has a problem, once paths are introduced which can be infinitely long or if there are too many possible branches. This means that symbolic execution does not scale well.

As symbolic execution offers an interesting approach for software testing, the two researchers Cadar and Sen [4] investigated the back-then current state of symbolic execution in the year 2013. In their paper, the authors presented techniques of symbolic execution used to counteract some of the downsides, gave insights in problems and ongoing challenges, suggested some solutions, and finally, presented popular tools for symbolic execution testing. After an extensive introduction to symbolic execution, the researchers presented two techniques for improving the use of symbolic execution. First, *Concolic Testing* is a technique, where symbolic execution gets mixed with concrete execution. For that, the initial valuations are (randomly) selected prior to the test execution. During the execution, the branching behavior is recorded and once the test has reached an end, certain constraints are picked by a tree search algorithm and inverted to selectively direct the execution in another, yet undiscovered branch of the execution tree during the next execution. That way, the critical valuations are generated through actual testing and probing only a few constraints need to be evaluated. The second approach, *Execution-Generated Testing* also uses mixing of concrete and symbolic executions. Here, the testing algorithm decides whether all valuations in a current step are discrete or symbolic. If all are discrete, the testing continues discretely, as if the program would be executed in that moment, but if at least one of the variables has a symbolic valuation, a symbolic evaluation of the next processing step is used. Both these techniques reduce the computational cost of symbolic execution as they make use of actual discrete valuations. Furthermore, these dynamic approaches also reduce or completely eliminate problems with parts of the code, that cannot be tested via pure symbolic execution.

As already mentioned above, symbolic execution has some problems. In the next part of their paper, the authors reasoned about these challenges: Usually, the amount paths increase exponentially with the number of branches in the code. As the computational cost for exploring all of those paths, only a select however important number can be visited in a given time frame. According to the authors, there are two main methods to countermeasure path exploration: *heuristic measures* and *sound program analysis*. The former one uses, as the name suggests, heuristics to prioritize paths over others. Often, these heuristics are selected in such a way, that high branch or statement coverage is reached. However, also other approaches can be effective such as static control-flow graphs – guiding the exploration to undiscovered branches using graphs metrics such as distance or execution times –, random testing – make key decisions based on randomness, e.g. select a branch randomly

– or evolutionary search – using a fitness function for decision making. The second approach uses methods form the program analysis and verification theory. Here, it is possible to use the constraint solver in order to make certain decision. This might work fine, but also leads to few complications. Alternatively, one tests the program repeatedly and checks if two branches converge together, in which case one can assume that after their convergence, they will result in the same execution path. Next, the researchers explore the problems of constraint solving. According to Cadar and Sen, the satisfiability checking and solving of clauses generated oftentimes is the main bottleneck. As a solution, one needs to try constraint elimination and incremental solving. In the former, the unused or irrelevant variables are eliminated from the set of path constraints to shrink it and in the latter, clauses from the constraint set are reused in order to eliminate duplicate solving and thus speed up the processing. As last two challenges, concurrency and memory management are named. Lastly, the researchers conclude their paper with a set of several tools used for symbolic execution testing, each with their advantages and disadvantages.

Most of the papers presented a platform-independent and often generic approach. Many of those can also be applied for testing mobile platforms, such as apps. However, we found only a few papers concerning those in particular. Nevertheless, we want to include also the platform dependent research in order to give a better understanding on how test effectiveness is tackled here.

Regarding mobile platforms, PATEL ET. AL. [25] examined the testing framework *UI/Application Exerciser Monkey*[11] or *Monkey* for short with the goal to evaluate if it can effectively generate and conduct test on Android applications regarding coverage. The researchers conducted experiments on actual applications with a large user-base and logged the behavior during the tests. To the surprise of the authors, all applications crashed under test load, some of them even after just a few seconds (e.g Instagram[12] after just 7 seconds or Splitwise[13] after 12 seconds), whereas the average time until a crash was 85 seconds. Furthermore, it was found out that the parameters of Monkey had no significant influence on the effectiveness, i.e. coverage, in the specific test environment. While testing the granularity of the produced test coverage, it was additionally noted that the testing with Monkey offered a rather simple and quick way to produce block coverage. This is, according to the authors, due to a mostly shallow method structure, supported by a low path count in the method control flow. In conclusion, this suggests that a coarse coverage at class or method level already offers a good indicator for fine-grained coverage. Additionally, the team found out that in comparison to manual tests, Monkey only did two to three percent points worse than the manually generated tests, which in the analyzing scenario, was insignificant. Finally, the authors presented that throttling the testing actions also had no significant effect on the coverage in the given test case.

4. DISCUSSION AND CONCLUSION

As shown by various research presented in this paper, one can reassuringly say that improving test suite effectiveness is a crucial part contributing to the quality of software. Factors like cohesion and coupling between test cases, coverage criteria and other similar notions influence the effectiveness of test suites. Also there are factors like test suite size that influences the suite's effectiveness from the maintainers point of view, but while executing the test it actually influences the efficiency and not the effectiveness. Taking these factors into consideration could lead to improvement in effectiveness of test suite. For an instance, BDD test framework could be improved by considering cohesion and coupling of test cases within the suite. Likewise, test framework for testing Deep Neural Networks (DNN) could be improved by controlling test suite size. Due to existence of linear positive correlation between suite size and number of neurons, increase in neurons would lead to increase in test suite size. Furthermore, we observed that software testing metrics play a core part in software testing. These metrics should be kept simple, easily understandable, coherent, cohesive and only loosely coupled with other test cases. This enormously helps to keep tests up to date with the code base to be assessed.

However, metrics about the assessing code, i. e. the testing code, need to be taken with a grain of salt. Research has shown that high results in test coverage, which indicate how much of the underlying code is tested, do not always indicate successful or effective testing. Oftentimes, the tests are faulty, e. g. bugs might not be correctly identified, but contribute to the coverage nevertheless. Therefore, it is important to not only rely on coverage criteria as a measure of test quality. Constant improvement and inspection of test code and test effectiveness is needed to ensure the best outcome and quality of the software project.

5. FUTURE WORK

While we did a relatively broad research and examination on the topic of factors contributing to the test suite effectiveness, it most definitely is not exhaustive. Most of the frameworks consider test suite size and coverage criterion to determine effectiveness. But instead, determining effectiveness by assertion quantity and assertion coverage as suggested by YUCHENG ZHANG AND ALI MESBAH [31] would be more appropriate. This calculated effectiveness value could then be used for test case prioritisation to reduce the suite size. Furthermore, technologies are constantly changing and new methods for improving software testing will eventually arise. This consequently means that these might out-date the results, we were able to gather so far. Therefore, constant pursuit of this topic is needed.

6. REFERENCES

[1] V. Antinyan, J. Derehag, A. Sandberg, and M. Staron. Mythical unit test coverage. *IEEE Software*, 35(3):73–79, may 2018.

[2] L. Briand and D. Pfahl. Using simulation for assessing the real impact of test coverage on defect coverage. In *Proceedings 10th International Symposium on Software Reliability Engineering*. IEEE Comput. Soc, 1999.

[3] M. Bures and B. S. Ahmed. On the effectiveness of combinatorial interaction testing: A case study. In *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE, jul 2017.

[4] C. Cadar and K. Sen. Symbolic execution for software testing. *Communications of the ACM*, 56(2):82, 2013.

[11] `https://developer.android.com/studio/test/monkey`
[12] Social Network. See: `https://www.instagram.com/`
[13] Expense Manager. See: `https://www.splitwise.com/`

[5] Y. Chen, R. L. Probert, and K. Robeson. Effective test metrics for test strategy evolution. In *Proceedings of the 2004 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '04, page 111–123. IBM Press, 2004.

[6] Y. Chernak. Validating and improving test-case effectiveness. *IEEE Software*, 18(1):81–86, 2001.

[7] P. G. Frankl and O. Iakounenko. Further empirical studies of test effectiveness. In *Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering - SIGSOFT '98/FSE-6*. ACM Press, 1998.

[8] A. Gupta and P. Jalote. An approach for experimentally evaluating effectiveness and efficiency of coverage criteria for software testing. *International Journal on Software Tools for Technology Transfer*, 10(2):145–160, jan 2008.

[9] IEEE Computer Society, P. Bourque, and R. E. Fairley. *Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0.* IEEE Computer Society Press, Los Alamitos, CA, USA, 3rd edition, 2014.

[10] L. Inozemtseva and R. Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*. ACM Press, 2014.

[11] R. Jabbarvand, A. Sadeghi, H. Bagheri, and S. Malek. Energy-aware test-suite minimization for android apps. In *Proceedings of the 25th International Symposium on Software Testing and Analysis - ISSTA 2016*. ACM Press, 2016.

[12] R. Just, M. D. Ernst, and G. Fraser. Efficient mutation analysis by propagating and partitioning infected execution states. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis - ISSTA 2014*. ACM Press, 2014.

[13] R. Just, M. D. Ernst, and G. Fraser. Mutation analysis for the real world: effectiveness, efficiency, and proper tool support. In U. Aßmann, B. Demuth, T. Spitta, G. Püschel, and R. Kaiser, editors, *Software-engineering and management 2015*, pages 53–54, Bonn, 2015. Gesellschaft für Informatik e.V.

[14] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014*. ACM Press, 2014.

[15] G. M. Kapfhammer and M. L. Soffa. Using coverage effectiveness to evaluate test suite prioritizations. In *Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies*. ACM Press, 2007.

[16] P. S. Kochhar, F. Thung, and D. Lo. Code Coverage and Test Suite Effectiveness: Empirical Study with Real Bugs in Large Systems. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2015.

[17] L. Lazic and N. Mastorakis. Cost effective software test metrics. *W. Trans. on Comp.*, 7(6):599–619, 2008.

[18] F. Liu, J. Zhang, and E.-Z. Zhu. Test-suite reduction based on k-medoids clustering algorithm. In *2017 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)*. IEEE, oct 2017.

[19] A. Marchetto, G. Scanniello, and A. Susi. Combining code and requirements coverage with execution cost for test suite reduction. *IEEE Transactions on Software Engineering*, 45(4):363–390, apr 2019.

[20] G. Myers. *The Art of Software Testing*. John Wiley & Sons, Hoboken, N.J, 2004.

[21] A. S. Namin and J. H. Andrews. The influence of size and coverage on test suite effectiveness. In *Proceedings of the eighteenth international symposium on Software testing and analysis - ISSTA '09*. ACM Press, 2009.

[22] R. Niedermayr, E. Juergens, and S. Wagner. Will my tests tell me if i break this code? In *Proceedings of the International Workshop on Continuous Software Evolution and Delivery - CSED '16*. ACM Press, 2016.

[23] F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, and A. D. Lucia. Automatic test case generation: what if test code quality matters? In *Proceedings of the 25th International Symposium on Software Testing and Analysis - ISSTA 2016*. ACM Press, 2016.

[24] H. Park, H. Ryu, and J. Baik. Historical value-based approach for cost-cognizant test case prioritization to improve the effectiveness of regression testing. In *2008 Second International Conference on Secure System Integration and Reliability Improvement*. IEEE, 2008.

[25] P. Patel, G. Srinivasan, S. Rahaman, and I. Neamtiu. On the effectiveness of random testing for android. In *Proceedings of the 13th International Workshop on Automation of Software Test*. ACM Press, 2018.

[26] M. Rahman and J. Gao. A reusable automated acceptance testing architecture for microservices in behavior-driven development. In *2015 IEEE Symposium on Service-Oriented System Engineering*. IEEE, mar 2015.

[27] Y. Sun, X. Huang, D. Kroening, J. Sharp, M. Hill, and R. Ashmore. Structural test coverage criteria for deep neural networks. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, may 2019.

[28] D. Tengeri, A. Beszedes, T. Gergely, L. Vidacs, D. Havas, and T. Gyimothy. Beyond code coverage: An approach for test suite assessment and improvement. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, apr 2015.

[29] A. Vahabzadeh, A. M. Fard, and A. Mesbah. An empirical study of bugs in test code. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, sep 2015.

[30] Y. Wei, B. Meyer, and M. Oriol. *Is Branch Coverage a Good Measure of Testing Effectiveness?*, pages 194–212. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[31] Y. Zhang and A. Mesbah. Assertions are strongly correlated with test suite effectiveness. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015*. ACM Press, 2015.

Automated Testing of Microservice-based Systems

Niklas Münzer
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
niklas.muenzer@rwth-aachen.de

Marcel Neis
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
marcel.neis@rwth-aachen.de

ABSTRACT

We provide an overview of microservice architectures their benefits and their challenges regarding automated testing strategies and test frameworks. Microservices are one approach to split an application into independent parts to increase the scalability and also to improve delivery problems. We focus on one major challenge which is the testability of microservices. This is also a problem in monolith applications, but the shift toward microservices also shifted the testing focus to more integration testing. The paper reviews various testing approaches regarding the test pyramid. The focus of the review is on functional-, fault-tolerance- and performance tests as these can be found on every layer of the pyramid. Also, new software like Screwdriver, which is being used to handle performance and fault-tolerance tests in large microservice systems, is reviewed. As microservices are highly scalable the testing complexity also increases. But testability should not suffer as a result and so a challenge is to keep it manageable. The review shows that each of these testing approaches is needed for a full system check, but applications might specialize in a specific one for results corresponding to their goals.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering
; D.2.9 [**Software Engineering**]: Management—*productivity, programming teams, software configuration management*

Keywords

Microservice, Distributed Systems, Automated Testing, Test Pyramid

1. INTRODUCTION

The recent trend to shift from monolith architecture toward microservice architecture solved a few monolith scaling problems, but also lead to new problems in software engineering. Monolith applications represent the classic design, every feature is include in one application. Compared to this, microservices attempt to split these features into small and independent parts. The aim is to increase the scalability by distributing independent services across multiple machines. But this scaling created new problems in the field of testing the developed software. Testing a monolith software included unit tests for the modules and integration between these modules, so mostly internal tests. In microservice architectures, testing shifted to more integration testing of distributed services. So the internal testing shifted to interservice testing with services physically decoupled. However, one microservice can have several units that also need to be tested on the unit test level as well as on the integration level. Systems can be tested for certain test aspects like functional tests or performance tests. The new structure of independent services communicating with each other leads to a new communication layer. Testing this layer appeared to be a new challenge in software engineering. Because more and more large companies like Google [1] and Netflix [2] apply the mircoservice based architecture, the interest in optimal test strategies and test frameworks increases. In the following paper, we introduce the microservice architecture, related it to the classic monolith architecture, show challenges to test these services and present testing solutions.

2. RELATED WORK

T. Clemson [1] describes a way to systematically testing distributed systems like microservices by referring to the testing pyramid. The author expand the pyramid by the additional layers: Component between Integration and End-to-End and Exploratory on top of the pyramid. The basic concept of Unit, Integration and End-to-End testing are described. Unit testing is split into sociable and solitary methods. Sociable as a black box test variation of unit testing. Solitary uses test doubles, which simulate real objects, on the connections from the service to its dependencies to test possible interaction errors. Another relevant part is the defined Component testing layer which separate a part of

[1]Google is an American multinational technology company that specializes in Internet-related services and products, which include online advertising technologies, search engine, cloud computing, software, and hardware
[2]Netflix, Inc. is an American media-services provider and production company headquartered in Los Gatos, California, founded in 1997 by Reed Hastings and Marc Randolph in Scotts Valley, California

the application and define new test boundaries, including the use of test doubles to check interactions. Also a important detail is the use of contract tests. Each time a service use another service a contract is defined which mark all values the given service requested. Now this contract is used to test if the service differ in its requests.

Savchenko et al. [2] aim to summarize challenges and microservice testing solutions. They split the testing process into three testing parts: component, integration, and system. The testing process is step by step described starting with the basic requirements. These are for automatic testing: a well-defined interface and usable testing components for each of the different code bases. In this context their solution is presented: an HTTP API based microservice testing service. The service provides extensions for different code bases and testing solutions like JUnit. Their results are that microservice testing is similar to Service Oriented Architecture(SOA) testing with added complexity especially in the infrastructure. The authors also specify that the complexity can further increase when the microservice uses different environments.

Ma et al. [3] describe their microservice testing approach using a graph-based solution with service dependency graphs. They aim to decrease upcoming problems in the early stages of development. Each service is represented by a node and each link to another service as an edge. This leads to a graph showing all the connections between the service. When Integration tests between these nodes are run, the results can be shown in the graph to simplify the search for the point of failure. The graph solution also aims to find cyclic dependencies which could result in a system crash or unlimited service calls. The authors show that their solution can handle small and large microservice-based systems and is able to handle complex service interactions.

3. MICROSERVICE ARCHITECTURE FOR DISTRIBUTED SYSTEM

Microservices are distributed systems, but their focus is to break down the components into small services. It is an approach to modulate monolith applications, to increase the scalability and cohesion for better maintainability. It reduces development time and coupling of the system. Classic monolith applications are limited in size and complexity, as the full application runs on a system. In large applications unnecessary or less frequently used features can increase the complexity and so reduce the performance. There was a need to improve monolithic applications. Such an approach to this problem is the use of microservices. Monolith applications are split into tiny parts, each as an individual service. These services each handle single functionality. The size is not limited, but to prevent scaling problems the size should be kept small. Each service can be modified to work on different code bases or hardware, increasing optimization. Microservices use horizontal scaling by distributing services across the hardware and adding multiple layers of each service. This scale-out from microservices handling monolith scaling problems. The scaling problems result from limited resources on the hardware. So the vertical scaling of a monolith application is always limited. Microservices use their independent service structure for the distribution and so scaling of tasks. When service requests a packet in a monolith application the corresponding answer is sent from a specific module. With microservices, there might be more than one layer of this service and so the requesting service does not have to care from which of these layers the answer comes. This results in the scaling of the communication rather than the hardware. Another idea for the use of microservices is to deliver ideas faster into features and optimize operations, e.g. databases for specific functionalities. These functionalities can be single operations, but can also cluster functions to modules to provide monolith style interfaces. These services can be quickly included without harming other parts of the application, as the services are loosely coupled. This allows independent development of these services. They can be seen as tiny applications with each of its own development teams.

3.1 Characteristics of a Microservice Architecture

One of the main characteristics of microservices is the shift from internal to external communication. The communication is often web-based with Hypertext Transfer Protocol (HTTP) as a RESTful API [4] and shifts toward event-based models. This trend leads to a full API based implementation of the services. Each service providing a single function. They commonly used HTTP interfaces to interact with other Microservices and can be realized by different topologies like bus, central controller or container-based [5]. The aspect of the communication between microservices differs from monolith applications, in these the communication was often fully internal. Internal in the aspect of communication between modules inside the application. With microservices, communication is based on inter-service communication, like an API. This leads to additional factors with an impact on communication like latency, bottlenecks in the bandwidth or timeouts. These factors have a small impact on physically near services e.g. routing inside a data center. But customers mostly use these services from outside the personal area network of a data center. This leads to different approaches for handling the communication: HTTP Interfaces like a RESTfull API or event-based.

REST API.

Representational State Transfer or short REST, is an attempt on handling the communication between services. The REST technology uses specific constraints to define a structured interface for individual use. The basic concept is based on a client-server structure and requests between these using HTTP (*Hypertext Transfer Protocol*) [4, 6].

- Client-Server: The use of an interface decouple client and server from the data storage. This means a client does not have to handle the underlying structure and the server not the user interface. The result is a simplified and scalable server structure.
- Stateless: Each request needs to send all necessary information to the server which are required to process it. So the stats are fully saved on the client and not on the server. That increases the independence of each resource on the server as they are not connected directly to a specific request. Used resources can quickly be freed and reused for further requests.
- Cache: Data packets can be marked as cacheable so a server can reuse this packet for later requests. This

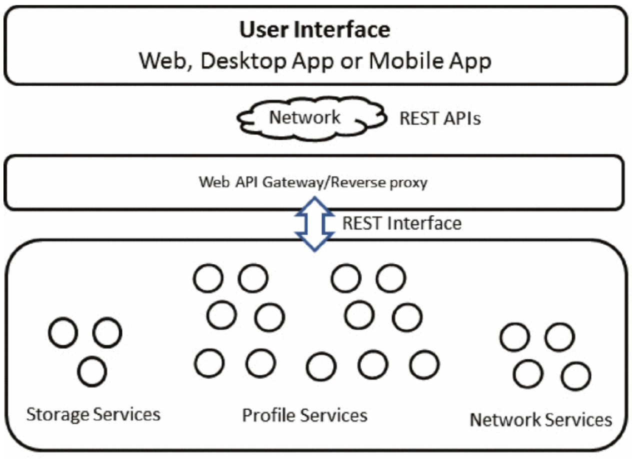


Figure 1: An example structure using REST API [6]

can lead to reduced packets sent but needs to be handled with care as an updating packet can be lost which results in false/outdated packets being requested.

- Uniform Interface: Using a standard and unique interface for all requests simplifies the system architecture. Requests can only use a specific format which prevents unknown options and furthermore increase the security. The downside of this feature is less efficiency as specific requests to services matching their implementation can increase the speed and amount of data send.
- Layered System: Each layer only can see components of the same layer, which results into caped complexity from the reduced amount of nodes known. Especially this prevents unnecessary information gain for an client, which increase the security of the system. On the server side there is the option to increase scalability by using load-balancing for the underlying layer.

The effect of the microservice communication is that it allows communication between physically decoupled locations. Also, REST is a rather secure communication option, so services can have public access. Figure 1 describes an example structure using microservices with a REST API. The users do not directly interact with the services. The communication is handled by a user interface connected to a web API gateway. Both the web API and the microservices are connected with REST solutions. This means that only specified requests are handled and with the Web API gateway also load balancing as a central node is possible. But this concept can lead to a single point of failure. The implementation of the inter-service as also the service to Web API connection can widely differ. Especially when the hardware solution has to be considered.

Event-based method.
The idea of event-based communication is that every action used by a service is handled as a trigger event. So when we have two services A, B, and A request something from B. Then B would answer directly, but there are multiple layers of B so there is no direct connection to one layer. So this request is sent as a broadcast to all layers of B and one of these answers the call. A does not have to care from which layer of B the answer came which reduces the communication overhead. This broadcasting would send a message to every service on the e.g. bus connection. Every service listening to this connection can then react on its own when an event occurs. Such an implementation of an event-based model is for example Serfnode [7]. Each node represents a service inside the microservice architecture. This system uses clustering of, also physically decoupled, nodes and advertise changes as events to these. The only requirement for this is that every node which wants to join a cluster must be reachable by an address, so a pair of IP and port. When a change occurs, e.g. a node join, leave or fail a broadcast is sent to the cluster to inform all nodes. Also on events like joins or address changes the new address of a node is broadcast to the cluster. The clustering of nodes reduces the number of events each node has to process, these scale up by the number of nodes joining the cluster. This is important because in large systems a large number of messages can slow down the service [8].

Another aspect of event-based communication is that each service has to listen to the broadcasts all the time. Otherwise, the service might be unavailable for an unknown reason. This means there must be a service that supervises the event communication and takes care when services fail to process an event, by e.g. restarting or excluding from the system when it might be compromised [9].
Each of these attempts has in common, that a system for handling the communication between the services is required. This also means that the effectiveness of this communication system directly impacts the effectiveness of the microservice. This leads to the shift from REST to more lightweight communication models, as these reduce the communication overhead. The basic problems of this external communication are related to the fact that microservices are distributed systems, so communication might fail sometimes. Especially with many services and a lot of messages between these services. This means that the internal communication from the monolith application was outsourced to the infrastructure of the communication [10].

Figure 2 shows the path from monolith applications toward microservices. The technology was not invented just in time, as more coupled variations were already used as so-called SOA, *Service-Oriented Architecture*. SOA was especially used in a web-based application and cloud services [5]. These can be seen as an early more coupled stage of microservices. The key feature used to be decoupled services with important functionalities and the option of monitoring these services. The architecture services are coupled to a central bus that manages and coordinates the services. This central bus is used to increase security, use load-balancing and for simplifying to add services. The structure support service reusability and service discovery. But these services are still not independent as they are in microservices, the coupling with the bus lead to dependencies between the services. So when the service needs to update the bus system it takes down all services as they are fully managed by the bus. Another problem is the size of these services because functions and resources might not be properly scaled [12].

3.2 Advantages of Microservices

The key benefits of microservices are scalability, independent development, synchronized changes, fast and continu-

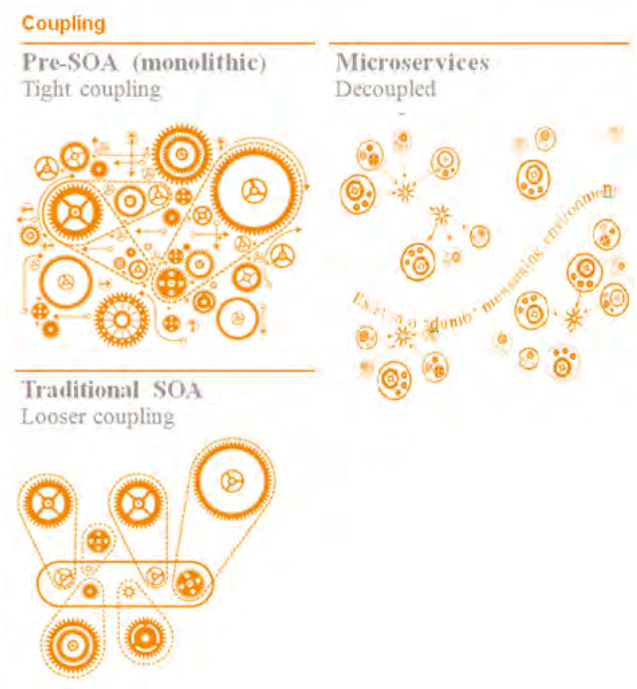


Figure 2: Monolithic vs SOA vs microservice[11]

ous delivery of new features and using different frameworks or libraries for different microservices [13, 14]. Monolith applications are limited scalable, for various reasons. One aspect is the size of the application as it has to be run fully on the devices. Increasing features or complexity will affect all users. Adding features in a microservice architecture can be realized as adding a new service. Such a change affects only the users of this specific service. This leads to the scalability as the application is not overloaded with unused services, the user chooses which of these services should be in the used application. Another scaling problem is based on the development, changes in monolith applications involve the whole development team. Each change must be verified to prevent possible errors. Also, a code error or interaction problem affects the full application. Using microservices can help in this case, when changing a service only the participating service developers and related services, basically, services and user which use this specific service, are involved. All other services are unharmed by these changes, so it is possible to independently develop services with the microservice architecture. With microservices faster and continuous delivery of features is possible. The faster aspect results from less involved developers in a specific change.

Monolith applications update the full application and for online services, this results in downtime for the service. This might be less of a problem for big applications with a few updates per year, but has a big impact on applications like cloud services. Continuous delivery can be essential for this type of service. This feature can be achieved by adding independent services or changing minor linked services inside the architecture.

3.3 Disadvantages of Microservices

Microservices also entail challenges like orchestration complexity in deployment of multiples services, managing and controlling tests for each service, the maturity of the team to coordinate changes, define the service boundaries and control shared libraries and code reuse [13]. There is no exact definition of the microservice size, so the definition can variate from a few hundred to a thousand lines of code. Overloading a service can lead to the same problems as using the classic monolith design. With the growth in size and complexity of the microservice, the number of managed and monitored services also increases. A central controller might be less usable with a scaled system, the number of occurring events harm the chance of choosing the right decision in real-time. Another aspect is the use of a monolith part as a central controller which leads to all downsides of the monolith architecture. Distributed systems are fragile and so the failure handling is a downside of microservices. In the classic monolith architecture just a few or none external calls where made. Services used to be clustered inside of packages and the majority of calls where internal. With microservices, the internal calls shifted toward the 'communication layer'. The components of the system decoupled into services and communication might fail. Failure sources can be the hardware, local unavailable systems or a message overload of the service. Especially, the external communication can timeout as the services might be physically decoupled. Another problem is located in the development. The independent development of services can lead to different interpretations of the functionality, complexity or architecture. Each development team working on a different service still has the opportunity to communicate future plans. This includes the structuring and distribution of functionalities across all services. This can lead to multiple functionalities of equal services, with different code handling.

4. AUTOMATED TESTING OF MICROSERVICES

Testing a microservice-based application differs from testing a monolith application. Monolith applications and their parts work on the same basis, a framework. With the invention of microservices, these parts of the application got outsourced to small services. So the internal testing of a part like a package is equal to an integration test of this part as an independent service in the microservice architecture. Classic designs of testing an application follow the testing pyramid. The focus is on Unit tests which still have a leading role in testing microservices, followed by integration tests. Integration in monolith applications needs cooperation from all involved developers. In a microservice architecture only the developers of this service and from each service the tested service reach, by using API interfaces, needs to be concerned.

4.1 Testing Pyramid

The testing pyramid first invented by Mike Cohn is represented in figure 3 and uses the classic testing process: Unit, Service and User Interface testing. This structure might be outdated for microservices and must be improved. There are a few attempts for such improvements: adding new layers, renaming or shuffling [1]. In Cohn's article, he describes the layers and their meaning about the position and size. The bottom layer, Unit testing, describes the most efficient and impacting aspect of testing. These Unit tests can be written

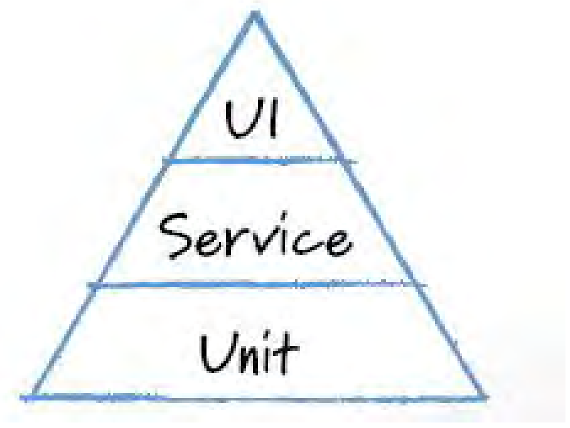


Figure 3: Testing pyramid by Mike Cohn [15]

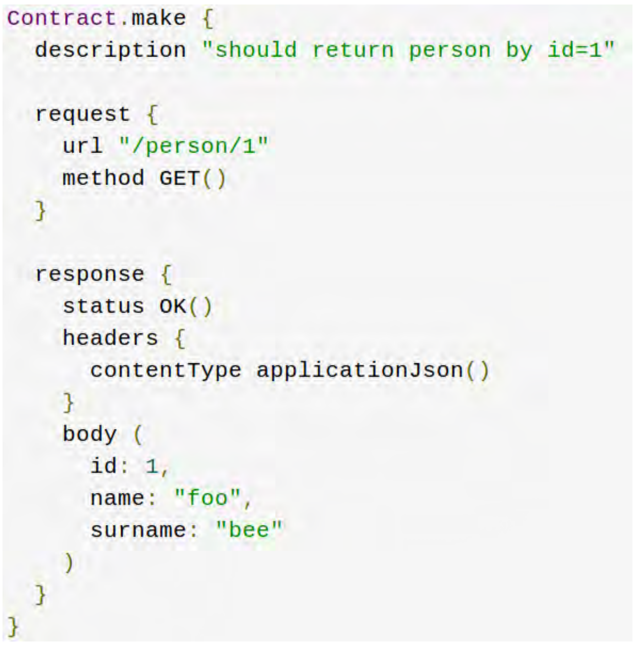
and executed fast. Cohn notes the impact is already given while developing the service as the results can be accurately compared to usual bug reports. The second layer, Service testing, refers to a feature of the code as a service. This layer represents the middle way between the low costs, in the form of time to write and execute, Unit testing and expensive UI testing. UI or User Interface testing is expensive in writing and executing time, as each time a test case runs not only the specific functionality is tested but also every code the functionality use. This leads to the fact that these UI tests need more time to write and execute compared to Unit or Service tests [15]. Compared to microservices the pyramid is still viable but needs to, at least, rename some of the layers. A modern but small version of the test pyramid would be Unit, Integration, End-to-End (E2E) testing, from bottom to top. Unit testing still applies to microservices because compared to a monolith application is each service the same, just smaller. E2E testing as equivalent to UI testing is related to the fact that microservices are not directly used by a user. Microservices describe the underlying infrastructure and so it is not directly a software solution. So the E2E layer acts the same as a user interface testing. Each time a test runs, full code structure generating the tested feature is checked. Checking the code multiple times lead to expensive results [16]. The middle part Integration just differs in its implementation for microservices compared to the Service layer. Service is defined by Cohn as the testing of the interaction from each service with each other. These interaction tests between modules inside an application can lead to lower-cost testing by reducing the writing time of test cases and the execution time [15]. In microservices interaction need to be tested inside each service as an own application but also between each service. So the Service/Integration layer can be seen as a doubled layer because the technique needs to be applied twice for microservices. The abstract concept of the testing pyramid does not show solutions for the testing, so the fundamental aspects of each layer need to be extracted.

4.2 Test types

There are several aspects of the software that can be inspected. At first, the software should be working correctly. Functionality is the most essential aspect of the software. If the functionality of the software is not given, the performance of it is irrelevant. If the software works correctly the resiliency of the system could be important. What if services go down? Does the whole system break down or do the other services continue to work? Another focus could be the performance of the system, the run time of single services or whole test scenarios could be analyzed. Every test type mentioned can be evaluated with the help of the test pyramid and used in each stage. These test types have different purposes but need to considered while designing a test for an application.

4.2.1 Functional Testing

Functional Tests will be found in every layer of the test pyramid. In the beginning, the functionality of the smallest units of the overall system will be tested. It is also possible that one microservice has several units that can be tested. For this purpose, unit tests are the most suitable [17]. But there are two ways to test a unit in software: black-box test and white-box test. Black-box tests are only considering the interface. The input parameters are often assigned to different equivalent classes, for example, positive numbers, negative numbers, zero and expected specific outputs. But the implementation of the unit is insignificant for the black-box test. One advantage of black-box tests is that there are well-suitable for a test-driven development (TDD) approach [18]. The test case can be written first and the code implementation afterward. With this approach, there will be a well-written test suite which is very important when the code base increased. On the other hand, white-box tests are not suitable for the TDD approach because to write a white-box test you need to know how the functionality is implemented. Nevertheless, a white-box test is way more expressive and reliable than a black-box test because they cover every path of a code base and black-box tests might miss important cases. Because the functionality of a unit is the most important aspect and it often has a manageable code base the white-box tests are recommended for this purpose. If all white-box tests for every unit of a microservice pass, the microservice itself can be inspected on his functionality. Here it depends on the size and complexity of the microservice whether to use black-box tests or white-box tests. If the microservice uses an HTTP interface and responds with JSON files it is helpful to use a Consumer-driven Contract (CDC) test [19]. By using CDC tests the consumer who wants to make use of service defines a contract. In this Contract, the syntax of the expected response is well-defined. The consumer defines his expectations on a specific request. An example of such a contract is represented in figure 4. The consumer sends an HTTP request with the parameter "/person/1" and expects a "Status OK", an id, a name, and a surname. The content of the response will not be tested here, only the schema will be matched. The advantage of a consumer written contract is that the consumer better knows the usage of the microservice than the microservice developer themselves. Additionally unused responses in the service can be eliminated. This contract then can be checked automatically for the microservice. This test strategy simultaneously covers parts of the integration tests because if the interface is well defined and checked, two microservices speaks on the same syntactic level. A more modern way to communicate between microservices is an event-driven architecture [21]. When a microservice wants to share some



```
Contract.make {
  description "should return person by id=1"

  request {
    url "/person/1"
    method GET()
  }

  response {
    status OK()
    headers {
      contentType applicationJson()
    }
    body (
      id: 1,
      name: "foo",
      surname: "bee"
    )
  }
}
```

Figure 4: Example of a Contract [20]

piece of work it produces an event that contains the output of its work. Any consumer service which is interested in the results of the provided service consumes this event. Unlike with REST, services that create requests do not need to know the details of the services consuming. One advantage of this approach is that the events can be stored in queues or buffers and the service would not be blocked by waiting on a response of a consumer. Furthermore scaling becomes easier because if new microservices are interested in existing task solvers, they only need to subscribe to them. A disadvantage of an event-driven microservice architecture is that it is challenging to provide the microservice functionality in public or public APIs that are required in your software. Also on an event-driven microservice architecture, it is possible to use CDCs with the difference, that now we have several consumers who will test the same event. If we go higher in the test pyramid we need to check the integration between several microservices. For this purpose mocks become important and helpful [22]. Mocks simulate the behavior of services. It is very time-consuming and unreliable to run integration tests on the runtime environment. If external APIs like the GitHub API will be used, a network connection will be needed, the latency can be time-consuming and there is no impact on whether the service is down or not. Furthermore, there is no longer control over the data that will be used by the external service. To avoid these problems mocks will be used. Mocks fake such services on the local environment. Thus there is no internet connection needed and a fast correct response can be guaranteed. To solve the problem that there is no control over the data that will be received mocks can respond with data that is expected. There is a way to combine CDCs with mocks. Mocks simulate the response which is specified in the contract and the consumer is decoupled from the provider and can be tested independently. Providers can also decouple from consumers for their testing. Mocks can be used to simulate consumer's

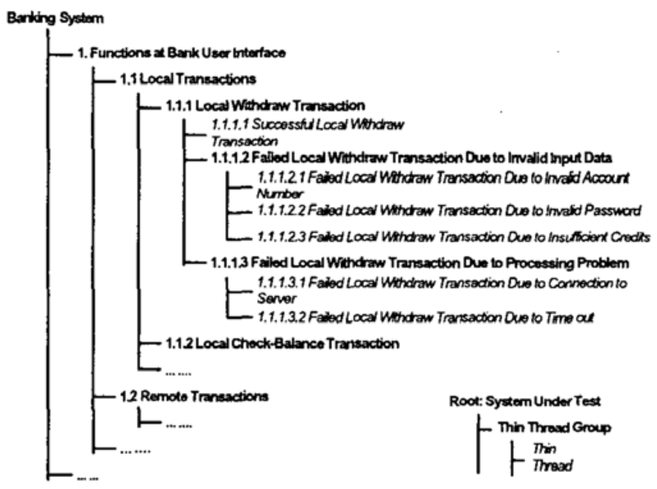

Banking System
1. Functions at Bank User Interface
1.1 Local Transactions
1.1.1 Local Withdraw Transaction
1.1.1.1 Successful Local Withdraw Transaction
1.1.1.2 Failed Local Withdraw Transaction Due to Invalid Input Data
1.1.1.2.1 Failed Local Withdraw Transaction Due to Invalid Account Number
1.1.1.2.2 Failed Local Withdraw Transaction Due to Invalid Password
1.1.1.2.3 Failed Local Withdraw Transaction Due to Insufficient Credits
1.1.1.3 Failed Local Withdraw Transaction Due to Processing Problem
1.1.1.3.1 Failed Local Withdraw Transaction Due to Connection to Server
1.1.1.3.2 Failed Local Withdraw Transaction Due to Time out
1.1.2 Local Check-Balance Transaction
1.2 Remote Transactions
Root: System Under Test
Thin Thread Group
Thin Thread

Figure 5: Thin Thread tree of a banking system [16]

requests which are defined. On the top layer, the E2E test will be attended. Tsai proposes an approach to design E2E integration testing, including test scenario specification in his paper [16]. His approach is to generate test cases systematically by using thin-threads. For example, the E2E functionality of a banking system can be tested by defining a thin thread tree, like in figure 5.

But testing is not limited to the execution of atomic functions. It is also possible to test a complex combination of atomic functions.

4.2.2 *Fault-Tolerance Testing*

Fault-tolerance tests check the resiliency of the overall system which can occur by failures such as node failures or network failures. To ensure that the system operating properly commonly occurring fault can be simulated by injecting them and evaluate the consequences. For example, if a shopping website has a rating system then it should be possible to buy items even if the rating service is down. The whole system should not completely break down if one service is not available. To prevent this scenario the individual services should simulate a downtime while the overall system will be monitored. Because software build in a microservice architecture involves many independent components that may not be reachable at any time, it is important that the overall system is not breaking down. This test type is settled in the integration- or the E2E-layer of the test pyramid, because only components or microservices will be deployed on different containers or servers. The following test method simulates downtimes of services and it is difficult to classify this method as a black or a white-box test. No input and output can be checked. Moreover the absence of services will be tested. Software often contains different types of components like load balancers (LB), network cores, service node or database nodes. These components have different types of failures such as high CPU, high memory utilization, full node failure, LB failure and network failure. They affect the behavior of overall system differently.

To realize an automated fault tolerance test approach, Adithya Nagarajan [23] declares five requirements :

- Ability to inject faults in a controlled manner
- Representation of system topology
- Identification of appropriate faults to inject on a selected machine
- Recovery from fault state
- Metrics representing the health of the overall system

He introduces a test tool that satisfies all the above requirements. It is called Screwdriver and is developed by the Groupon [3]company [24]. The infrastructure of Screwdriver is shown in figure 6.

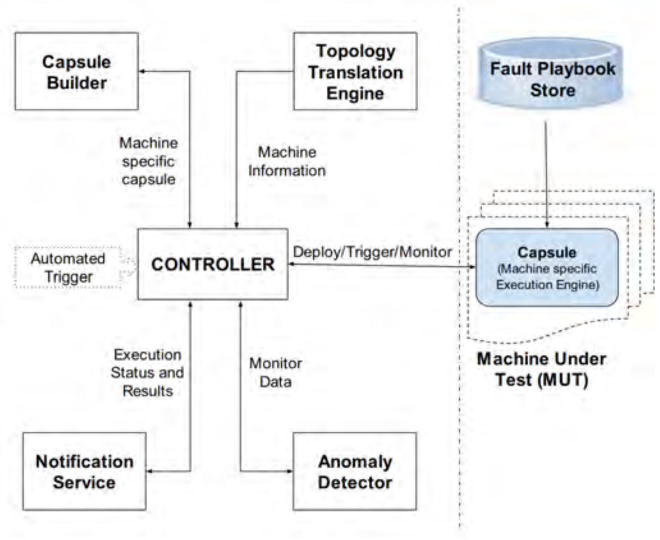


Figure 6: Infrastructure of the Screwdriver framework [23]

A Capsule Builder injects faults in a controlled manner. On every fault injection request, a Capsule is built to solve all the above requirements. It exposes a secure REST API through which we can control the fault, and stop it if necessary. It uses a Topology Translation Service which stores the system topologies in a SQL datastore to understand the service dependencies. In the so-called Playbook, the different types of appropriated faults will be stored. Using this Playbook service, one can define the set of scripts, and the commands to inject a fault, and also to abort the fault. Groupon builds a Metric Adapter to observing the behavior of the machine when injecting a fault using the Capsule. All machines are equipped with agents to monitor the host both on the system level as well as the application level. The monitors uses the metrics published by each host, and alerts on any outliers based on custom thresholds. It can be adapted with any given metrics system such as RRDtool, and Splunk [25, 26].Theses tools aim to handle time series data such as network bandwidth, temperatures or CPU load. The so-called Capsule component guarantees that a fault injection run can be killed and recovered at any time. Faults are configured as Java objects and are run as bash scripts thereby providing a layer of abstraction. Ming-Yee Laia and Steve Wang recommended a so-called Software Fault Injection Testing (SFIT) technique in their book ”Software Fault Tolerance” [27] which is similar to the Groupon companies approach. There is no need for waiting on occurrence of failures and testing happens in a systematic way [27]. In the pursuit of methodology for testing fault tolerance, Ming-Yee Laia and Steve Wanga also define several challenges:

- Complexity of telecommunications and software
- Dormancy of faults
- Diversity of telecommunications systems
- Constraint of resource availability

Because the book ”Software Fault Tolerance” was published in 1995 and ”Screwdriver” was published in 2016 they have different views on how a Fault Tolerance Testing Tool should be designed. The Complexity of telecommunication and software is comparable with the representation of the system topology, both agree that the systems must be understood to test the components and services. One aspect which Screwdriver my not be put attention on but Ming-Yee Laia and Steve Wang do, is the dormancy of faults. Some faults may only be triggered when a system is under extreme stress, abnormal use, or severe failures. Ming-Yee Laia and Steve Wang established that services have different telecommunication interfaces but today this is not a problem anymore because most developers offer their product on an API Interface like weather data or the Git API. On the other hand, Ming-Yee Laia and Steve Wang do not consider the Recovery Steps which the Groupon company did. Furthermore, they do not care about the metric representing the health of the overall system. As seen in the following both prefer a Fault Insertion method to test the underlying system. The testing Methodology from Ming-Yee Laia and Steve Wang is depicted in figure 7. The Proactive Software Architecture Analysis step, the Reactive Root Cause Analysis on trouble reports step and the test selection step are all Pre SFIT Steps which will be executed before the actual test will be running. Ming-Yee Laia and Steve Wang tried to analyze the software architecture in a proactive way similar to the Groupon company by analyzing the system topology. Additionally, the root cause faults infuse in the test cases. These could result from internal testing or external field problems like customer reports. They help to identify common problems. Both approaches have a library or a Playbook where they store different test cases and fault insertions. The test set needs to be selected before the actual SFIT testing. Regarding Ming-Yee Laia and Steve Wang, the test set should consider criteria like test cases for errors that have direct customer service impact. If the test cases were selected the tests will be planned by writing scripts that inject different faults in the system, for example, the break down of an extern API should be simulated. Then the actual SFIT step will be executed the fault insertion step involves the actual insertion of faults, while the test execution trigger will activate the inserted faults by input values from the user for

[3]Groupon is an American global e-commerce marketplace connecting subscribers with local merchants by offering activities, travel, goods and services in 15 countries

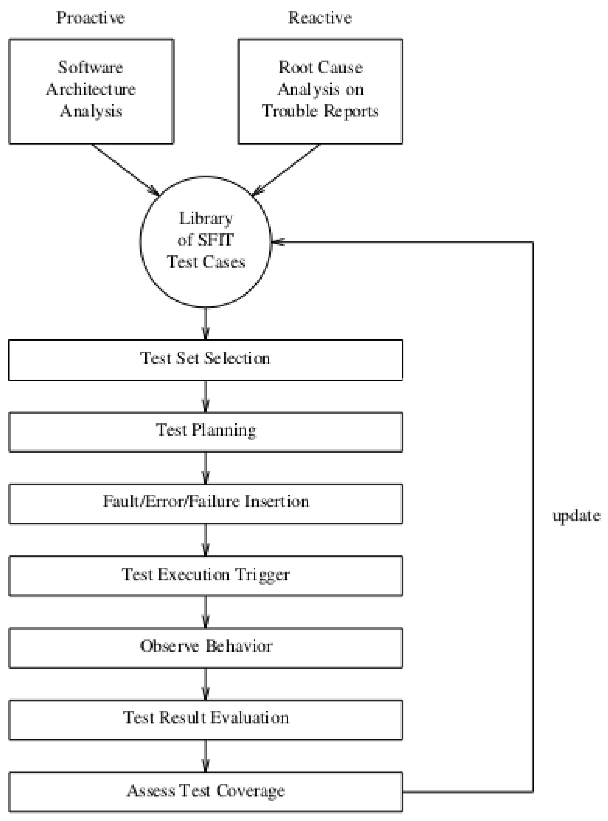


Figure 7: SFIT Methodology [27]

example. If the fault is inserted the overall system reactions need to be observed. Finally, the Post SFIT step will be executed and the test results which were recorded need to be analyzed and evaluated and after that the test case library can be updated if the test coverage were assessed. Summarized both approaches have three steps in common: Analysing the underlying system, creating test cases, inserting faults, observe the behavior of the system and evaluate the results. If the overall system has been passed the fault tolerance test the performance of the software can be inspected.

4.2.3 Performance Testing

Performance tests are located on the system level and belong to black-box tests according to Camargo [13]. Primarily the performance which the user experience on the end product is the most important because it could affect buy decisions for users. But to ensure good performance on the end product the single microservice has to be performant. Performance tests measure the throughput and the response time of selected services or the overall system. Performance-related tests are load tests, stress tests, and capacity tests. Because with a microservice approach there are single and well-defined tasks the measurement of the above aspects is much easier than in a monolithic system. On the following approach by Camargo, the microservices communicate over HTTP operations like GET and POST. To test metrics like throughput and response time of microservices a test runner is needed which sends a request to microservice and measure the criteria with tools like JMeter. Because the test should be executed in an automated way the test runner needs to know which request is possible to a microservice and what is the expected response. By increasing the number of microservices in a system, it is costly to write every possible request on a microservice. For this purpose, Camargo developed a framework that automated this process. The basic idea is that the test runner can send an OPTION request to a microservice and get a response with the test specification in a JSON schema. This test specification contains all possible request which can be sent to the microservice, for example, a POST command to save a new financial transaction in a banking system or GET request to receive the transaction by id. The framework for performance test (FPTS) build these test specification and delivers them to the test runner. Figure 8 shows the procedure of the way the test specification will be added to the FTPS.

Every time a service client runs an HTTP request (GET, POST, DELETE or PUT) the FTPS processes this request by saving the structure a forwarded them to the REST service which then gives a response to the service client. Further in his paper, he shows this framework does not noticeably affect the performance of the system itself. Sai Prakash describes an approach for performance tests on event-driven microservices [11]. Event-driven microservices communicate over messages. Microservices can subscribe to queues and receive all messages that will be put in this queue. The advantage of this approach is that several microservices can subscribe to the same queue which results in less communication overhead. The performance test described by Prakash works as follows. A tool like JMeter sends numerous messages to a so-called Notification Service and then calculate the metrics like the throughput or the response time.

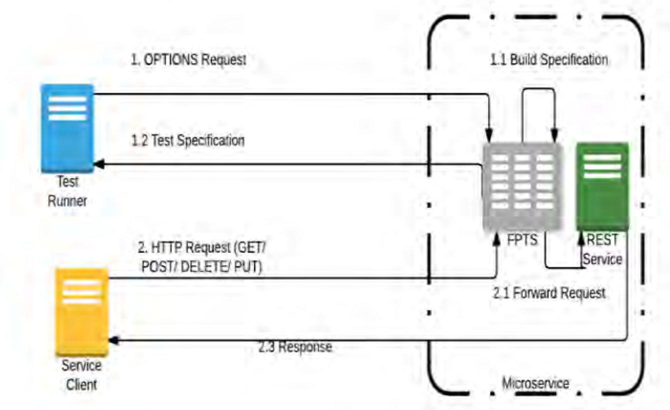


Figure 8: Framework Behavior for HTTP Requests [13]

5. DISCUSSION

Comparing different testing strategies for microservices and their benefits, but also we have to keep the downsides in mind. Tailing a system under development around testing can lead to a better quality of the service. First, we start with the functional tests there we have introduced two strategies: black-box test and white-box test. Both have their advantages and disadvantages it depends on the concept the team developed the software. If a TDD approach will be preferred the white-box test is not suitable. But if the path coverage must be on 100 % a white-box test should be used. If teams have enough time and want a very high quality of code both tests can be implemented. First, they write a black-box test for the specific service, then the codebase will be implemented and finally, a white-box test can be created. Next, we introduced CDCs. With these tests, it can be guaranteed that all service speaks the same language but it is not guaranteed that the system works correctly. For this purpose, Tsai has introduced an approach that tests several scenarios and cases by simulating systematically users behaviors. After that, we focus on fault tolerance testing here two approaches will be opposed. The one from Groupon and Lyu but there also resemble in a few aspects. For example, the Screwdriver tool developed by Groupon does not consider about root cause analysis or trouble reports. On the other hand, screwdriver attaches great importance to recovery from fault states and Lyu does not broach this subject. Both use fault insertion to systematically check different test cases. Finally, we approached for performance tests. There we had an approach published by Camargo and one published by Prakash. They use different communication bases. Camargo technique is based on a simple HTTP communication and Prakash uses event-driven microservices which internal is based on HTTP communication but not visible to the outside.

6. CONCLUSIONS

Automatic testing of an application that is based on a microservice architecture is different compared to a monolith application. The problems of testing a monolith application were not solved by using the microservice architecture, they shifted and also created new problems. But on the other hand, microservices simplify the building and maintenance of applications compared to monolith development. Parts of an application can now be independently developed, faster delivered and hardware optimized. There were new test methods developed for several test types. Unit tests on the lowest layer of the testing pyramid can be implemented easier than in a monolithic system because a microservice is a well-defined task. But the less the effort on the unit test layer the more a good integration between the services is needed. But the benefits of the microservice prevail which is recognizable by many large players like Google or Netflix. They already use microservices in their software. In this paper, we introduced functional tests, fault tolerance tests, and performance tests but more aspects can be inspected in the future like security checks or user interface tests.

7. REFERENCES

[1] T. Clemson, “Testing Strategies in a Microservice Architecture,” 2014. [Online]. Available: https://martinfowler.com/articles/microservice-testing/

[2] D. Savchenko, G. I. Radchenko, T. Hynninen, and O. Taipale, “MICROSERVICE TEST PROCESS : DESIGN AND IMPLEMENTATION,” 2018.

[3] Y. C. W.-T. L. S.-J. L. N.-L. H. Shang-Pin Ma, Chen-Yuan Fan, “Using service dependency graph to analyze and test microservices,” *IEEE Software*, 2018. [Online]. Available: https://ieeexplore.ieee.org/document/8377834

[4] R. T. Fielding, “Representational state transfer (rest),” 2000. [Online]. Available: https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

[5] C. Esposito, A. Castiglione, and K.-K. R. Choo, “Challenges in Delivering Software in the Cloud as Microservices,” *IEEE Cloud Computing*, vol. 3, no. 5, pp. 10–14, Sep. 2016.

[6] S. S. Divyanand Malavalli, “Scalable microservice based architecture for enabling dmtf profiles,” *IEEE Software*, no. 11, Jan. 2015. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7367395

[7] waltermoreira, “Serfnode,” May 2015. [Online]. Available: https://github.com/waltermoreira/serfnode

[8] R. D. Joe Stubbs, Walter Moreira, “Distributed systems of microservices using docker and serfnode,” *IEEE Software*, no. 7, 2015. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7217926/

[9] D. T. Björn Butzin, Frank Golatowski, “Microservices approach for the internet of things,” *IEEE Software*, no. 21, Nov. 2016. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7733707

[10] P. Jamshidi, C. Pahl, N. C. Mendonça, J. Lewis, and S. Tilkov, “Microservices: The Journey So Far and Challenges Ahead,” *IEEE Software*, vol. 35, no. 3, pp. 24–35, May 2018.

[11] S. Prakash, “Performance Testing of Event-Driven Microservices,” Dec. 2018. [Online]. Available: https://medium.com/capital-one-tech/performance-testing-of-event-driven-microservices-f5de74305985

[12] D. D. Ervin Djogic, Samir Ribic, “Monolithic to microservices redesign of event driven integration platform,” *IEEE Software*, May 2018. [Online]. Available:

https://ieeexplore.ieee.org/document/8400254

[13] A. Camargo, I. Salvadori, R. Mello, and F. Siqueira, "An architecture to automate performance tests on microservices," Nov. 2016, pp. 422–429.

[14] L. Chen, "Microservices: Architecting for Continuous Delivery and DevOps," in *2018 IEEE International Conference on Software Architecture (ICSA)*, Apr. 2018, pp. 39–397.

[15] M. Cohn, "The forgotten layer of the test automation pyramid," 2009. [Online]. Available: https://www.mountaingoatsoftware.com/blog/the-forgotten-layer-of-the-test-automation-pyramid

[16] W. Tsai, X. Bai, R. Paul, W. Shao, and V. Agarwal, "End-to-end integration testing design," in *25th Annual International Computer Software and Applications Conference. COMPSAC 2001*, Oct. 2001, pp. 166–171, iSSN: 0730-3157.

[17] D Language Foundation, "Unit Tests." [Online]. Available: https://dlang.org/spec/unittest.html

[18] S. saini, "How TDD Changed the Way I Approach Software Development," Oct. 2019. [Online]. Available: https://medium.com/better-programming/how-tdd-changed-the-way-i-approach-software-development-38509263f9ec

[19] T. Hombergs, "7 Reasons to Choose Consumer-Driven Contract Tests Over End-to-End Tests," Nov. 2017. [Online]. Available: https://reflectoring.io/7-reasons-for-consumer-driven-contracts/

[20] P. Software, "Getting Started · Consumer Driven Contracts." [Online]. Available: https://spring.io/guides/gs/contract-rest/

[21] mostlyjason, "Best Practices for Event-Driven Microservice Architecture," Sep. 2019. [Online]. Available: https://hackernoon.com/best-practices-for-event-driven-microservice-architecture-e034p21lk

[22] J. June, "How to test software, part I: mocking, stubbing, and contract testing," Apr. 2019. [Online]. Available: https://circleci.com/blog/how-to-test-software-part-i-mocking-stubbing-and-contract-testing/

[23] A. Nagarajan and A. Vaddadi, "Automated Fault-Tolerance Testing," in *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, Apr. 2016, pp. 275–276.

[24] Groupon and kavin, "Screwdriver: Improving Platform Resiliency at Groupon." [Online]. Available: https://engineering.groupon.com/2016/java/screwdriver/

[25] Tobias Oetiker, "About RRDtool," Feb. 2017. [Online]. Available: https://oss.oetiker.ch/rrdtool/

[26] "About Splunk Enterprise." [Online]. Available: https://docs.splunk.com/Documentation/Splunk/8.0.1/Overview/AboutSplunkEnterprise

[27] S. Wang and M.-Y. Lai, *Software Fault Tolerance*, 1st ed., M. R. Lyu, Ed. Chichester ; New York: Wiley, Apr. 1995.

Determining Metric Thresholds for Code Smell Detection: A Systematic Mapping Study

Lukas Stief
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
lukas.stief@rwth-aachen.de

Tim Jentzsch
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
tim.jentzsch@rwth-aachen.de

ABSTRACT

Code smells in software systems indicate possible issues in the software architecture or its design. They are widely considered an important indicator for design flaws in source code. They signal that the software quality and maintainability should be improved through code refactoring. Metrics, such as lines of code of a method, are being used to find code smells in software projects. The definition of thresholds for these metrics is a difficult process, as its determination may result in undetected or falsely detected code smells.

This paper provides an overview of current techniques to define metric thresholds for code smell detection. Approaches are found and classified by conducting a *systematic mapping study.* This is a secondary study investigating all papers in a given field to identify research gaps. Most of the articles are using techniques from the following fields: *Rule Based Detection*, *Probalistic Systems* and *Machine Learning.* Mainly the categories of *Design*, *Implementation* and *Architecture* code smells have been investigated. We could not identify any studies determining thresholds for *Energy* and *Test* smells. *Java* and its object-oriented nature is being referenced the most in the papers.

Keywords

software engineering, code smell, metric thresholds, systematic mapping study

1. INTRODUCTION

During the development process, bad code structures are introduced into software [15]. While they do not add functional errors, they can hinder maintenance, readability and comprehensibility of the code base [10]. *Code smells* are indicators of these issues. They are an important indicator that the software quality should be improved to prevent possible errors [15]. Different metrics, such as the number of code lines, can be used to find code smells. For example, a class with a very large number of code lines could indicate the need to separate the functionality into multiple classes. To avoid false positive or negative results, it is essential to choose effective thresholds, which determine if a smell indicates a valid threat to the software quality [16]. So far, many different techniques have been used to derive the thresholds, but it is difficult to choose an adequate one for a given project [1].

To simplify this selection process and to reveal potential research gaps in this area, this paper presents a *systematic mapping study* to classify the different techniques. This is a type of secondary study that aims to provide a comprehensive overview over a given research topic. It primarily analyses the metadata, title and abstract of the primary studies to provide a classification scheme of the research area. A *systematic literature review* (SLR) is another type of secondary study which analyses every paper in-depth to provide answers to more detailed research questions. It could be conducted after a systematic mapping study.

The remaining paper is structured as follows: Section 2 gives an overview of systematic studies and reviews that are targeting similar topics. Section 3 describes the research questions used for this systematic mapping study. In section 4 the used methods to execute this study are explained, resulting in the extracted date presented in section 5. The data's interpretation has been done in section 6. The threats to validity are discussed in section 7 and in section 8 a discussion of possible research gaps and the studies conclusion is presented.

2. RELATED WORK

To the best of our knowledge, two mapping studies have been presented in this research area. Bandi et al. [2] conducted a systematic mapping study, but focused on code decay, a gradual process which negatively affects the software quality. Kitchenham [8] proposed a preliminary mapping study about trends in software metric studies. SCOPUS[1] has been searched, resulting in 91 relevant papers. Techniques to derive the code smell thresholds have not been investigated.

Additionally, we identified five SLRs. Rasool et al. [13] performed a SLR on the detection of code smells in 2015. They executed the search on Google Scholar[2] and identified 46 relevant studies. Castro Lima et al. [4] performed a SLR in order to obtain reference values and thresholds for software metrics. They investigated 19 studies out of

[1]SCOPUS (scientific database): `https://www.scopus.com/`
[2]Google Scholar (scientific search engine): `https://scholar.google.com/`

five databases. Fernandes et al. [5] also conducted a SLR and selected 107 studies out of six electronic data sources. Gupta et al. [7] presented another SLR which focused on code smells in *Java* and reviewed 60 papers. Additionally, Azeem et al. [1] proposed a SLR specifically to analyze machine learning techniques to detect code smells.

Furthermore, Vale et al. [16] presented an *ad-hoc* literature review on 50 selected articles to identify methods to determine thresholds. However, the study focused only on the available methods and did not investigate the demographics. Moreover, it did not aim for completeness and only searched four electronic databases.

In summary, no systematic mapping study with the focus on code smell metric thresholds has been performed yet.

3. RESEARCH QUESTIONS

The goal of this research is to give an overview over the techniques currently used to derive code smell thresholds and to classify the current solutions. Research questions (RQs) were defined to steer towards this objective. They were used to guide the definition of the search string, the selection process and the data extraction. The presented mapping tries to answer the RQs.

RQ1 : *What are the demographics of the studies, their authors and their publishers?*
Rationale: We are interested in the demographics of the studies backgrounds: Their publication years, their source and their authors' professional background. This way, we can determine a trend of the interest in this research topic.

RQ2 : *Which techniques can be used to determine code smell metric thresholds?*
Rationale: Since there are different kinds of code smells and their corresponding metrics, different techniques can be used to determine thresholds for them. We should give an overview over the available techniques.

RQ3 : *Which domains have been targeted by the given approaches?*
Rationale: Several approaches are available to define code smell thresholds, but not all of them be suitable for every business domain. Domains with a lot or very little support should be highlighted in the study as an indication for practitioners. We are also interested in the programming languages referenced in the articles.

RQ4 : *Which software tools are mentioned in the studies?*
Rationale: Multiple tools are available to detect code smells in software systems. The tool used or mentioned in each study are extracted in order to give an overview available tools for certain derivation techniques.

4. MAPPING STUDY METHODS

We followed the guidelines provided by Kitchenham et al. [9] and Peterson et al. [12] to perform the systematic mapping study. Additionally, we used the study by Li et al. [11] about technical depth as a reference.

4.1 Study Search

4.1.1 Search Scope

The search scope is an important aspect of a systematic mapping study, as it directly influences the completeness of the search [11]. In order to perform an exhaustive search and to find all relevant primary studies, it is necessary to search in many different electronic databases [3, 9, 12].

Table 1: Selection of electronic databases

#	Database	Selected
DB1	ACM Digital Library	Yes
DB2	CiteSeerX	Yes
DB3	dblp	Yes
DB4	IEEE Xplore	Yes
DB5	IET Digital Library	Yes
DB6	ScienceDirect	Yes
DB7	Scopus	Yes
DB8	Springer Link	Yes
DB9	Web of Science	Yes
DB10	Wiley Online Library	Yes
DB11	Ei Compendex	No
DB12	Google Scholar	No

We conducted the search in ten electronic databases, which are depicted in table 1. The investigated databases have been proposed or used by Brereton et al. [3], Kitchenham et al. [9] or Li et al. [11]. Additionally we chose to include dblp[3] to increase the completeness of the study. We excluded Ei Compendex, as the trial searches did not yield any relevant results. Google Scholar was excluded as its contents are mostly covered by the other databases. Furthermore, the configuration of the search query is very limited, leading to many irrelevant results.

The start of the search period is 2002, as the term 'code smell' has been coined at that time [6]. The end is November 24 2019, the time of the search.

4.1.2 Search Strategy

The goal of the search strategy is to gather all relevant papers in the databases to increase the completeness of the study. At the same time we also want to minimize the amount of irrelevant results. Because this is difficult to achieve with the database search alone, we additionally performed *snowballing* as described in section 4.3. We used the following strategy:

1. Trial searches with multiple search strings were performed on each databases in table 1. During this process we determined effective search strings for the topic.

2. The search string `(software OR code) AND (smell OR antipattern) AND threshold` has been selected for the formal search.

3. DB1–DB10 were formally searched with the defined search string.

4.2 Study Selection

During the study selection, the results of the electronic data bases are filtered manually by the researchers. This process determines the relevant papers for the data extraction.

[3]dblp (computer science bibliography): `https://dblp.uni-trier.de/`

4.2.1 *Selection Criteria*

We defined different criteria for inclusion and exclusion of the studies. We included papers that meet the following criteria:

I1 The paper is a primary study. Some of the secondary studies were used during the *snowballing* process, described in section 4.3.

I2 The paper is related to software engineering.

I3 The paper is focused on code smell detection metrics and their thresholds.

We excluded papers according to the following criteria:

E1 The paper is not written in English.

4.2.2 *Selection process*

The study's selections process consists of the of the following steps:

1. Study selection by metadata (title and keywords). In this step, mainly papers about other research areas (e.g. biology and geology) have been excluded.

2. Study selection by abstract. This step identified most papers which did not focus on the identification of code smells in software projects.

3. Study selection by conclusion/full text. Here, we could exclude studies which did not present techniques to derive metric thresholds for code smells.

Each step has been conducted by two researchers independently. If they disagreed about the inclusion/exclusion of a study it has been included for the next step. In the last step, if no agreement could be found, the seminar supervisor reviewed the paper.

4.3 Snowballing

To increase the scope of the search, *snowballing* was used. This is a process where the references of the selected papers are investigated and filtered according to the selection criteria [17]. For example, papers which do not use the term 'code smell', but are still relevant to the research question can be identified by this technique. We only performed a single iteration of snowballing.

Additionally, the studies selected by similar literature reviews [16, 5, 4] have been investigated to improve the completeness of the search.

4.4 Data Extraction

For answering the defined Research Questions the data listed in table 2 has been extracted. The drawn data was recorded on a spreadsheet. The definition of the extraction has been carried out after reviewing 20 studies.

4.5 Data Synthesis

The extraction was done with the help of spreadsheet tools and descriptive statistics. The data items D1-D8 (table 2) have been pulled from each study and recorded in a spreadsheet. The data has been visualized using plotting tools, these visualizations are presented in section 5.

D4 was derived by extracting the approach used to determine metric thresholds, they have been categorized into their fields description. D5 was extracted by categorizing the papers targeted domain, this was based on the targeted programming languages paradigm or the targeted development environments. When specific programming languages were targeted solely they were extracted. D6 has been classified by collecting information about programming languages targeted by the study as well as programming languages used for showcases and examples. To determine D7 the tools which were used for showcased or examples or for which the proposed approach was implemented for were gathered. The data extraction for D8 has been carried out by tracking the targeted code smells of each study and then mapping them to the categories proposed by Suryanarayana et al. [14].

5. STUDY RESULTS

In the following section the results of the study will be presented. It is structured as follows: Section 5.1.1 depicts the demographic results, section 5.1.2 shows the approaches used to derive the thresholds, section 5.1.3 presents the targeted domains and the targeted code smells are listed in section 5.1.5.

5.1 Search Results

The number of papers after each step of the selection process is depicted in figure 1. The search in the electronic databases returned a total of 429 results. The selection by metadata reduced the total number of papers to 215. Afterwards, 73 duplicates have been identified, resulting in 142 studies. With the full text selection, 54 relevant papers have been selected. The *snowballing* process added 19 papers. In total, 73 studies have been selected for the data extraction. The final selection is listed in appendix A.

5.1.1 *Demographic Results*

The demographic results aim to answer RQ1 by presenting the statistics obtained by the extraction of D1, D2 and D3.

Classification by Publication Date.

The distribution of studies by publication year is shown in figure 2. Starting at 2009, the interest in this research field has increased significantly.

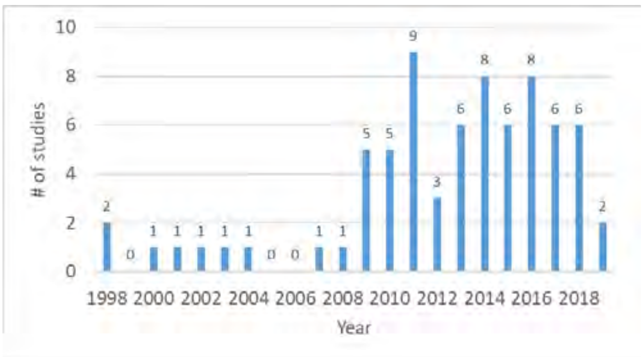


Figure 2: Amount of publications of the selected studies per year.

Classification by Author Type.

Figure 3 shows the number of studies classified by the origin of the authors. Most of them work in academics.

Table 2: Data extracted from each study.

#	Extraction data	Description	Targeted RQ
D1	Year	The year the study has been published in.	RQ1
D2	Author Type	Wheather the study has been published by academia, industry or both.	RQ1
D3	Publication Type	Wheather the study has been published on a conference, in a journal, workshop, book or thesis.	RQ1
D4	Presented technique	To give an overview of available techniques to determine code smell metric thresholds the presented derivation techniques of the studies are presented.	RQ2
D5	Targeted domain	The targeted software domains to give an overview of the approached domains.	RQ3
D6	Programming language	To determine the representation of certain programming languages, the references of studies on them are tracked.	RQ4
D7	Mentioned tools	Available software solutions to determine the mentioned metric thresholds are identified.	RQ4
D8	Targeted code smells	For the detection of research gaps, the studies targeted code smells are tracked.	None

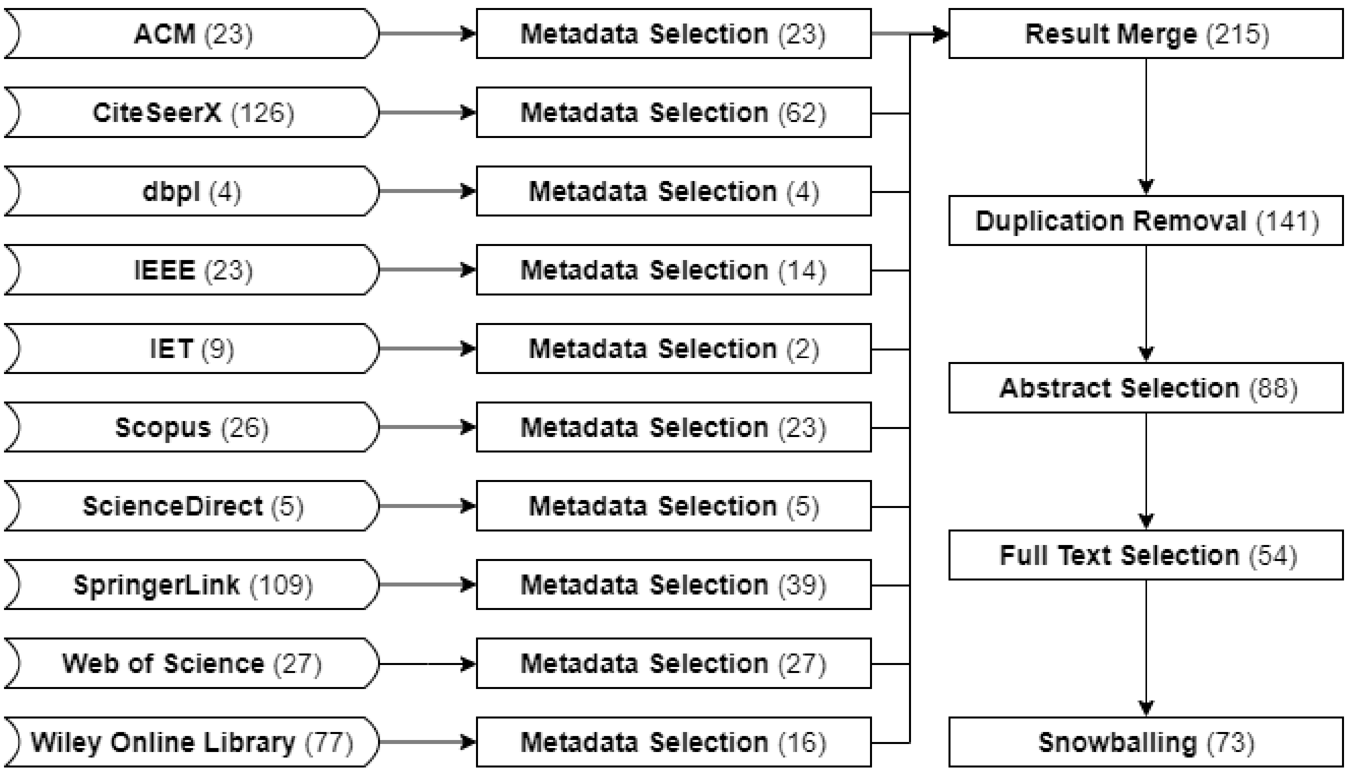


Figure 1: The number of papers after each selection step of the study.

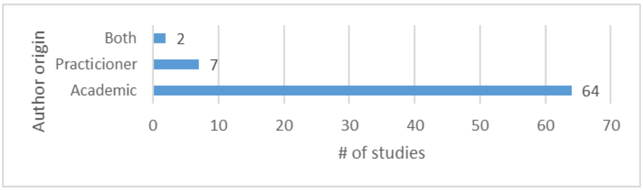


Figure 3: Number of selected studies by origin of the author.

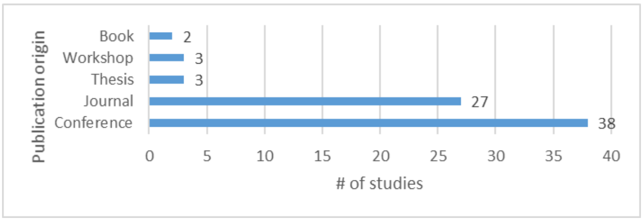


Figure 4: Number of selected studies by origin of the paper.

Classification by Publication-Type.
The distribution of the origins of the selected studies is shown in figure 4. Most of the papers have been released in either a conference or a journal. Additionally, a few thesis, books and workshops about this topic have been published.

5.1.2 Approaches Used to Determine Metric Thresholds

The main approaches used to derive the metric thresholds are *Probabilistic*, *Rule Based*, *Machine Learning* and *Iterative* approaches. *Probabilistic* techniques use statistical or probabilistic models such as Bayesian Belief Networks and regression algorithms. *Rule Based* approaches use logic based models like majority functions and boolean expressions. *Machine Learning* systems use models such as Artificial Neural Networks or Support Vector Machines. *Iterative* techniques use iterative algorithms, e.g. search based and greedy algorithms.

These results are depicted in figure 5 and answer RQ2.

5.1.3 Targeted domains

According to RQ3, the extracted studies are mapped to their approached domain. Almost all code smell research has been focused on object oriented programming (OOP), as shown in table 3.

This trend is also observable when comparing the referenced programming languages. Figure 6 shows that *Java* has been used the most to evaluate the approaches.

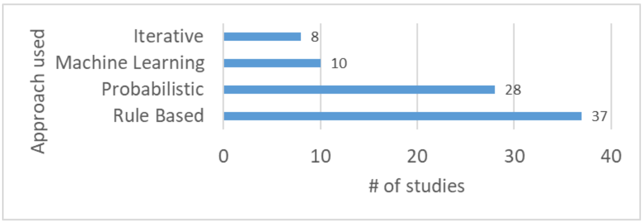


Figure 5: Approaches used to derive the metric thresholds.

Table 3: The domains targeted by the given approaches.

Targeted domain	# of studies
Object Oriented Programming	53
General Software Engineering	11
Web Applications	7
Mobile Applications	2
C Programming	2

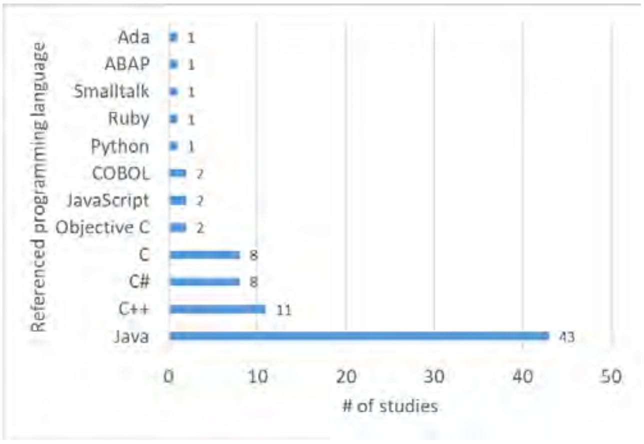


Figure 6: Referenced programming languages.

5.1.4 Referenced Tools

In order to answer RQ4, the tools referenced by the studies are shown in table 4, grouped by the targeted code smell category. The extracted tools either have the targeted metric derivation technique implemented or were used for showcasing examples.

5.1.5 Targeted Code Smells

A large number of code smells have been investigated to derive the appropriate thresholds. The code smells have been clustered in the following categories: *Design*, *Implementation*, *Architecture*, *Performance*, *Energy* and *Test* [14]. Most of the studies are dealing with design smells, while none of the studies approaches energy or test smells. Figure 7 shows the number of selected studies per defined category.

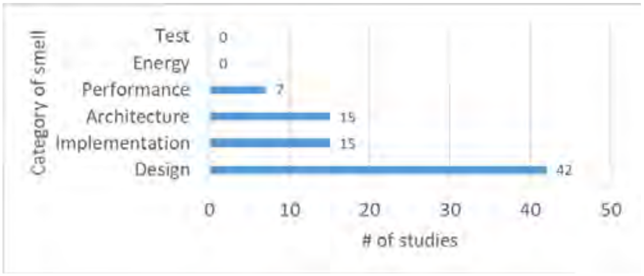


Figure 7: The code smell categories targeted by the threshold derivation.

Table 4: Referenced tools depending on code smell category.

Smell category	Mentioned tools
Design	Analyst4J [S48], CodePro Analytix [S19], Columbus Tool [S15], FindSmells [S65], FxCop [S48], InFusion [S43, S45], iPlasma [S29, S45], JDeodorant [S45, S60, S65], JSmell [S60], PMD [S29, S32, S34, S43, S45, S48], Pysmell [S72], Sonar Qube [S19], WekaNose [S12, S30], Xerces [S20, S29]
Implementation	Pysmell [S72], Sonar Qube [S19], WekaNose [S73]
Architecture	PMD [S34], McCabe IQ [S1], WekaNose [S30]
Performance	McCabe IQ [S1]

6. DISCUSSION

There is a steady interest in the derivation of code smell metrics. Starting from 2009, approximately 6 studies have been published per year in this research area. Only two papers have been released in 2019 so far, but it can be assumed that several more have been written, but not published yet.

The research field is dominated by authors from academic origin. Only about 10% of the selected studies have been (partially) performed by practitioners.

Most of the articles have been released on a conference or in a journal. However, a few have been published as a thesis, book or workshop.

The majority of approaches uses a rule based concept to determine the metric thresholds. Probabilistic techniques have also been thoroughly researched. Not as many studies investigated machine learning as a possible solution to the problem. This is still a growing area of research and investigated in more detail in the SLR by Azeem et al. [1]. Only 8 papers evaluated the potential of iterative processes.

The applicable business-domains of the derivation techniques reveal a big potential for future research. Kitchenham [8] presented that in the area of software metrics, about a third of papers are related to object oriented programming (OOP). However, for the derivation of thresholds, we identified that approximately 70% of the approaches are connected to OOP. This trend is also visible in the selection of programming languages. *Java* was used or referenced in over half of the studies. This observation is also supported by the SLR of Rasool et al. [13]. They show that 92–95% of code smell detection tools are based on *Java*. In contrast to the rising popularity of Python[4], only a few papers used it or other interpreted languages for their research.

The majority of articles investigated design related code smells. Here, we find a research gap for energy and test smells, as categorized by Suryanarayana et al. [14]. None of the selected papers explored their possible thresholds.

7. THREATS TO VALIDITY

The determination of threats to internal and external validity follows the definition by Wohlin et al. [18]. A possible bias of the researchers should be considered during data selection, data extraction and data evaluation.

7.1 Threats to Internal Validity

To avoid *threats to internal validity* [18], all steps of the study filtering and data extraction have been executed by two researchers independently and were discussed by both until an agreement was found. For research steps where no agreement was found the research's supervisor was consulted. This approach aims to minimize the influence of subjective interpretation of each researcher.

[4]As of December 2019, the TIOBE Index (`https://tiobe.com/tiobe-index/`) considers to make Python the programming language of the year for the second time. It has identified a growing interest in Python.

7.2 Threats to External Validity

Conditions which hinder the ability to generalize the results of a study are called *threats to external validity* [18]. One of those threats is the selection of the search string. We used *snowballing* to increase the scope of the search and to remove a potential bias of the employed search string. The large number of studies identified by the snowballing process (19) indicate that the search string has been too specific. Most of the added papers did not use the terms 'code smell' or 'antipattern' and have therefore been excluded by the initial search.

8. CONCLUSION AND FUTURE WORK

To present an overview over the research on deriving metric thresholds for code smells, we performed a *systematic mapping study*. We conducted an automated search on 10 electronic databases, resulting in 429 papers. They were filtered according to our criteria and snowballing was performed, leaving 73 relevant studies to be mapped. We extracted data according to our research questions and presented it visually.

This mapping study should be used as a starting point for both primary and secondary studies. While code smells in object oriented programming languages such as *Java* have been well researched, we identified gaps for other systems like interpreted languages. Additionally, we suggest to explore thresholds for code smells related to *energy* and *testing* (as defined by Suryanarayana [14]), as we have not found any research on these subjects.

9. REFERENCES

[1] M. I. Azeem, F. Palomba, L. Shi, and Q. Wang. Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. *Inf. Softw. Technol.*, 108:115–138, 2019.

[2] A. Bandi, B. J. Williams, and E. B. Allen. Empirical evidence of code decay: A systematic mapping study. pages 341–350, 2013.

[3] P. Brereton, B. A. Kitchenham, D. Budgen, M. Turner, and M. Khalil. Lessons from applying the systematic literature review process within the software engineering domain. *Journal of Systems and Software*, 80(4):571–583, 2007.

[4] E. de Castro Lima, de Resende, Antônio Maria P, and T. C. Lethbridge. The Uncomfortable Discrepancies of Software Metric Thresholds and Reference Values in Literature. *ICSEA 2016*, page 14, 2016.

[5] E. Fernandes, J. Oliveira, G. Vale, T. Paiva, and E. Figueiredo. A review-based comparative study of bad smell detection tools. In S. Beecham, editor, *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, pages 1–12. ACM, 2016.

[6] M. Fowler. CodeSmell. `https://martinfowler.com/bliki/CodeSmell.html`, 2006. Accessed: 2019-11-16.

[7] A. Gupta, B. Suri, and S. Misra, editors. *A Systematic Literature Review: Code Bad Smells in Java Source Code*, volume 10408, 2017.

[8] B. Kitchenham. What's up with software metrics? – A preliminary mapping study. *Journal of Systems and Software*, 83(1):37–51, 2010.

[9] B. Kitchenham and S. Charters. Guidelines for performing Systematic Literature Reviews in Software Engineering. 2007.

[10] P. Kruchten, R. L. Nord, and I. Ozkaya. Technical Debt: From Metaphor to Theory and Practice. *IEEE Software*, 29(6):18–21, 2012.

[11] Z. Li, P. Avgeriou, and P. Liang. A systematic mapping study on technical debt and its management. *Journal of Systems and Software*, 101:193–220, 2015.

[12] K. Petersen, R. Feldt, S. Mujtaba, and M. Mattsson. Systematic Mapping Studies in Software Engineering. 2008.

[13] G. Rasool and Z. Arshad. A review of code smell mining techniques. *Journal of Software: Evolution and Process*, 27(11):867–895, 2015.

[14] G. Suryanarayana, G. Samarthyam, and T. Sharma. *Refactoring for software design smells: Managing technical debt.* Elsevier Morgan Kaufmann Morgan Kaufmann is an imprint of Elsevier, Amsterdam and Boston, 2015.

[15] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. de Lucia, and D. Poshyvanyk. When and Why Your Code Starts to Smell Bad. In *BIGDSE 2015*, pages 403–414, Los Alamitos, California and Washington and Tokyo, 2015. Conference Publishing Services, IEEE Computer Society.

[16] G. A. D. Vale and E. M. L. Figueiredo. A Method to Derive Metric Thresholds for Software Product Lines. pages 110–119. Institute of Electrical and Electronics Engineers Inc, 2015.

[17] C. Wohlin. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the 18th international conference on evaluation and assessment in software engineering*, pages 1–10, 2014.

[18] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in software engineering.* Springer Science & Business Media, 2012.

APPENDIX

A. SELECTED STUDIES

[S1] O. Alan and P. Cagatay Catal. An outlier detection algorithm based on object-oriented metrics thresholds. In *2009 24th International Symposium on Computer and Information Sciences (ISCIS 2009)*, pages 567–570, Piscataway N.J., 2009. IEEE.

[S2] T. L. Alves, J. P. Correia, and J. Visser. Benchmark-Based Aggregation of Metrics to Ratings. In *Joint conference of the 21st Int'l Workshop on Software Measurement, 2011 and 6th Int'l Conference on Software Process and Product Measurement (IWSM-MENSURA)*, pages 20–29, Piscataway, NJ, 2011. IEEE.

[S3] T. L. Alves, C. Ypma, and J. Visser. Deriving metric thresholds from benchmark data. In *IEEE International Conference on Software Maintenance (ICSM), 2010*, pages 1–10. IEEE, Piscataway, NJ, 2010.

[S4] B. Amal, M. Kessentini, S. Bechikh, J. Dea, and L. B. Said. On the Use of Machine Learning and Search-Based Software Engineering for Ill-Defined Fitness Function: A Case Study on Software Refactoring. In C. Le Goues and S. Yoo, editors, *Search-based software engineering*, volume 8636 of *LNCS sublibrary. SL 2, Programming and software engineering*, pages 31–45. Springer, Cham, 2014.

[S5] M. T. Aras and Y. E. Selcuk. Metric and rule based automated detection of antipatterns in object-oriented software systems. In I. C. o. C. S. Technology and Information, editors, *2016 7th International Conference on Computer Science and Information Technology (CSIT)*, Piscataway, NJ, 2016. IEEE.

[S6] D. Arcelli, V. Cortellessa, and D. Di Ruscio. Applying Model Differences to Automate Performance-Driven Refactoring of Software Models. In M. S. Balsamo, W. J. Knottenbelt, and A. Marin, editors, *Computer performance engineering*, volume 8168 of *LNCS sublibrary: SL 2 - Programming and software engineering*, pages 312–324. Springer, Heidelberg, 2013.

[S7] D. Arcelli, V. Cortellessa, and C. Trubiani. Experimenting the influence of numerical thresholds on model-based detection and refactoring of performance antipatterns. *Electronic Communications of the EASST*, 59:1–30, 2013.

[S8] D. Arcelli, V. Cortellessa, and C. Trubiani. Performance-Based Software Model Refactoring in Fuzzy Contexts. In A. Egyed and I. Schaefer, editors, *Fundamental approaches to software engineering*, volume 9033 of *LNCS sublibrary. SL1, Theoretical computer science and general issues*, pages 149–164, Heidelberg, 2015. Springer.

[S9] F. Arcelli Fontana, V. Ferme, M. Zanoni, and A. Yamashita. Automatic Metric Thresholds Derivation for Code Smell Detection. In *Proceedings of the 2015 IEEE/ACM 6th International Workshop on Emerging Trends in Software Metrics*, WETSoM '15, pages 44–53, Washington, DC, USA, 2015. IEEE Computer Society.

[S10] T. arsons and J. Murphy. Detecting Performance Antipatterns in Com-ponent Based Enterprise Systems. In *In Middleware Doctoral Symposium*, volume 7, page 55, 2008.

[S11] K. Asano, S. Hayashi, and M. Saeki. Detecting Bad Smells of Refinement in Goal-Oriented Requirements Analysis. In S. de Cesare and U. Frank, editors, *Advances in conceptual modeling*, volume 10651 of *LNCS sublibrary. SL 3, Information systems and applications, incl. Internet/Web, and HCI*, pages 122–132. Springer, Cham, Switzerland, 2017.

[S12] U. Azadi, F. A. Fontana, and M. Zanoni. Poster: Machine learning based code smell detection through WekaNose. pages 288–289. IEEE Computer Society, 2018.

[S13] R. Baggen, J. P. Correia, K. Schill, and J. Visser. Standardized code quality benchmarking for improving software maintainability. *Software Quality Journal*, 20(2):287–307, 2012.

[S14] T. Bakota. Tracking the Evolution of Code Clones. In I. Černá, editor, *SOFSEM 2011*, volume 6543 of *LNCS sublibrary. SL 1, Theoretical computer science and general issues*, pages 86–98. Springer, Heidelberg, 2011.

[S15] D. Bán and R. Ferenc. Recognizing Antipatterns and Analyzing Their Effects on Software Maintainability. In B. Murgante, editor, *Computational science and its applications - ICCSA 2014*, volume 8583 of *LNCS sublibrary. SL 1 - Theoretical computer science and general issues*, pages 337–352. Springer, Heidelberg, 2014.

[S16] H. Barkmann, R. Lincke, and W. Löwe. Quantitative Evaluation of Software Quality Metrics in Open-Source Projects. In *2009 International Conference on Advanced Information Networking and Applications Workshop (WAINA 2011*, pages 1067–1072, Piscatraway NJ, 2009. IEEE.

[S17] S. Benlarbi, K. El Emam, N. Goel, and S. Rai. Thresholds for object-oriented measures. In *11th International symposium on software reliability engineering*, pages 24–38. 2000.

[S18] T. Beranič, V. Podgorelec, and M. Heričko. Towards a Reliable Identification of Deficient Code with a Combination of Software Metrics. *Applied Sciences (Switzerland)*, 8(10):1902, 2018.

[S19] M. A. Bigonha, K. Ferreira, P. Souza, B. Sousa, M. Januário, and D. Lima. The usefulness of software metric thresholds for detection of bad smells and fault prediction. *Inf. Softw. Technol.*, 115:79–92, 2019.

[S20] M. Boussaa, W. Kessentini, M. Kessentini, S. Bechikh, and S. Ben Chikha. Competitive Coevolutionary Code-Smells Detection. In G. Ruhe and Y. Zhang, editors, *Search based software engineering*, volume 8084 of *LNCS sublibrary. SL 2, Programming and software engineering*, pages 50–65. Springer, Heidelberg, 2013.

[S21] Z. Chen, L. Chen, W. Ma, X. Zhou, Y. Zhou, and B. Xu. Understanding metric-based detectable smells in Python software: A comparative study. *Inf. Softw. Technol.*, 94:14–29, 2018.

[S22] S. R. Chidamber, D. P. Darcy, and C. F. Kemerer. Managerial use of metrics for object-oriented software: an exploratory analysis. *IEEE Transactions on Software Engineering*, 24(8):629–639, 1998.

[S23] T. Coq and J.-P. Rosen. The SQALE Quality and Analysis Models for Assessing the Quality of Ada Source Code. In A. Romanovsky and T. Vardanega, editors, *Reliable Software Technologies - Ada-Europe 2011*, volume 6652 of *LNCS sublibrary. SL 2, Programming and software engineering*, pages 61–74. Springer, Heidelberg, 2011.

[S24] S. Ducasse. Reengineering object-oriented applications.

[S25] K. El Emam, S. Benlarbi, N. Goel, W. Melo, H. Lounis, and S. N. Rai. The optimal class size for object-oriented software. *IEEE Transactions on Software Engineering*, 28(5):494–509, 2002.

[S26] W. Fenske, S. Schulze, D. Meyer, and G. Saake. When code smells twice as much: Metric-based detection of variability-aware code smells. In M. Godfrey, D. Lo, and F. Khomh, editors, *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 171–180. IEEE, Piscataway, NJ, 2015.

[S27] K. A. Ferreira, M. A. Bigonha, R. S. Bigonha, L. F. Mendes, and H. C. Almeida. Identifying thresholds for object-oriented software metrics. *Journal of Systems and Software*, 85(2):244–257, 2012.

[S28] T. G. S. Filó, M. Bigonha, and K. Ferreira. A catalogue of thresholds for object-oriented software metrics. *Proc. of the 1st SOFTENG*, pages 48–55, 2015.

[S29] F. A. Fontana, M. Zanoni, A. Marino, and M. V. Mantyla. Code Smell Detection: Towards a Machine Learning-Based Approach. In *2013 29th IEEE International Conference on Software Maintenance (ICSM)*, pages 396–399, Piscataway, NJ, 2013. IEEE.

[S30] M. Foucault, M. Palyart, J.-R. Falleri, and X. Blanc. Computing contextual metric thresholds. In Y. Cho and A. S. I. G. o. A. Computing, editors, *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, pages 1120–1125. ACM, 2014.

[S31] N. Göde and R. Koschke. Studying clone evolution using incremental clone detection. *Journal of Software: Evolution and Process*, 25(2):165–192, 2013.

[S32] M. Gradišnik, T. Beranic, S. Karakatic, and G. Mausa. Adapting God Class thresholds for software defect prediction: A case study. In M. Koricic, editor, *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 1537–1542. Croatian Society for Information and Communication Technology, Electronics and Microelectronics - MIPRO, Rijeka, Croatia, 2019.

[S33] S. Hassaine, C. S. Hamel, E. David, and W. Binkley. *Evaluating Design Decay during Software Evolution.* Doctoral Dissertation, Universite de Montreal, Monreal, P.Q., Canada, 2012.

[S34] S. Herbold, J. Grabowski, and S. Waack. Calculation and optimization of thresholds for sets of software metrics. *Empirical Software Engineering*, 16(6):812–841, 2011.

[S35] Y. Higo, S. Kusumoto, and K. Inoue. A metric-based approach to identifying refactoring opportunities for merging code clones in a Java software system. *Journal of Software Maintenance and Evolution: Research and Practice*, 20(6):435–461, 2008.

[S36] M. Kessentini, W. Kessentini, H. Sahraoui, M. Boukadoum, and A. Ouni. Design Defects Detection and Correction by Example. In *2011 19th International Conference on Program Comprehension*, pages 81–90, Piscataway, July 2011. IEEE.

[S37] M. Kessentini and A. Ouni. Detecting Android Smells Using Multi-Objective Genetic Programming. In *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems*, MOBILESoft '17, pages 122–132, Piscataway, NJ, USA, 2017. IEEE Press.

[S38] M. Kessentini, H. Sahraoui, M. Boukadoum, and M. Wimmer. Search-Based Design Defects Detection by Example. In D. Giannakopoulou and F. Orejas, editors, *Fundamental Approaches to Software Engineering*, volume 6603 of *LNCS sublibrary. SL 1, Theoretical computer science and general issues*, pages 401–415. Springer, Heidelberg, 2011.

[S39] W. Kessentini, M. Kessentini, H. Sahraoui, S. Bechikh, and A. Ouni. A Cooperative Parallel Search-Based Software Engineering Approach for Code-Smells Detection. *IEEE Trans. Softw. Eng.*, 40(9):841–861, 2014.

[S40] F. Khomh, S. Vaucher, Y.-g. Guéhéneuc, and H. Sahraoui. A Bayesian Approach for the Detection of Code and Design Smells. In B. Choi, editor, *QSIC 2009*, pages 305–314, Los Alamitos, Calif., 2009. IEEE Computer Society.

[S41] F. Khomh, S. Vaucher, Y.-g. Guéhéneuc, and H. Sahraoui. BDTEX: A GQM-based Bayesian approach for the detection of antipatterns. *Journal of Systems and Software*, 84(4):559–572, 2011.

[S42] L. Lavazza and S. Morasca. Identifying Thresholds for Software Faultiness via Optimistic and Pessimistic Estimations. In A. S. I. G. o. S. Engineering, editor, *10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 1–10, New York, NY], 2016. Association for Computing Machinery.

[S43] S.-J. Lee, L. H. Lo, Y.-C. Chen, and S.-M. Shen. Co-changing code volume prediction through association rule mining and linear regression model. *Expert Syst. Appl.*, 45:185–194, 2016.

[S44] H. Liu, Q. Liu, Z. Niu, and Y. Liu. Dynamic and Automatic Feedback-Based Threshold Adaptation for Code Smell Detection. *IEEE Trans. Softw. Eng.*, 42(6):544–558, 2016.

[S45] X. Liu and C. Zhang. DT : a detection tool to automatically detect code smell in software project. In Y. Fang and Y. Xin, editors, *Proceedings of the 2016 4th International Conference on Machinery, Materials and Information Technology Applications*, volume 71 of *Advances in Computer Science Research*, pages 681–684, 2016.

[S46] R. Mahouachi, M. Kessentini, and K. Ghedira. A New Design Defects Classification: Marrying Detection and Correction. In J. D. Lara and A. Zisman, editors, *Fundamental Approaches to Software Engineering*, volume 7212 of *LNCS sublibrary. SL 1, Theoretical computer science and general issues*, pages 455–470. Springer, Heidelberg, 2012.

[S47] R. Marinescu. Detection strategies: metrics-based rules for detecting design flaws. In *ICSM 2004*, pages 350–359. 2004.

[S48] N. Moha, Y.-g. Guéhéneuc, L. Duchien, and A.-F. Le Meur. DECOR: A Method for the Specification and Detection of Code and Design Smells. *IEEE Trans. Softw. Eng.*, 36(1):20–36, 2010.

[S49] A. Mori, G. Vale, M. Viggiato, J. Oliveira, E. Figueiredo, E. Cirilo, P. Jamshidi, and C. Kastner. Evaluating domain-specific metric thresholds. In *TechDebt 2018*, TechDebt '18, pages 41–50, New York, New York and [Los Alamitos, California], 2018. The Association for Computing Machinery and IEEE Computer Society, Conference Publishing Services.

[S50] H. V. Nguyen, H. A. Nguyen, A. T. Nguyen, and T. N. Nguyen. Mining interprocedural, data-oriented usage patterns in JavaScript web applications. In *Proceedings of the 36th International Conference on Software Engineering*, pages 791–802. ACM, Hyderabad, India, 2004.

[S51] P. Oliveira, F. P. Lima, M. T. Valente, and A. Serebrenik. RTTool: A Tool for Extracting Relative Thresholds for Source Code Metrics. In *Proceedings, 30th International Conference on Software Maintenance and Evolution*, pages 629–632. Conference Publishing Services, IEEE Computer Society, Los Alamitos, California, 2014.

[S52] P. Oliveira, M. T. Valente, and F. P. Lima. Extracting relative thresholds for source code metrics. In *Software*

Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week - IEEE Conference on, pages 254–263. IEEE, 2/3/2014 - 2/6/2014.

[S53] A. Ouni, M. Daagi, M. Kessentini, S. Bouktif, and M. M. Gammoudi. A Machine Learning-Based Approach to Detect Web Service Design Defects. In I. Altintas, S. Chen, and I. I. C. o. W. Services, editors, *2017 IEEE 24th International Conference on Web Services - ICWS 2017*, pages 532–539, Piscataway, NJ, 2017. IEEE.

[S54] A. Ouni, R. Gaikovina Kula, M. Kessentini, and K. Inoue. Web Service Antipatterns Detection Using Genetic Programming. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO '15, pages 1351–1358, New York, NY, USA, 2015. ACM.

[S55] A. Ouni, M. Kessentini, M. Ó Cinnéide, H. Sahraoui, K. Deb, and K. Inoue. MORE: A multi-objective refactoring recommendation approach to introducing design patterns and fixing code smells. *Journal of Software: Evolution and Process*, 29(5):e1843, 2017.

[S56] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. de Lucia, and D. Poshyvanyk. Detecting Bad Smells in Source Code using Change History Information. In E. Denney, editor, *Proceedings of the 28th IEEEACM International Conference on Automated Software Engineering*, pages 268–278, Piscataway, NJ, 2013. IEEE Press.

[S57] R. Ramler, K. Wolfmaier, and T. Natschlager. Observing Distributions in Size Metrics: Experience from Analyzing Large Software Systems. In *31st Annual International Computer Software and Applications Conference (COMPSAC 2007): Beijing, China - 24-27 July 2007*, pages 299–304, [Place of publication not identified], 2007. IEEE Computer Society Press.

[S58] L. Rodriguez and Angela. *Assessing the effect of source code characteristics on changeability*. PhD thesis, Open University, 2009.

[S59] D. Sahin, M. Kessentini, S. Bechikh, and K. Deb. Code-Smell Detection as a Bilevel Problem. *ACM Trans. Softw. Eng. Methodol.*, 24(1):1–44, 2014.

[S60] G. Saranya, H. K. Nehemiah, and A. Kannan. Hybrid particle swarm optimisation with mutation for code smell detection. *Int. J. Bio-Inspired Comput.*, 12(3):186, 2018.

[S61] R. Shatnawi. A Quantitative Investigation of the Acceptable Risk Levels of Object-Oriented Metrics in Open-Source Systems. *IEEE Transactions on Software Engineering*, 36(2):216–225, 2010.

[S62] R. Shatnawi. Deriving metrics thresholds using log transformation. *Journal of Software: Evolution and Process*, 27(2):95–113, 2015.

[S63] R. Shatnawi, W. Li, J. Swain, and T. Newman. Finding software metrics threshold values using ROC curves. *Journal of Software Maintenance and Evolution: Research and Practice*, 22(1):1–16, 2010.

[S64] I. Shoenberger, M. W. Mkaouer, and M. Kessentini. On the Use of Smelly Examples to Detect Code Smells in JavaScript. In G. Squillero and K. Sim, editors, *Applications of evolutionary computation. Part II*, volume 10200 of *LNCS sublibrary. SL 1, Theoretical computer science and general issues*, pages 20–34, Cham, Switzerland, 2017. Springer.

[S65] B. L. Sousa, P. P. Souza, E. M. Fernandes, K. A. M. Ferreira, and M. A. S. Bigonha. FindSmells: Flexible Composition of Bad Smell Detection Strategies. In I. I. C. o. P. Comprehension, editor, *2017 IEEE 25th International Conference on Program Comprehension - ICPC 2017*, pages 360–363, Piscataway, NJ, 2017. IEEE.

[S66] G. Vale, E. Fernandes, and E. Figueiredo. On the proposal and evaluation of a benchmark-based threshold derivation method. *Software Quality Journal*, 27(1):275–306, 2019.

[S67] R. Vasa, M. Lumpe, P. Branch, and O. Nierstrasz. Comparative analysis of evolving software systems using the Gini coefficient. In *2009 IEEE International Conference on Software Maintenance*, pages 179–188, New York, 2009. IEEE.

[S68] B. Vasilescu, A. Serebrenik, and M. van den Brand. You can't control the unfamiliar: A study on the relations between aggregation techniques for software metrics. In *27th IEEE International Conference on Software Maintenance (ICSM), 2011*, pages 313–322, Piscataway, NJ, 2011. IEEE.

[S69] H. Wang, M. Kessentini, and A. Ouni. Bi-level Identification of Web Service Defects. In Q. Z. Sheng, E. Stroulia, S. Tata, and S. Bhiri, editors, *Service-oriented computing*, volume 9936 of *LNCS sublibrary. SL 2, Programming and software engineering*, pages 352–368. Springer, Switzerland, 2016.

[S70] H. Washizaki, H. Yamamoto, and Y. Fukazawa. A metrics suite for measuring reusability of software components. In *9th International software metrics symposium*, pages 211–223. IEEE Comput. Soc, 2003.

[S71] K. Yamashita, C. Huang, M. Nagappan, Y. Kamei, A. Mockus, A. E. Hassan, and N. Ubayashi. Thresholds for Size and Complexity Metrics: A Case Study from the Perspective of Defect Density. In *2016 IEEE International Conference on Software Quality, Reliability and Security*, pages 191–201, Los Alamitos, California, 2016. IEEE Computer Society, Conference Publishing Services.

[S72] C. Zhifei, C. Lin, M. Wanwangying, Z. Xiaoyu, Z. Yuming, and X. Baowen. Understanding metric-based detectable smells in Python software: A comparative study. *Inf. Softw. Technol.*, 94:14–29, 2018.

[S73] Y. Zhou, B. Xu, and H. Leung. On the ability of complexity metrics to predict fault-prone classes in object-oriented systems. *Journal of Systems and Software*, 83(4):660–674, 2010.

Towards a Catalog of Refactoring Solutions for Enterprise Architecture Smells

Lukas Liß
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
lukas.liss@rwth-aachen.de

Henrik Kämmerling
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
henrik.kaemmerling@rwth-aachen.de

ABSTRACT

Enterprise Architecture (EA) embodies the integration between business and IT architectures which aims to optimize the business value of IT investment. The qualities of EA greatly support businesses in achieving their goals. EA smells can decrease the quality of this architecture. Recently, there has been an advance to transfer the concept of code smells into the domain of EA to support the identification of those weaknesses. To improve the quality of code, code refactoring solutions are an already well-known tool. But, despite the recent foray into the research of EA smells, the field of EA refactoring solutions still remains unexplored. To address this research gap, we introduce the concept of refactoring solutions to EA. Therefore, a mapping from EA smells to code refactoring solutions is built. Then, we transform these code refactoring solutions to EA refactoring solutions. As a result, a catalog of EA refactoring solutions is set up.

Keywords

Enterprise Architecture, Refactoring, Smell, Catalog

1. INTRODUCTION

Lately, the concept of code smells has been transformed into EA [20]. This opens up the opportunity as well as the need to transfer the concept of refactoring solutions to EA. This paper presents a collection of EA refactoring solutions and their transformation from code refactoring solutions, which is a solution to the business problem of refactoring EA. The refactoring solutions allow correcting found EA smells in a standardized way and help to improve the quality of an EA. Moreover, a definition of EA refactoring solutions can serve as a first step towards software that automatically refactors EA smells.

In order to obtain a collection of EA refactoring solutions it is important to answer the question which EA refactoring solutions can be derived from existing code refactoring solutions. The main research question is split up in three sub-questions to clarify and separate the steps needed to answer the research question.

RQ1: What EA refactoring solutions can be derived from the existing code refactoring solutions?

RQ1.1: What code refactoring solutions are known in scientific literature or other catalogs?

RQ1.2: What is an adequate definition of EA refactoring solutions?

RQ1.3: How to derive EA refactoring solutions from code refactoring solutions?

The structure of this paper derives from the research questions as follows. Section 2 describes the basic concepts of smells and their refactoring solutions in code as well as in EA. Section 3 describes the method we used to transform refactoring solutions from the domain of code to EA. In section 4 the resulting EA refactoring solutions are presented and grouped. A discussion of the results can be found in section 5. Afterward, threats to validity are pointed out in section 6. In the conclusion, the method and the results are summarized and future work is proposed. The resulting catalog of EA refactoring solutions and the related EA smells can be found in section 4.1.

2. KEY CONCEPTS AND RELATED WORK

2.1 Code Smells

A code smell is a "surface indication that usually corresponds to a deeper problem of the system" [17]. The benefit of this indication is that it speeds up the process of searching for errors. Although, it is not certain that there is a real problem connected to a smell. Therefore, it can sometimes be a valid consideration to not change the detected smell.

About code smells there exist the following major studies:

- Sabir et al. [4] presented a list of code smells that were object-oriented as well as service-based anti-patterns.
- Bogner et al. [10] focused his systematic literature review on service-based anti-patterns.
- The book *Refactoring: improving the design of existing code* by Martin Fowler [5] contains a collection of code smells.

2.2 Code refactoring solutions

Code refactoring solutions were introduced by William Griswold [6] in 1991 into the world of functional and procedural programming. One year later William Opdyke [18] transferred this concept to the world of object-oriented programming.

Code refactoring describes the process of changing the structure of code while preserving the semantics of the code. It aims to improve the following system quality features:

- *Maintainability:* The code is shorter, better named or rearranged. This makes it easier for a developer to understand the functionality of each section of the code. It improves debug speed and shortens the period of vocational adjustment [16].

- *Extensibility:* Software features are easier to add to the existing ones. This makes the software flexible enough to adapt to frequent changes in requirements. As a result, the overall technical debt of the code should be lower than before [13].

The work in the field of code refactoring solutions is strongly connected with the one in code smells. There are several collections of code refactoring solutions:

- The book *Refactoring: improving the design of existing code* by Martin Fowler contains a collection of code refactoring solutions [5].

- Kebir et. al. about *Automatic Refactoring of Component-based Software by Detecting and Eliminating Bad Smells* [12].

- In the book *Anti Patterns*, Brown et. al. collect refactoring solutions [2].

All of the refactoring solutions described in the resources above can be applied to code regardless of the context. This is achieved by a level of abstraction on top of the context-sensitive functionality of the code. But this level of abstraction is not enough to apply the refactoring solutions to EA. Therefore the code refactoring solutions needs to be transformed before refactoring EA.

2.3 EA

The *ISO/IEC/IEEE FDIS 42010:2011 standard* [9] describes architecture as following: “fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution”.

As stated by Booch [1], it is mandatory to differentiate between technical architecture, which “attends to the architecture of the software-intensive systems that support that business”, and EA, which ”attends to the architecture of a business that uses technology” [1]. When comparing EA to technical architecture, EA gives a more holistic overview and a higher abstraction level. EA should be an important tool in achieving and supporting the goals of the business and not be focused on the technical domain only [14].

According to Saint-Louis et al. [19] there still exist many varying definitions for EA despite a growing interest in recent years. Most of these definitions describe EA either as a “Deliverable”, “Process”, “Tool”, or “Discipline and Practice” [19]. One of the more common approaches is to define architecture as “a collection of artifacts (models, descriptions, etc.) that define the standards of how the enterprise should function or provide an as-is model of the enterprise” [11].

Winter and Fischer [24] introduced a separation of EA into different domains. As they propose, EA is subdivided in the five domains “Business Architecture”, “Process Architecture”, “Integration Architecture”, “Software Architecture”, and “Technology Architecture” [24]. Every domain has different subsystems which should communicate with each other (horizontal integration). Also, the domains should not be treated separately but have an integration with one another (vertical integration). The components of every domain should, despite being horizontally integrated, “organized in their own hierarchies” [20]. This scheme matches with the from the *TOGAF Standard* [23] proposed architecture domains “Business Architecture”, “Data Architecture”, “Application Architecture”, and “Technology Architecture”.

2.4 EA Smells

Due to their analogy with code smells, the definition of EA smells is derived from code smells.

They highlight common bad examples and patterns of EA design and can, when ignored, decrease the quality of an EA. Because an EA is supposed to support the goals of a business [1] this could harm the performance of an organization as a whole. To prevent this, EA smells should be taken care of or at least be documented and kept in mind when taking future EA design decisions. EA smells should assist an EA architect in improving an existing system and making unconscious EA debt visible. While using this method an automated approach in discovering these anti-patterns can be developed [20].

The later mentioned EA smells in this work are mainly based on the catalog proposed from Salentin [20]. Also, Salentin [20] comes up with the following categories: “Business Antipatterns”, “Application Antipatterns”, and “Technology Antipatterns” [22, 23].

3. METHOD

In this section, we present how we answer our research questions. At first, we collect code refactoring solutions from existing catalogs and scientific literature (*RQ 1.1*). Then, we set up a scheme to present the later found EA refactoring solutions (*RQ 1.2*). Finally, we evaluate how to derive EA refactoring solutions from the collected code refactoring solutions (*RQ 1.3*).

3.1 EA Refactoring Solutions

As introduced in chapter 2.4, EA smells need to be resolved if future harm is meant to be avoided. These solutions for EA smells are called EA refactoring solutions. They are best practices in transforming an anti-pattern to an improved state.

While code refactoring solutions serve the purpose of compensating technical debt, EA refactoring solutions compensate EA debt. In contrast to code refactoring solutions, EA refactoring solutions do not concentrate on changing single lines of code, but adjusting the design of an EA or certain components. As code refactoring solutions should not alter the semantics of the code, EA refactoring solutions should not change the function of an EA. An EA refactoring should lead to a more flexible and maintainable architecture, and thus should directly or indirectly support the business goals.

It is important to note that the refactoring solutions should not improve the quality of only the technological architecture but the EA as a whole. This approach is consistent with Booch's [1] differentiation of technical architecture and EA.

3.2 Refactoring Selection Process

Because there exist many code refactoring catalogs in scientific literature, it is mandatory to have a defined process of choosing the refactoring solutions for this catalog. Isaenko [8] discusses three different approaches in his thesis on microservice refactoring solutions. The approaches all have in common that they collect smells first, and then find refactoring solutions for the specified smells. As Isaenko [8] evaluates, the approach *Match Bad Smells to Microservice Smells* leads to the most information to work with, and therefore, has the most sound basis of those three different approaches. Due to the need of building a catalog with a valid basis, we chose this approach and adapted it to the domain of EA (*Match Code Smells to EA Smells*). This approach consists of six main steps [8]:

1. Collect code smells from defined reliable resources.
2. Take one code smell from the bank.
3. Extract the general idea of the design issue the code smell represents.
4. Find a characteristic of EA that indicates the same design issue as the original code smell.
5. Adapt the code smell to an EA smell based on the identified design issue.
6. Adapt the code smell refactoring techniques to EA refactoring solutions.

Because this work is based on the EA smell catalog proposed from Salentin [20], we mainly focus on step 6.

3.3 Transformation to EA

As mentioned earlier, this work starts with a pre-defined EA smell catalog. Salentin[20] derived those EA smells from code smells. For said code smells already exist code refactoring solutions. The whole list of EA smells can be found in Salentin's catalog [20].

3.3.1 Transformation Process

The next step is to adapt the code refactoring solutions connected to the EA smells into the domain of EA. At first, we collect those code refactoring solutions from scientific catalogs and literature. Most of those refactoring solutions can be found in *Refactoring: improving the design of existing code* by Martin Fowler [5], *Automatic Refactoring of Component-based Software by Detecting and Eliminating Bad Smells* by Kebir et al. [12], and *AntiPatterns: refactoring software, architectures, and projects in crisis* by William H. Brown[2]. The most relevant aspect in matching the refactoring solutions are the earlier detected code smells and connected EA smells.

Figure 1 shows the necessary steps to build up the catalog. As earlier mentioned, the *Matching* phase was already done by Salentin [20]. Then, there is the need for *Discovering* the matching code refactoring solutions. After that, the *Transforming* of code refactoring solutions to EA refactoring solutions is the remaining step. Because there exist no established rules for this transformation yet, guidelines for this process need to be proposed. Therefore, the next section explains the transformation process in more detail.

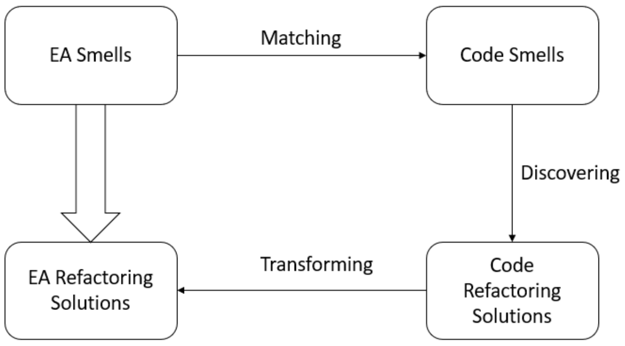


Figure 1: Visualization of the process we used to get EA refactoring solutions from given EA smells

3.3.2 Transforming Code Refactoring Solutions to EA Refactoring Solutions

The main step in this work is to adapt code refactoring solutions to EA refactoring solutions. Later, the adapted EA refactoring solutions need to be evaluated towards their applicability [8].

The code refactoring solutions, which need to be adapted, are associated with at least one code smell, which is already matched to an EA smell. Also, there are analogies in the adaption of smells and refactorings. Therefore, the matching process between the code smell and the EA smell can be taken into consideration when transforming code refactoring solutions to EA refactoring solutions. The code refactoring solutions need to be closely investigated to get awareness on the possible motivation for the refactoring solutions in the field of EA. Due to the high abstraction level of EA, the motivation is most of the time also on a higher abstraction level. The mechanics of the EA refactoring solutions can be adapted from the mechanics of the code refactoring solution. When doing this, the components involved in the mechanics can be transformed from the domain of code to the domain of EA.

Still, it can occur that some refactoring solutions cannot be transformed. Also, due to the abstraction level of EA, some distinctive code refactoring solutions could be transformed to the same EA refactoring. Therefore, it needs to be considered, that different components from the domain of code can be matched to the same component in the domain of EA. Because these guidelines are not that well established yet, they may change during different iterations of the transformation process.

3.4 Categorization of EA Refactoring Solutions

As mentioned earlier, there already exist definitions of sub-domains of EA in scientific literature. Because the catalog of EA refactoring solutions is based on Salentin's [20] catalog of EA smells, the categories for this catalog are taken from the EA smells catalog. From the four architecture domains proposed in *The TOGAF Standard, Version 9.2* [23] and the three layers proposed by the *ArchiMate 3.0.1 Specification* [22], Salentin [20] derived the following three main categories:

- **Business anti-patterns** impact business services, which are realized in the organization by business processes performed by business actors.

- **Application anti-patterns** impact application services that support business and applications that realize them.
- **Technology anti-patterns** impact technology services that support the business and its applications, including IT infrastructure such as logical software and hardware capabilities, communication, processing or storage, etc.

The EA refactoring solutions found in this catalog are assigned to those three categories based on the categorizations in the EA smell catalog [20].

These three categories gain their legitimation as they are based on two well-established EA subset definitions. Although, as Salentin [20] states, this taxonomy “does not provide a complete mapping” yet and should be further developed in the future.

3.5 Representation of EA Refactoring Solutions

To achieve a consistent catalog of EA refactoring solutions there needs to be a reusable scheme for the representation of said refactoring solutions. This scheme also leads to a more defined method of transforming the refactoring solutions to EA, because this scheme should be applicable to most EA refactoring solutions.

The used scheme is derived from the one Isaenko [8] uses in his catalog on microservice refactoring solutions [15, 21, 13]. It consists of 10 sections:

Name. The name should give the reader a first insight into what the EA refactoring is about and what it does.

Connected EA smells. This should list the EA smells which can be resolved by applying this refactoring solution.

Derived from. This is the code refactoring which this EA refactoring is derived from.

Summary. A short summary to explain what this refactoring is about. This should not explain the refactoring in detail or describe the steps necessary to perform this refactoring. It should serve as a first point of information after that the readers can decide, whether it is necessary to look into the following sections.

Intent. This describes the main goal of the refactoring. It explains to the reader *what* will be done in the refactoring. This should not already explain the mechanics in detail.

Motivation. While the intent describes the *what*, the motivation describes the *why* of the refactoring. This section should explain the reasons to use said refactoring.

Prerequirements. For some refactoring solutions there can be conditions for applying the refactoring. Those conditions should be stated here. It is possible for a refactoring to have no prerequirements.

Impact. The goal of any refactoring is to improve the quality of an EA. Therefore, this section should describe which quality factors are affected in which way.

Mechanics. The mechanics should be concrete steps on how to perform the refactoring. After understanding the mechanics, the reader should be able to perform the refactoring.

Discussion. In the discussion should be evaluated when to use said refactoring. Also, possible trade-offs should be mentioned.

Example. There should be given a small visual example. All the examples given in this catalog are from one domain of EA.

The categorization is added to the connected EA smells, as the categorization is based on these smells. We used abbreviations to shorten the resulting catalog. The letter ”b” represents ”business anti-patterns”, the letter ”a” represents ”application anti-patterns”, and the letter ”t” represents ”technology anti-patterns”.

3.6 Validation of transformed EA Refactoring Solutions

A good validation is important when building a refactoring solution catalog, because then, the quality and applicability in real-life scenarios, and therefore, the usability of the catalog can be evaluated. To qualify a valid refactoring, it needs to be applicable in a possible scenario. Therefore, the refactoring solutions in this catalog are derived from EA smells and have an explicit application. Other than that, validation helps to identify mistakes and also can be used to improve the mechanics of each refactoring. To validate a refactoring, there should be a first evaluation if there is a practical use for it. Second, it should be assessed if the refactoring actually leads to an improvement in EA quality. Also, it needs to be classified if the refactorings are realistically applicable. A refactoring can theoretically improve the quality of an EA, but does not have any value, if the mechanics cannot be performed.

Additionally, the method of setting up the catalog needs to be evaluated. It needs to be checked for internal and external validity.

4. RESULTS

As seen in figure 2, the transformation process starts with Salentin’s [20] catalog of 49 EA smells. 20 out of those 49 EA smells are directly derived from code smells and can, therefore, be easily matched with those code smells. The other 29 EA smells are left out, because they are not directly derived from code smells, and therefore, do not fit the used method. For the 20 matched code smells, we found 20 code refactoring solutions in scientific literature and catalogs. Then, we transformed those 20 code refactoring solutions to EA refactoring solutions.

For the catalog, we use the scheme proposed in section 3.5. Due to space limitations, we leave out the *Intent* and *Motivation*, as they are already described in the *Summary*. In the catalog, the *Summary* follows directly after the refactoring name. The *Examples* are also left out because of space limitations.

4.1 Catalog of EA Refactoring Solutions

5 Viewpoints. As different viewpoints and different stakeholders focus on different parts of a system, it is necessary

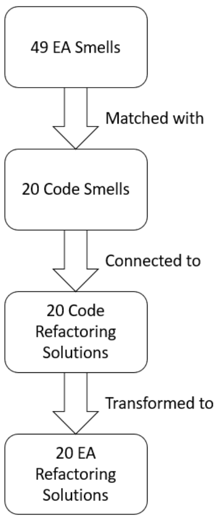


Figure 2: Visualization of the amount of smells and refactoring solutions for each step during the set up of the catalog

to document the viewpoint together with the model of the system. This refactoring does so by using the viewpoints corresponding to the layers of EA: Business, application, technology, strategy, implementation, and migration [24].

Connected EA smells. Ambiguous Viewpoint (a, b, t)
Derived from. 3 Viewpoints: Business, Application or Technology
Prerequirements. None
Impact. Efficiency: The program can be understood faster. And it is easier to evaluate if this model fits your own viewpoint.
Mechanics. 1. Define the Layer the model belongs to (Business, Application, Technology, Strategy, Implementation & Migration). 2. If the viewpoint is mixed, differentiate the model by the viewpoint connected to the layer. 3. Explicitly note down the viewpoint.
Discussion. Sometimes even different stakeholders with the same viewpoint focus on different parts of a model. Then it can be useful to differentiate them as well and note down to which stakeholder they belong.

Add Middleman. When a service is dependent on the implementation of a second service, it can lead to issues when the second service needs to be changed. This can be resolved by adding a level of abstraction to the second service. Then, the first service becomes independent of the actual implementation of the second service. This leads to a more maintainable system.

Connected EA smells. Message Chain (a, t)
Derived from. Hide Delegate
Prerequirements. A chain of services, where the first one depends on the structure of the following because the first one operates on this structure.
Impact. Efficiency: It decreases as abstraction increases. The model will be harder to understand and an extra layer of abstraction costs resources. Maintainability: As there is less coupling and dependency on the structure of the architecture, it becomes easier to maintain. This is especially the case when multiple components depend on the same structure of service calls.
Mechanics. 1. Check if there already exists an abstraction of the service chain. Then use this one by changing the calls in the first component to calls to the abstraction service. Done. 2. Else: Create a new Service with an interface. 3. For each needed trigger in the chain of services starting by the second, create one in the interface. 4. The new service delegates these calls to the chain of services. Test after one is added. 5. Change calls, where the first service calls the second one, to calls to the newly created one. 6. Test.
Discussion. When chains are long this refactoring can be applied multiple times within the rest of the chain. But this will increase complexity.

Architecture Framework. A stovepipe system has high complexity and is hard to adapt. When using architecture frameworks to introduce a component architecture, the system becomes more flexible and adaptable. Therefore, efficiency and maintainability are increased.

Connected EA smells. Stovepipe System (a, b, t)
Derived from. Architecture Frameworks
Prerequirements. None.
Impact. Adaptability and Extensibility: Due to the nature of a component architecture it is easy to add new components to extend the architecture and adapt to changes. Maintainability: Changes need to be made only at a single place. Efficiency: Work can be split according to the different services.
Mechanics. 1. Introduce a component architecture. 2. Identify a base level of functionality that most of the applications should support (mostly for data interchange and conversion). 3. Define a system interface for those base functionalities. 4. Create a base level of component services. 5. Create thin application services that use the base services. Those application services should only add specialized functions and interfaces.
Discussion. Exceptions: Mock-ups or prototype systems. When entering a new domain it can be useful to build a stovepipe system first to gather domain knowledge.

Architecture Partitioning. Intermixed vertical and horizontal design elements destabilize an architecture. This refactoring stabilizes it by partitioning the design elements. Each layer is separated and an interface is created to manage the interaction between the layers.

Connected EA smells. Jumble (a, b, t)
Derived from. Architecture Partitioning
Prerequirements. An architecture with intermixed design elements.
Impact. Maintainability: A clear design is a key aspect to make the system modifiable.
Mechanics. 1. Partition the architecture with respect to horizontal and vertical design. 2. Separate the groups and define an interface for interaction if needed. Use the *Move Component* refactoring to move the components in their group.
Discussion. A clear separation between design elements improves the maintainability a lot. But an experienced worker is needed to determine how to group the elements.

Enterprise Architecture Planning. There is a big effort in maintaining a stovepipe system. To lower the resources needed for maintaining the system, it is important to increase the consistency of the architecture by using EA planning. A consistent architecture is easier to understand and because of that, adjustments are more straightforward.

Connected EA smells. Stovepipe System (a, b, t)
Derived from. Enterprise Architecture Planning
Prerequirements. None.
Impact. Efficiency: The fewer resources are concerned with maintaining the system, the more resources can focus on the business goals themself. Maintainability: A consistent architecture needs less maintenance than a stovepipe system.
Mechanics. 1. Use EA planning to coordinate system conventions at several levels. 2. Use *Move Component*, *Extract Component*, *Merge Components* and other refactoring solutions to refactor the complete system regarding the new conventions.
Discussion. When the architecture has already grown to a certain size, it can be a lot of effort to refactor it as a whole. This process can lead to a lot of challenges and can result in further problems. Therefore, the taken steps need to be evaluated carefully in advance.

Extract shared functionality. Duplication increases complexity and is hard to maintain. This refactoring reduces duplication. The functionality that multiple components have in common is centered on a separate component that is triggered by the others. Then it is possible to make changes to this functionality in one place.

Connected EA smells. Duplication (a, b, t)
Derived from. Pull up Method
Prerequirements. Multiple components have at least partially the same functionality.
Impact. Maintainability: Changes can be done in one place. But this increases the coupling of the components.
Mechanics. 1. Move the duplicated functionality into a new component. 2. Everywhere, where the duplicated functionality appears, change it to use the new component. 3. Test after each replacement. 4. When a component now only calls the newly created, use *Encapsulate Component* to remove it.
Discussion. It is important to keep in mind that this refactoring increases the coupling in exchange for lowering duplication. Especially in the context of microservices a certain amount of duplication is tolerated.

Ghostbusting. Components with limited responsibility that simply pass messages clutter the architecture. As their responsibility is so limited they should be removed. This refactoring removes these components. The functionality of the architecture is sustained because the limited responsibility to pass the messages is transferred to the callers, so they have to send the message directly.

Connected EA smells. Lazy Component (a, b, t)
Derived from. Poltergeist, Remove Middle Man
Prerequirements. A component that is short-lived or is just delegating messages to other components.
Impact. Efficiency: Faster to understand due to de-cluttered design. Maintainability: Less abstraction leads to less complexity, but also comes with the cost of flexibility.
Mechanics. 1. Remove the lazy component. 2. Every trigger that triggered the lazy component before now needs to directly trigger the one that was triggered by the lazy component. 3. Test.
Discussion. The evaluation whether a level of abstraction is needed or not is a hard one in some cases. As well as increasing the understandability of the architecture it also decreases the flexibility.

Goal Question Architecture. Experienced architects tend to not document implicit architecture. This can lead to a hidden risk. To remove them, this refactoring aims to document all parts of the system, also the implicit ones.

Connected EA smells. Architecture by Implication (a, b, t)
Derived from. Goal Question Architecture
Prerequirements. None
Impact. Efficiency: A Person with less domain-specific knowledge can understand the model faster. Maintainability: Less hidden risk.
Mechanics. 1. An experienced person document all the implications into the model.
Discussion. Implications are risky as it is very time-consuming to find out about them. When an experienced employee leaves the company there is a danger that a lot of implicit knowledge is lost.

God Object Decomposition. When one component has too many purposes it can become too convoluted. This leads to unclear EA where the tasks of each component are not well-defined. To tackle this issue, the god object can be split into multiple components with a well-defined purpose to increase coherence [3].

Connected EA smells. The God Object (a, b, t)
Derived from. God Class Decomposition
Prerequirements. There exists a bloated service, that does too many tasks, so his purpose is not well-defined anymore.
Impact. Maintainability: It is easier to change the process of different tasks when the task has a dedicated service. Extensibility: It is easier to extend the functionalities of the services when creating a new well-defined service instead of extending a bloated service. The internal structure of each service does not need to be known to extend the functionalities this way.
Mechanics. 1. Identify the tasks of the bloated service. 2. Arrange those tasks into purposeful groups. 3. Create a new service for each group. 4. Check where the service is accessed from outside and change the usages. 5. Test. 6. Remove the bloated service.
Discussion. Most of the time, a bloated service is not the result of bad design, but it grows in time whenever new features are added. Therefore, it is necessary to evaluate if a service can be split whenever new features are added to it.

Inline Service. Services with very simple and short functionality can be integrated into callers of the service. Thereby the architecture gets de-cluttered. This refactoring moves copies of all the sub-components from a simple service into all the components that call the service. Then the service can be removed safely. As this refactoring increases duplication, it should only be applied to services with simple functionality that change very rarely.

Connected EA smells. Lazy Component (a, b, t)
Derived from. Inline Function
Prerequirements. A service with very little complexity. The components that realize this service are self-explaining.
Impact. Efficiency: Faster to understand due to de-cluttered design. Maintainability: Less abstraction leads to less complexity. As duplicates of the former service are now in every

caller changing this functionality will be harder than before.
Mechanics. 1. Move components from the service into every component that triggers the service. Start with the one with the least dependency, the last one triggered in the service. Use the *Move Component* refactoring to do so. 2. Remove the service. 3. Test.
Discussion. When the service is triggered by many components and needs to change often, the duplication created by this refactoring will be extreme. In this case, it could be useful to keep the abstraction. But when this is not the case, this refactoring can improve the understandability of the architecture.

Isolation Layer. When using a proprietary product there can be a strong dependency on the vendor. To resolve this dependency, it can be necessary to introduce an additional isolation layer. Then, most services do not interact with the bought-in product directly, but interact with the isolation layer. This reduces the risks involved in being too dependent on one single vendor and also increases the extensibility of the architecture.

Connected EA smells. Vendor Lock-in (a, t)
Derived from. Isolation Layer
Prerequirements. Changes in the vendor product are anticipated and the isolation layer will be changed accordingly.
Impact. Maintainability: New features can be added easier through the isolation layer. Reduced Risk: The dependency on the vendor will be reduced.
Mechanics. 1. Detect what functions of the product are used. 2. Create a layer between the services used by business actors and the bought-in product.
Discussion. The vendor lock-in is acceptable when a single vendor's product makes up the majority of the used business functions.

Merge Components. An EA can become too complex when there are multiple components with a shared concern or functionality. This leads to poor maintainability and therefore decreases efficiency. To take care of this scattered functionality the components with shared concern can be merged into one component to increase cohesion.

Connected EA smells. Scattered Parasitic Functionality (a, b, t)
Derived from. Merge Components
Prerequirements. Components with a shared concern.
Impact. Efficiency: The created component is easier to reuse. Maintainability: The complexity is reduced, as it is easier to find which components are responsible for what.
Mechanics. 1. Create a new component. 2. Move all the sub-components from the concerned components into the new one. Use Move Component refactoring to do so. Begin with the one with the least dependencies. 3. Delete the old components. 4. Test.
Discussion. Sometimes it is difficult to decide if to service have the same function or serve a slightly different purpose. Then, the different components need to be evaluated carefully.

Merge Input. A service with a long list of required input is hard to understand. This refactoring reduces this complexity by grouping the input. A new data-object with all the required information is created. Then this data-object is given to the service and not all the individual data.

Connected EA smells. Bloated Service (a, b, t)
Derived from. Replace Parameter with Query
Prerequirements. A Service that has a lot of required information from a common object.
Impact. Efficiency: The service becomes less complex to understand.
Mechanics. 1. If necessary merge the required information into an abstract object. 2. Make the service require this object. 3. Replace each use of a required input with the referenced object. Test after each change.
Discussion. It can be hard to decide which information to merge into an abstract object. There is needed to find a well-suited metric in future work.

Move Component. As architecture is constantly changing sub-components sometimes need to be moved from one component to another. Starting with sub-component with the least dependency, this refactoring moves sub-components. It can be necessary to rename the moved components with the rename component refactoring.

Connected EA smells. Feature Envy (a, b, t)
Derived from. Move function
Prerequirements. A sub-component that belongs to a component and another component where the sub-component should be moved to.
Impact. Efficiency: Moving a component can increase understandability. Maintainability: Moving a component can make the architecture less complicated.
Mechanics. 1. Decide if by this component triggered components should move too. Then move them first with the *Move Component* refactoring. 2. Copy the component to the target context. Adjust it to fit in there. 3. If necessary rename the component with the *Rename Component* refactoring. 4. Refer from the old home to the moved component. 5. Remove the component from the old home. 6. Test.
Discussion. Whether the impact of moving a component is positive or negative depends on the reason why to move the process. In general, moving a process is mainly only meaningful when it removes a smell.

Move Service to different Layer. When having an architecture with several layers, every layer must be well-defined. This architecture can be violated when services access other services from non-adjacent layers. By moving one or more services to their respective layers, the structure of the architecture can be made more coherent [7].

Connected EA smells. Strict Layer Violation (a, b, t)
Derived from. Move Class
Prerequirements. Having an architecture with multiple layers.
Impact. Maintainability: When every service is in the layer it belongs, the system is easier to understand and thus, can be easier maintained.
Mechanics. 1. Identify a service that uses a service from a non-adjacent layer. 2. Identify which of those two services is in the wrong layer. 3. Add a copy of the service to the right layer. 4. Update every usage of the service. 5. Test. 6. Remove the old service. 7. Reuse when this refactoring leads to another service that uses a service from a non-adjacent layer.
Discussion. Sometimes it is hard to identify which service should be moved to which layer. It can be useful to create a delegate service in the layer between the two services. Also,

sometimes this refactoring can lead to more layer violations. In those two cases, it can be better to leave the architecture in its current state.

Remove Dead Component. Unused components clutter the model of a system. This refactoring removes unused components in a safe way. Thereby this increases the maintainability and efficiency.

Connected EA smells. Dead Component (a, b, t)
Derived from. Remove Dead Code
Prerequirements. The component is not used. There are no concrete plans to use this component in the near future.
Impact. Efficiency: Resources consumed by the unused component are available again. Maintainability: Removing the unused component lowers the complexity.
Mechanics. 1. If the component has an interface then check that no one refers to this. 2. Remove the component. 3. Test.
Discussion. Sometimes an unused component is intended to serve as a documentation in case this component or a similar component is needed in the future. But as even unused components consume resources this is not a good way.

Rename Component. Unfitting names for components slow down the process of understanding the model. They can even lead to errors when they are misleading. This refactoring changes the name of a component to increase efficiency.

Connected EA smells. Deficient Names, Documentation (a, t)
Derived from. Rename Field
Prerequirements. A component with a name that does not fit well to the functionality of the component.
Impact. Efficiency: Faster to understand.
Mechanics. 1. If the scope of use of this component is limited: 1.1 Rename the component. 1.2 Rename all accesses. 1.3 Test. 2. Else: 2.1 Use Add Middleman to create abstraction. 2.2 Rename internally. 2.3 Adjust internal accesses. 2.4 Test.
Discussion. Renaming a component should only be done when the old name is not fitting anymore or was never fitting. Otherwise, it could lead to increased search times when the name is changed too often.

Small Project. A process with too many business actors involved can decrease productivity. Also, few milestones with a big scope instead of more milestones with smaller scope decrease the motivation and therefore productivity of every business actor. To resolve this, large teams working on one process can be split into smaller groups that work on sub-tasks for this process.

Connected EA smells. Warm Bodies (a)
Derived from. Small Project
Prerequirements. There exists a process with more than five business actors involved or a duration longer than four months.
Impact. Efficiency: Smaller project teams need less coordination and are more likely to succeed. Therefore, there are fewer resources needed to fulfill the goals of the process.
Mechanics. 1. Split large project teams into smaller groups of four people. 2. Split the main task into smaller sub-task that can be solved by the smaller groups. 3. Split big milestones into smaller ones.
Discussion. Processes with 100 or more actors involved have very low efficiency. Working in smaller groups will bring the efficiency level up. Although, coordinating too many teams can produce coordination overhead and can, therefore, lower efficiency again.

Split Phase. There can be a component with multiple responsibilities and a low cohesion in general. This can lead to decreased maintainability because of the component's high complexity. Then, the component can be split into one separate component for each responsibility of the old component, to increase maintainability.

Connected EA smells. Multifaceted Abstraction (a, b, t)
Derived from. Split phase
Prerequirements. A component where the cohesion is low.
Impact. Maintainability: This separation increases the change that the components can be changed independently.
Mechanics. 1. Extract the second part in its own component using the *Extract Component* refactoring. 2. Test this component. 3. Introduce an intermediate interface and data object. 4. Test. 5. The first component now triggers the second one.
Discussion. When there are more than just two phases this refactoring can be done recursively to split the phases until they only have one responsibility.

4.2 Transformation

A full description of all the refactoring solutions would be very long. Therefore, we demonstrate the process with the Remove Dead Component refactoring.

The goal is to create a refactoring for the EA smell called Dead Component [20]. First, it is necessary to understand the EA smell. It can be helpful to clarify the structure that defines the smell especially as this is needed for the prerequirements attribute of the EA refactoring. In the given example of a Dead Component, Salentin states that "isolated elements" (p. 47 [20]) are elements, where the element itself, as well as sub-elements, have no incoming or outgoing behavioral dependencies to other components. We used this structure as the prerequirements for the found EA refactoring.

The next step is to match a code refactoring to the EA smell. The given EA smell was derived from the code smell Dead Code. Fowler describes a refactoring for this smell called Remove Dead Code [5]. We extracted the main idea of the given code refactoring: First, check if the thing you want to delete is really unused. Then remove it. Then test.

Now the terminology of the main idea needs to be matched to EA: First, check if the component you want to delete has no external behavioral dependencies. Then remove it. Then test.

By matching the terminology the mechanics of the EA refactoring are derived from the main idea. This is also done with the motivation and intention. The remaining attributes of the EA refactoring definition, like summary, impact, or example, can be derived from the already defined attributes. In the end, we derived the following definition of the Remove Dead Component refactoring:

Name. Remove Dead Component

Connected EA Smells. Remove Dead Component

Derived from. Remove Dead Code

Summary. Unused components clutter the model of a system. This refactoring safely removes unused components. Thereby this increases maintainability and efficiency.

Intent. The intent is to remove unused components.

Motivation. Unused components do not signal that they are unused at a first glance. Therefore it takes time to understand them. People and resources will be concerned with this component until it is removed.

Prerequirements. The component is not used. There are no concrete plans to use this component in the near future.

Impact. Efficiency: Resources consumed by the unused component are available again. Maintainability: Removing the unused component lowers the complexity.

Mechanics. 1. If the component has an interface then check that no one refers to this. 2. Remove the component. 3. Test.

Discussion. Sometimes an unused component is intended to serve as documentation in case this component or a similar component is needed in the future. But as even unused components consume resources this is not a good way.

Example. The example is shown in figure 3. An insurance company modeled the business process of dealing with a request to review a previous decision. In the past, they used to print the result of this process to document it in their files. Nowadays they only save the document digitally but the "Print Documentation" process is still in the model. By deleting it the model is de-cluttered.

5. DISCUSSION

An aspect that needs to be discussed is the legitimation. As described before testing the EA refactoring solutions is not in the scope of this paper. Thus the legitimation for real-world applicability remains uncertain. But the main idea of the EA refactoring solutions gains legitimation from the fact that the used code refactoring solutions are well established.

Also, there may be additional EA refactoring solutions than the ones that can be derived from code refactoring solutions. The reason for this is that EA has a higher level of abstraction than the domain of code. Thus there could be refactoring possibilities that exist in the domain of EA but not in the domain of code. These possibilities stay unexplored when transforming code refactoring solutions to EA refactoring solutions.

In addition to that, for EA smells, that have no corresponding code smell, it is not possible to find a refactoring with the used method.

Despite these limitations, the used method follows established guidelines. Therefore, the results of this method can be validated easier. Moreover, the transformation process can be replicated and further advanced. It is precisely these characteristics that are decisive in a field as young as EA refactoring solutions.

A discussion about each EA refactoring can be found in the catalog in section 4.1.

6. THREATS TO VALIDITY

6.1 Internal Validity

Threats to internal validity are influences that can affect the independent variable with respect to causality [25]. As this work relies on several systematic literature reviews it is necessary to point out the threats to internal validity that derive from that. Although guidelines have been developed for performing a systematic literature review, there are still subjective influences possible. Soft criteria for including and excluding work for example. In addition, the transformation process from a code refactoring into an EA refactoring has subjective parts as well. There are multiple definitions or slight variations of most code refactoring solutions. Hence we needed to extract a common idea out of all of them. Thereby subjective perspectives are a threat to internal validity.

6.2 External Validity

Threats to external validity are conditions that limit the ability to generalize the result [25]. EA operates with a high level of abstraction. Because of this, it is important that the resulting EA refactoring solutions are generalizable enough to be used in a wide variety of architectures. Thus we tried to minimize dependencies that could be a threat to external validity. We focused on the structure of the architecture to able to generalize the refactoring solutions to every viewpoint and stakeholder. But there are threats remaining that need to be taken under consideration. All the EA smells we created a refactoring for are derived from code smells. That makes the process of transformation only suitable for those smells. Another threat to external validity is the limited number of sources. The resources used in this work, as well as this work itself, base on a limited number of datasets and search engines. Because of this, there are variations and special cases that may not be taken into consideration. Here we especially need to point out that there was only one source for EA smells available.

7. CONCLUSION AND FUTURE WORK

In this paper, we present a catalog of EA refactoring solutions. We propose refactoring solutions for recently introduced EA smells. We adapt code refactoring solutions into EA refactoring solutions to come up with said catalog. The process of adapting the refactoring solutions was modified from already known code refactoring adaptions. We derived an existing refactoring representation scheme for the catalog due to the particularities of EA. This should improve the reusability of this scheme, because of its similarity to already established schemes. This scheme should also encourage the extensibility of the catalog, due to its general applicability to EA refactoring solutions.

The catalog should serve as a starting point for the research field of EA refactoring solutions. It can be further extended with other EA refactoring solutions derived from code refactoring solutions an EA refactoring solutions coming from different domains of EA (i.e. Process Architecture). The catalog is right now in its first iteration. It is part of the future work to publish the full catalog including the examples, which are left out in this paper due to space limitations. Because the refactoring solutions are derived from theoretical sources, they need to be tested in a real-world environment. Other than that, the catalog can be reviewed

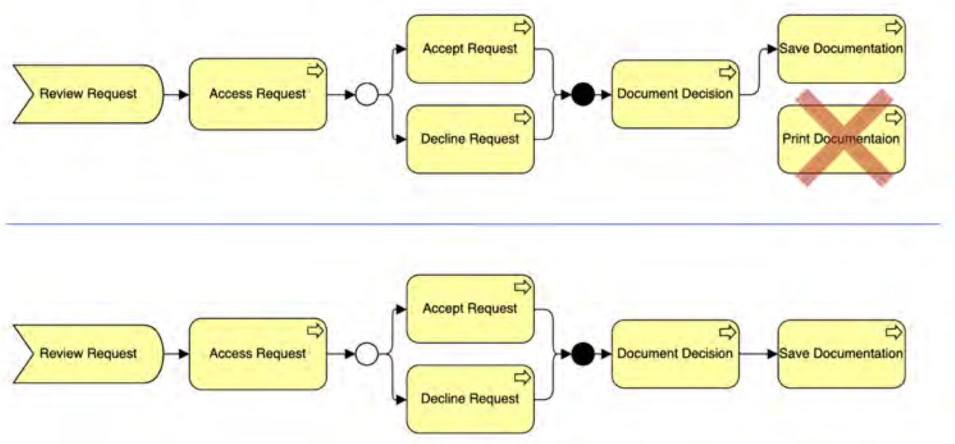


Figure 3: EA refactoring: Remove Dead Component

to increase its validity. Also, other sources for refactoring solutions and EA smells should be taken into consideration.

References

[1] G. Booch. Enterprise architecture and technical architecture. *IEEE Software*, 27(2):96, 2010.

[2] W. H. Brown, R. C. Malveau, H. W. McCormick, and T. J. Mowbray. *AntiPatterns: refactoring software, architectures, and projects in crisis.* John Wiley & Sons, Inc., 1998.

[3] B. Du Bois, S. Demeyer, J. Verelst, T. Mens, and M. Temmerman. Does god class decomposition affect comprehensibility? In *IASTED Conf. on Software Engineering*, pages 346–355, 2006.

[4] F. Sabir et al. A systematic literature review on the detection of smells and their evolution in object-oriented and service-oriented systems. *Software: Practice and Experience*, (49.1):3–39, 2019.

[5] M. Fowler. *Refactoring: Improving the design of existing code.* A Martin Fowler signature book. Addison-Wesley and Pearson Education , Boston and Amsterdam and London and Boston and Amsterdam and London, second edition edition, 2019.

[6] W. Griswold. Program restructuring as an aid to software maintenance. Juli 1991.

[7] S. Hickey and M. O. Cinn é ide. Search-based refactoring for layered architecture repair: An initial investigation. In *Proc. 1st North American Search Based Software Engineering Symposium*, 2015.

[8] V. Isaenko. Towards a catalog of refactorings for microservices. 2019.

[9] ISO/IEC/IEEE, editor. *Systems and software engineering — Architecture description: ISO/IEC/IEEE 42010:2011(E).*

[10] J. Bogner et al. Towards a collaborative repository for the documentation of service-based antipatterns and bad smells. Mar. 2019.

[11] L. Kappelman, T. McGinnis, A. Pettite, and A. Sidorova. Enterprise architecture: Charting the territory for academic research. *AMCIS 2008 Proceedings*, page 162, 2008.

[12] S. Kebir, I. Borne, and D. Meslati. Automatic refactoring of component-based software by detecting and eliminating bad smells. In *Proceedings of the 11th International Conference on Evaluation of Novel Software Approaches to Software Engineering*, pages 210–215. SCITEPRESS-Science and Technology Publications, Lda, 2016.

[13] J. Kerievsky. *Refactoring to patterns.* Pearson Deutschland GmbH , 2005.

[14] M. Lankhorst. *Enterprise Architecture at Work.* Springer Berlin Heidelberg , Berlin, Heidelberg, 2017.

[15] F. Martin et al. *Refactoring: improving the design of existing code.* Pearson Education India, 1999.

[16] R. Martin. Clean code. 2009.

[17] Martin Fowler. Codesmell, 09.02.2019.

[18] W. Opdyke. Refactoring object-oriented frameworks. 1992.

[19] P. Saint-Louis, M. C. Morency, and J. Lapalme. Defining enterprise architecture: A systematic literature review. In S. Hall é , editor, *2017 IEEE 21st International Enterprise Distributed Object Computing Conference workshops - EDOC 2017: 10-13 October 2017, Quebec City, Quebec, Canada : proceedings*, pages 41–49, Piscataway, NJ, 2017. IEEE.

[20] J. Salentin. Towards a catalogue of enterprise architecture smells and their detection. 2019.

[21] G. Suryanarayana, G. Samarthyam, and T. Sharma. *Refactoring for software design smells: managing technical debt.* Morgan Kaufmann, 2014.

[22] The Open Group. Archimate 3.0.1 specification.

[23] The Open Group. The togaf standard, version 9.2. 2018.

[24] R. Winter and R. Fischer. Essential layers, artifacts, and dependencies of enterprise architecture. In *Proceedings / 10th IEEE International Enterprise Distributed Object Computing Conference: Workshops]; Hong Kong, China, October 16 - 20, 2006*, page 30, Piscataway, NJ, 2008. IEEE.

[25] C. Wohlin, P. Runeson, M. H ö st, M. C. Ohlsson, B. o. r. Regnell, and A. Wessl é n. *Experimentation in software engineering.* Springer, Berlin, 2012.

Data Processing Frameworks: What is the right tool for my task?

Philipp C. Peeß
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
philipp.peess@rwth-aachen.de

Christian Schwier
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
christian.schwier@rwth-aachen.de

ABSTRACT

Data processing is a central topic in this digital age, both on a very large corporate scale as well as in smaller businesses and private projects. But with the large amount of available frameworks and the variety of different requirements for projects it is difficult to find the right framework for a task. This paper will explain the importance of key characteristics the processing mode, the architecture, and the interface and language support, to support making the right decision. Furthermore, five important data processing frameworks, Apache Hadoop, Apache Spark, Apache Storm, Apache Heron, and Apache Flink will be compared based on these aspects. Additionally, a few commercial solutions as well as frameworks which provide a visual coding approach for data processing will be presented. Even though no generally best framework is found, advice is given for a small number of scenarios and use-cases.

Keywords

Data Processing, Stream Processing, Processing Frameworks, Comparison

1. INTRODUCTION

Large scale data processing is a wide spread requirement in modern businesses as well as research. Well known companies which rely heavily on their capability to handle big data are Google, Facebook, or Twitter. CERN is one of the most prominent examples in academia which collected 12.3 petabytes of data in a single month [9]. Besides big international parties like the ones named above, scientists from the machine learning and the Industry 4.0 [6] communities are noticeable demand carriers. Furthermore, the growing Internet of Things (IoT) and in particular the Industrial Internet of Things (IIoT) share challenges caused by the high amount of devices which provide data to a process. The global amount of generated data in 2018 was estimated to be 33 zettabytes and is expected to grow to 175 zettabytes by 2025 [14]. Working with the large amounts of data gathered in these contexts presents distinct challenges. It requires efficient algorithms, well-organised server infrastructures and solutions that integrate well with existing soft- and hardware.

Data processing frameworks are a family of software stacks which promise to handle these scenarios. In the last decade a variety of them have emerged with focus on different aspects of the problem domain. The *Apache Foundation* in particular, hosts many projects related to this topic, e.g., *Apache Spark*[1], *Apache Storm*[2], *Apache Heron*[3], *Apache Flink*[4], *Apache Beam*[5], and *Apache Samza*[6]. In addition, commercial players like Google and IBM advertise proprietary solutions which are hosted on their cloud platforms.

With the rise of ubiquitously accessible cloud computing infrastructure these technologies are no longer exclusive to companies but become available for individuals. Picking an appropriate platform for a given use case is still a non-trivial task due to the variety of characteristics the systems have. This paper presents a guide to help with the selection process by aggregating and comparing current noteworthy systems.

This paper is structured as follows: First, we discuss a set of criteria used to characterize data processing frameworks in Section 2. In Sections 3 to 5 we present a selection of systems split in three categories: The Hadoop Eco-System, Commercial Solutions, and Special Systems. Lastly, we compare the systems in relation to different design vectors in Section 6.

2. CHARACTERISTICS

This paper will focus mainly on three aspects which are crucial when choosing a framework: the offered processing modes, the architecture the framework uses for structuring the nodes in clusters, and the offered interfaces and supported programming languages as described below.

[1]`https://spark.apache.org` (visited: 11/02/2019)
[2]`https://storm.apache.org` (visited: 11/02/2019)
[3]`https://heron.apache.org` (visited: 11/02/2019)
[4]`https://flink.apache.org` (visited: 11/02/2019)
[5]`https://beam.apache.org` (visited: 11/02/2019)
[6]`https://samza.apache.org` (visited: 11/02/2019)

2.1 Processing Mode

The processing mode is probably the most important difference between the different frameworks. It determines for which applications the framework is suitable. The covered approaches are *batch processing* and *stream processing* as well as *micro-batching* as an in-between solution.

2.1.1 Batch processing

Batch processing is the traditional approach for computing large amounts of data. This approach usually requires a static, most of the times historical collection of data as input. The data is then processed with little to no human interaction, outputting the result when finished.

2.1.2 Stream processing

The obvious restriction of batch processing is that all data has to be available before processing can be started. *Stream processing* is a more modern approach which became more common when demand of near real-time data processing began to rise. Instead of receiving a single collection of data, these frameworks are capable of dealing with a constant data stream. Data is processed as soon as it is received. This makes them capable of dealing with input like constant sensor data, ongoing event logs, and similar data sources.

2.1.3 Micro-batching

Between batch and stream processing, a third solution exists. So-called *micro-batching* uses principles of batch processing to emulate real stream processing. Incoming data is grouped into batches, which are regularly processed depending on chosen rules. Different methods for creating these batches exist, processing could for example start when a certain threshold of data has been met or a fixed time interval has passed.

2.1.4 Comparing Processing Modes

While stream processing enables the use for near real-time applications, traditional batch processing still has advantages in certain situations. It can be more efficient for complex operations on static data. In addition to that, not all algorithms can be used efficiently with stream processing in their current implementation [4]. Furthermore, it is easier to implement fault tolerance and load balancing while simultaneously have a higher throughput with batch processing [12]. However, stream processing can deal with any case that batch processing can compute and more [4], even though for pure batch processing applications it should not be the preferred solution due to the reasons above. Its main advantage is the ability to process unbounded data streams.

Micro-batching on the other hand comes with its own advantages and drawbacks. The manual splitting of the data stream into smaller batches leads to abstraction which can be disadvantageous. In addition to that, micro-batching generally has a worse performance when compared to native stream processing. It is also more difficult to maintain states across multiple different batches of data. On the other hand, micro batching maintains the advantages of normal batch processing previously mentioned [12]. It is a good compromise when mainly batch processing is needed, but stream processing should also be supported.

Overall, all processing modes come with their own advantages and drawbacks, making the decision central to the design and success of an application.

2.2 Architecture and Scaling

Another central aspect is the underlying architecture of the framework. When data processing is only done in a small scale with a single machine, this is no significant concern. But when large amounts of data need to be processed or processing time is crucial, single machines hardly are an option.

2.2.1 Clusters

To be capable of processing large amounts of data, so-called clusters are used. These are networks of many connected real or virtual machines called *nodes*. The processing job is then distributed into smaller tasks which are run in parallel on these nodes.

While this increases the processing capabilities, it also creates problems that have to be solved. The tasks and resources have to be scheduled and distributed, making a resource manager necessary. Node failures have to be expected in large clusters, making fault tolerance a fundamental requirement.

In addition to that, the organisation of the cluster needs to allow easy and efficient scaling options. Scaling is generally done horizontally by adding more nodes as the workload increases, as opposed to vertically, where single machines receive more computation power.

2.2.2 Scheduling and Resource Management

In general, two things have to be managed when executing a job on a cluster: the task distribution and the resource allocation. Most architectures use one or more special master nodes for managing this. These take care of distributing the tasks and the resources in the cluster, managing the task progress, and keeping track of the node status. In special cases like node failures or exceptionally slow progress in a task, they may intervene and recover or redistribute the task or approximate solutions ahead of time to provide a temporary result.

While each application usually has its own cluster, newer approaches allow for pooling of resources across multiple server clusters, enabling multiple applications to share the resources. Using this approach, resources can be used more efficiently and assigned to different applications based on their current needs. This makes it very advantageous when frameworks allow the use of such a resource and cluster management.

2.2.3 Processing Guarantees

The architecture can also allow for specific processing guarantees. Three different options exist here: *exactly-once-*, *at-most-once-* and *at-least-once-processing*. In at-least-once-processing, every bit of data is processed at least once, but potentially multiple times. While this can be inefficient regarding the use of resources, it also can decrease processing time in case of single node failures, if the same data was simultaneously processed on multiple nodes. At-most-once processing is the other extreme, where data may potentially be never processed. This is useful for near-real-time applications, where new data may already be available, making timely reprocessing after failures possibly unwanted. Exactly-once-processing is the most complex guarantee, but can in return assure that every part of the data is processed exactly once.

2.3 Interfaces and Languages

An argument that should not be overlooked is the compatibility of frameworks with pre-existing software and hardware. Different frameworks often require different programming languages and come with distinct application programming interfaces (APIs).

Ideally, the development should be possible in a programming language the developers are already familiar with and that is fitting for the problem. Some frameworks come with the ability to use simple std-in and std-out. Instead of only being able to use programming languages supported by the framework, this makes it possible to create scripts in any programming language that supports std-in and std-out. These scripts are then run on the nodes and communicate with the framework using only std-in and std-out. This can be a major advantage, providing much more flexibility in the choice of programming language.

Choosing a fitting API is also crucial. Some technologies can be mandatory, for example support for specific file systems, preferred database structures or for certain input sources. Ideally, support of these technologies would already be offered by the framework itself. Otherwise, depending on the importance of these technologies for the project, interfaces may have to be implemented manually or switching to different solutions has to be considered. Additionally, frameworks often offer features like pre-made operations for data processing. This can simplify the process of developing complex programs.

3. SYSTEMS

In the following, five data processing frameworks will be presented in detail: Hadoop, Spark, Storm, Heron, and Flink. The focus of this section will be on the characteristics mentioned above in addition to special features and key implementation details.

3.1 Hadoop MapReduce

Hadoop is a widely used batch processing frameworks which also spawned a large ecosystem around it. The focus of this section will be mainly on the MapReduce algorithm, the foundation of data processing with Hadoop. In addition, the Hadoop Distributed File System (HDFS) will be discussed in detail, since it is the second core component of the Apache Hadoop. Its architecture provides the necessary components to support large-scale data processing with high amounts of nodes. Unless otherwise noted, this section is based on the information provided in the official documentation [3].

3.1.1 Processing: The MapReduce Algorithm

The idea of the MapReduce algorithm was proposed by Google developers in 2004 [5]. After that, development on an open source implementation began, leading to the creation of Hadoop and the implementation of Hadoop MapReduce.

The MapReduce algorithm consists, as the name suggests, of two parts, a map function and a reduce function. A job is split into many smaller tasks that can then be executed on different nodes in parallel to use a cluster efficiently. The input data is distributed on a series of map-processes, which then start the work. This is done by the framework which splits the input into partitions. A subtask is created for each of the partitions, resulting in each of the data splits being processed separately.

The map function takes an input and applies the key-value-combination needed for processing the data. It is possible to start the processing with any sort of data, the map function is then used to create the fitting key-value pairs. If key-value-pairs were already associated with the data, then the map function creates new intermediate key-value-pairs.

The reduce function then applies the actual transformation and works in three phases. In the shuffle phase, data is transferred from the nodes which mapped the data to the nodes that are tasked with the reduce job. Simultaneously, the sort phase takes place. In the sort phase, the key-value-pairs are sorted and then grouped by the key. Thus, a list of values is now associated to every key. Thereafter, the reduce function is applied, iteratively reducing the list of values that were associated with a key. In this process, the processed data is merged and the result is calculated. The output can then be stored in the file system.

It is possible to apply a combiner function between mapping and shuffling. This function is run on the node which also mapped the data before the data is passed to different nodes. Using this, the logic of the reduce function could already be applied to the separate data splits before they are passed into the network. This can, if applicable, reduce the network traffic by already merging equal keys or reformat data to allow easier processing in the following steps.

3.1.2 The Hadoop Distributed File System

A key component that is often compatible with other frameworks or even necessary for them to work is the HDFS. As such, it is also interesting to discuss the HDFS in detail. It supports a variety of useful features, with the most prominent ones being fault tolerance, support for large distributed data sets, and tools for cluster rebalancing.

The HDFS uses a master-slave-architecture to organize the file system. The cluster nodes are divided into the *NameNode* which is the master and the *DataNodes* which are the slaves that handle the actual storage.

The NameNode organizes the filesystem. It manages the namespace of the filesystem as well as the file access and operations like renaming files or directories. Because of their size or available storage, files may be partitioned into multiple smaller files, blocks, and stored on different DataNodes. The creation of these blocks and the assignment to DataNodes is also task of the NameNode. DataNodes reply to read or write requests by the NameNodes and perform the requested action.

The HDFS is fault tolerant, with high availability implemented in current versions. Fault tolerance is achieved by replicating data blocks and storing them on different nodes. The NameNode regularly checks the status of the DataNodes and the stored blocks. When the DataNodes do not reply, failures are assumed. Due to the data replication, the block of data can be retrieved from a different node.

The high availability is achieved by running redundant NameNodes. With only a single NameNode managing the entire cluster, a single point of failure exists. As soon as the NameNode fails, the files can no longer be accessed. To prevent this, a second NameNode can be used. This secondary NameNode regularly receives backups by the primary NameNode and is otherwise on standby. If the primary NameNode fails, the secondary NameNode takes its place.

Because of the regular backups, the state of the file system is mostly known at any point in time and can be restored easily.

3.1.3 APIs and Ecosystem

The Hadoop framework comes with a whole ecosystem of sub-systems that are widely used by other frameworks.

A noteworthy part of the ecosystem is Hadoop YARN (Yet Another Resource Negotiator)[7]. It is, as the name suggests, a tool to manage the resources of a cluster. The core concept is the division between a global ResourceManager and ApplicationMasters for each application which accept resource requests and supervise the nodes. With the introduction of YARN, it became possible to use different processing engines like Apache Spark instead of MapReduce. This made the framework very flexible.

In addition to that, a large amount of related projects exist which make Hadoop more versatile. As an example, a machine learning engine, Hadoop Submarine[8], exists.

Hadoop is mainly used with Java, but any language can be used with it since it offers data input via std-in and output via std-out using Hadoop Streaming.

3.2 Spark

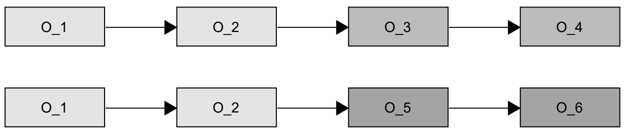
Apache Spark is a cluster computing system implemented in *Java* and *Scala* with a focus on large batch processing. Its central concept is the *Resilient Distributed Dataset (RDD)* which is a data sharing abstraction. This allows a running process to be oblivious if it is processing the complete data or only a partition. The computational model is similar to MapReduce, that is a sequence of operations which either map a value via an operation or reduce multiple values. Applying an operation on an RDD creates a new RDD. The abstraction allows tasks to access data while its physical location is transparent to the process. By default these RDDs are ephemeral. If two different subsequent computations are to be done on an RDD it has to be explicitly persisted. This step allows sharing of intermediate results between different computations. As a consequence it is possible to create a tree of computations instead of a set of linear processes. This is depicted in Figure 1. Since RDDs are immutable there is no risk of side effects due to concurrent computations on the same data set.

To provide fault tolerance each RDD tracks its lineage, that is, the history of operations on previous data sets it was derived from [16]. This allows Spark to avoid the costly materialization of intermediate results to a hard disk or a network storage. In case of a failure Spark can recompute the required data from the last known data set. Since the history of a data set can become quite large, manually checkpointing intermediate results allows to limit the amount of recomputations. This trade-off between recomputation in case of failure and processing slow down due to I/O-Operations is configurable by the user to align it with its requirements.

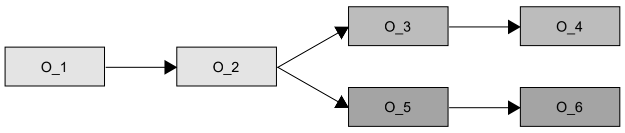
3.2.1 Architecture

A process implemented with Spark has an application at its center which is called *Driver*. It uses a *Cluster Manager* which distributes tasks on a cluster. Spark ships its own implementation of a cluster manager but can connect to others,

[7]https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html (visited on 12/13/2019)
[8]https://hadoop.apache.org/submarine/ (visited on 12/13/2019)



(a) Two computations with ephemeral intermediate results.



(b) Two computations with a materialized intermediate result after *O_2*.

Figure 1: A comparison of two independent computations with and without a persisted intermediate result.

e.g., YARN or Mesos[9], if they are already present on the infrastructure. The only requirement is that the manager can spawn an executor process on any of its worker nodes. The driver contains the application and the spark context which allows interaction with the executor processes. It is the drivers responsibility to run jobs on the executors which notify the driver of results of the computation. To enable this, the executors need network access to the driver. The executors should be run close to where the required data is stored (regarding network topology) to reduce I/O overhead.

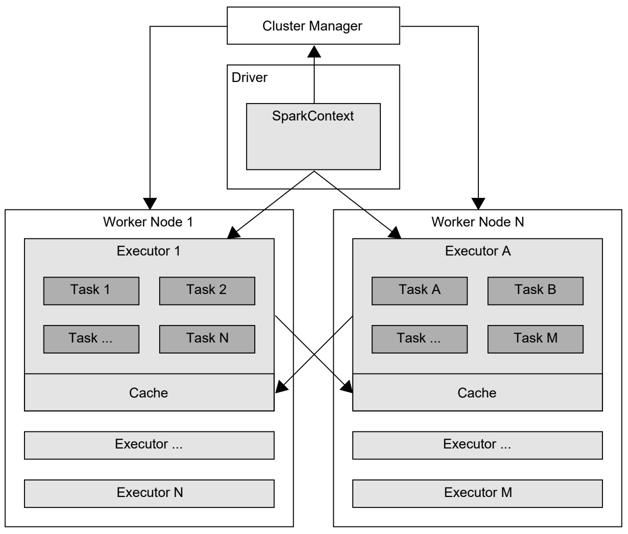


Figure 2: Spark cluster architecture. Spark uses multiple *Nodes* which each run multiple *Executors*. The *Driver* holds process state in the *SparkContext* and uses the *Cluster Manager* to distribute its *Tasks* on the executors.

It is important to note that a Spark application is not a single data processing job but a complete application which spawns jobs on the cluster continuously over its lifetime. To achieve this it uses the *Spark Context* which hides most of the complexity. An overview is given in Figure 2.

[9]http://mesos.apache.org/ (visited on 11/20/2019)

3.2.2 Spark Streaming

Spark Streaming is an extension to *Spark* designed to support continuous live data processing. Instead of working on a single RDD representing a partition of a large batch data set, Spark Streaming uses *Discretized Streams (D-Streams)* [17]. This is a sequence of RDDs which are created by sampling the incoming data into small batches in a given interval. Other stream processing systems, e.g., Apache Storm, use long running operators which continuously consume and produce data. Spark Streaming models its computations as functions on an element of an RDD and invokes them each time a new micro batch is aggregated. Since the batches are provided as RDDs, the advantages from Spark translate to Spark Streaming.

To exploit the full potential of this approach the functions are required to be 1) *stateless* and 2) *deterministic*. If both requirements are fulfilled it allows to run multiple instances concurrently on partitions of the D-Stream. If state is required it has to be shared between the different processes which incurs I/O overhead for the communication and synchronization of mentioned state. The functions can be run on different machines which is orchestrated by a scheduler controlling the distribution.

3.3 Apache Storm

Apache Storm "is a real-time distributed stream data processing engine" [15] originally developed at *Twitter* and open sourced as an Apache Incubator project. At first, it was implemented in *Clojure* but was re-written in *Java* for its 2.0.0 release to increase the accessibility for contributors. The core concept of Storm is the *Topology* which is a graph of processing vertices which are connected via data streams. The vertices are split into two categories: *Spouts* and *Bolts*. The former are sources with only outgoing edges, which are used to introduce data into the system. The latter are generic processing nodes, which consume data from their predecessors and emit data to their successors. The data tokens are modeled as tuples of arbitrary types.

3.3.1 Architecture

Storm uses a distributed cluster to execute topologies as shown in Figure 3. The computations of a topology are done on *worker nodes* which are physical machines in a Storm cluster. A master node, called *Nimbus*, orchestrates all worker nodes. When a new topology is started it is submitted to *Nimbus*. This sends instructions to the *Supervisors* running on the worker nodes which manage multiple *worker processes*. The actual computations are done by the worker processes. They separate the workload on multiple *executors* which are threads within the worker process. Each of the executors runs tasks of a single Bolt or Spout.

This architecture allows Storm to distribute the computation for a topology over multiple physical machines. The executors isolate the Bolts and Spouts to avoid conflicts between different topologies. A cluster of heterogeneous computational power is supported by allowing each worker node to run a different amount of worker processes.

3.3.2 Processing Guarantees

Storm provides a mechanism to track tuples through the topology which allows it to provide at-least-once, at-most-once and exactly-once-semantics for the processing. This is implemented by attaching a unique 64-bit number to each

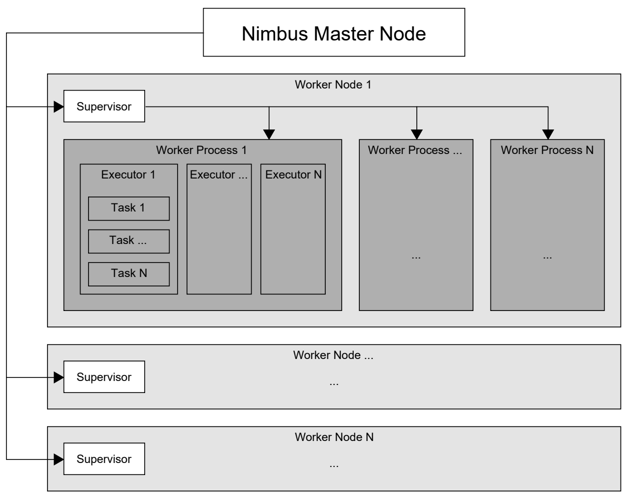


Figure 3: Storm architecture overview. A nimbus supervisor manages multiple worker nodes by communicating with the supervisors running on them. Each supervisor spawns a configurable number of worker processes which start an executor thread per Bolt/Spout they are running.

created tuple. When a tuple $\mathtt{t}_n$ is received and as a new tuple $\mathtt{t}_m$ is created this relation is stored by declaring $\mathtt{t}_n$ the anchor of $\mathtt{t}_m$. It is possible for a tuple to have multiple anchors if it was computed as the result of multiple predecessors. This creates a directed a-cyclical graph (DAG) representing the pedigree for each tuple. For historic reasons this graph is called *tuple tree*. Bolts can then either fail or acknowledge a digested tuple. In the first case, the source tuples are replayed which then causes the computations through the graph to be done again. When all tuples in a tuple tree are acknowledged the source message is considered *fully processed* and the meta data is discarded. If this does not happen within a configurable timeout the tuple is considered to have failed and the Spout will replay its message. This mechanism allows fine grained configuration of computational dependencies and trade-offs between processing time and reliability. Since Storm's topologies can contain cycles the user has to ensure that these processing loops terminate to prevent triggering timeouts indefinitely.

3.3.3 APIs and Ecosystem

In addition to its *Trident API* Storm, provides less verbose wrappers for Java and Python. These allow for easy configuration and execution of topologies. Since Storm runs on the JVM the preferred method to implement Bolts is Java although every other JVM compatible language works as well. Other technologies can be used by spawning the Bolt or Spout as a separate process which is connected to Storm via a JSON-based protocol communicating on std-in and std-out. Adapter implementations for Ruby, Python and Fancy are shipped with the default distribution.

3.4 Heron

Heron [10] is Twitter's successor to Storm which aims at overcoming the limitations which became apparent during its deployment. Heron still shares a lot of the terminology with similar semantics like the topology which consists of

Spouts and *Bolts*. Since it was aimed to replace Storm in production, one of Herons key features is the API compatibility. While this has little impact in a green field project it allows for an easier migration of existing solutions. One of the key benefits is the possibility to use existing shared infrastructure where Storm required dedicated resources for itself. Furthermore, the different tasks in a Heron process are more isolated from each other which allows for better resource management and performance monitoring. Additional design decisions aimed at increasing the separation of concerns cause a more complex architecture but decoupled.

3.4.1 Architecture

While a Heron topology configuration is conceptually similar to Storm's, the materialization is vastly different. First, each Spout and Bolt task is executed in a separate *Heron Instance* which is a JVM process instead of sharing the same physical process. This enables Heron to manage the resources a single Spout has available in a more fine grained manner and prevents failures from directly impairing other tasks. Next, Heron uses containers to group tasks and related processes which can be deployed on a server cluster without requiring pre-installed Heron specific services on each machine. One of the above mentioned related processes is the *Stream Manager* of which an instance resides in each container. It manages the routing of data between different Heron Instances within a container as well as between different containers. This special service allows for an easier monitoring of traffic and workload for each Heron Instance. The k Stream Managers create a network of $O(k^2)$ physical connections. If Heron Instances communicate within a container the connection is short circuited by the Stream Manager. All other connections from Heron Instances between two specific containers are multiplexed over a single connection between the associated Stream Managers. A *Metrics Manager* can be spawned to extract additional information, e.g., for debugging purposes.

When a topology is submitted a *Topology Master* is instantiated as an orchestrator. It creates the execution plan from the configuration and manages the execution through the complete life-cycle. Once the execution plan is created a set of containers is spawned via a resource management system. This can either be done with the Heron internal manager or deferred to the manager of a shared infrastructure like Mesos, YARN or ECS[10]. Each container contains a set of Heron Instances. One of the optimization criteria of the execution plan creation is to keep related Heron Instances close to each other, preferably in the same container, to reduce communication overhead. An exemplary Heron topology materialization is depicted in Figure 4.

3.4.2 Processing Mode

Heron, like Storm, is a pure stream processing framework. While a Spout can consume batch data and serialize it into the topology, the internal computation model works on streams of tuples.

Different to Storm, Heron implements multiple back pressure mechanisms. Their purpose is to prevent the repeatedly replaying of data from the Spouts in case a bottleneck in the processing graph is causing delivery failures. The simplest implementation uses the TCP buffers for back pressure propagation. When the receive buffer of a Heron Instance fills up the upstream sending buffers will fill up which is detectable by the sender. The disadvantage of this approach is that a single slow node can congest the physical connection between two containers. This will cause all upstream Heron Instances to stall, too, potentially bringing the complete topology to halt.

[10]https://aws.amazon.com/ecs/ (visited on 01/14/2020)

The second approach is to slow down Spouts in case of an overloaded Heron Instance. This directly reduces the amount of new tuples introduced into the topology. The local Stream Manager than communicate this information to the other Stream Managers which leads to a complete slow down of the topology. While this allows a single slow Heron Instance to slow down the complete computation it keeps the topology in a sane state where all data is delivered.

The last method is to propagate the back pressure from the faulting node backwards through the graph layer by layer up to the Spouts. This has the advantage to only affect its upstream nodes and to not globally slow down the processing.

3.5 Flink

Flink is a data processing framework which offers both batch and native stream processing. This makes it a very versatile data processing framework, allowing its usage in a variety of projects. Unless otherwise noted, this section is based on the information provided in the official documentation [2].

3.5.1 Processing Modes

Flink offers both batch and stream processing. This is particularly advantageous when both modes can be utilized in a project, e.g., when both large static collections of data as well as streams of inputs need to be processed. In addition, if experience with the framework has been acquired, it can be used well in almost any future project since both important processing modes are supported.

Stream processing is the central principle of Flink as data is always treated like streams. Flink programs consist mainly of two components, *streams* and *transformations*. Transformation refers to any operation with at least one stream as input. The result of a transformation is another stream. To realize parallel operations, streams are split into *stream partitions* and each operator can be split into subtasks.

During execution, a so-called dataflow graph is created consisting of the previously mentioned components. This dataflow is a DAG with at least one source and at least one sink, representing the general structure and flow of the program.

Stateful processing is one of the key features of Flink. Operators can require information of multiple events on a continuous, possibly unbounded stream of data. To resolve this, stateful processing is used. Windowing in Flink is a prime example of stateful processing. Flink offers the use of different *windows*, in which events and data are aggregated and can then be used collectively. As an example, these windows can aggregate data and events over a certain amount of time or store a certain amount of elements.

Fault tolerance and exactly-once processing are provided by Flink. For that, checkpoints are created, relating to the current point of the input stream as well as the state of the operators at that time. By recovering from a checkpoint, the state can be kept consistent when replaying the events starting at that point.

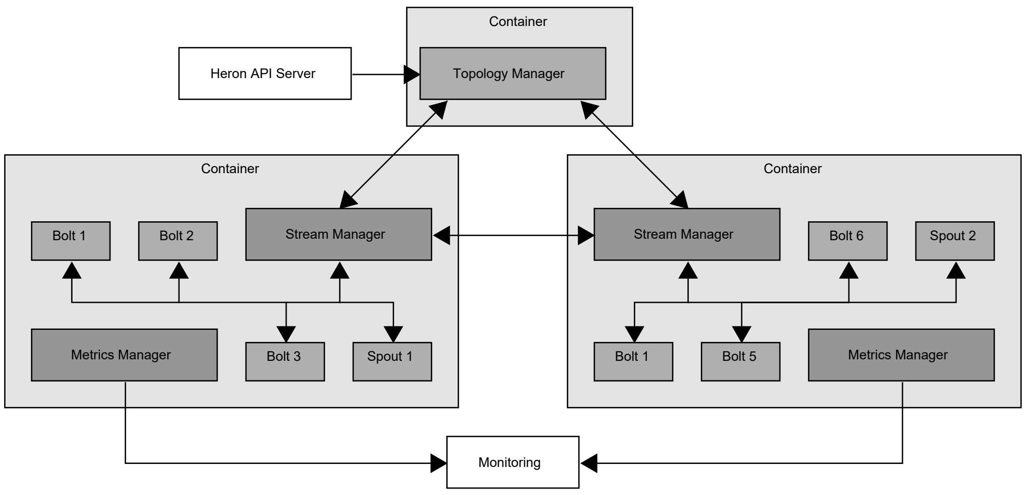


Figure 4: Heron architecture overview. Two containers with a stream manager and a metrics manager each are orchestrated by a topology manager. If multiple tasks are spawned for the same Bolt, here **Bolt 1**, they do not have to reside in the same container. The data between the containers is routed over the stream managers.

Batch processing in Flink is treated like stream processing, with the difference that it assumes a bounded stream since the input data set is finite. As a result of this, the previously presented ideas also apply with minor changes. Stateful operations are implemented in a different way since the data is available all at once, making windows obsolete. Additionally, no checkpoints or backups are made for fault tolerance. In case of failures, the whole data is simply reprocessed as this will always lead to the same end result. This may be inefficient when failures occur, but the cost for making checkpoints is saved and makes correct processing cheaper.

3.5.2 Architecture

Flink uses a slave-master-architecture to organize its nodes. The *JobManagers* are in control of coordinating the processing in the cluster. At least one has to exist, with the option of having additional JobManagers to guarantee high availability of the cluster. If multiple JobManagers exist, only one is actually managing the cluster. The other JobManagers are in standby and are used for keeping checkpoints. In case of a failure, recovery is easy because of the constant backups, enabling the system to always restore its state. Overall, Flink offers a comparably high fault tolerance for a stream processing framework [1]. The processing is done on *TaskManagers*, the workers which execute the tasks and subtasks.

Each operator in the dataflow graph is considered a subtask which needs to be processed. The framework offers some tools to speed up processing. An example is the so-called *chaining* of operators. Multiple operators can be linked together and are then treated as a single task. This can improve the performance since related operations can be run together on the same node instead of potentially being passed on to a new node which would increase overhead in terms of network traffic and coordination.

3.5.3 APIs and programming languages

An interesting feature of Flink is its Table API and SQL support. This allows for a relational view on the stream and batch process. Query operations can easily be done using either of these APIs with SQL being the higher-level API. The operations provided work the same for both, batch processing as well as stream processing, leading to the identical results.

While Flink does not provide its own storage management, it has a variety of *connectors* to third-party systems, notably Apache Kafka, the HDFS and Amazon Kinesis Streams.

In terms of programming languages, Flink currently supports Java, Scala and Python.

4. COMMERCIAL SOLUTIONS

Besides open source solutions various proprietary systems exist. In this section, two commercial data processing options will be presented: IBM Streams and Google Cloud Dataflow.

IBM Streams is a stream processing solution. It offers connectors to use data from outside IBM Streams as an input and options for exporting the data to external storage. In addition to that, a set of predefined operators and a high-level interface that require no manual interaction with low-level tasks are already implemented. Featuring a visual coding solution, IBM Streams allows a graphical approach that requires less understanding of the subject. This visual representation can be used for dashboard views of the processed data, as a graph view of the processing flow or as an overview of the current stream of tuples. Congestion detection, scaling during runtime as well as parallelization can also be done using the visual representation [8].

Google Cloud Dataflow is a tool for serverless data processing as part of the Google Cloud Platform. The implementation is based on Apache Beam, which is a very flexible framework. Apache Beam can run a variety of other execution engines like Spark, which makes it possible to use other, more suitable frameworks to process the data. During development, the processing logic is defined in pipelines. Cloud Dataflow then manages the backend, focussing on cluster organization and other details. The Google Cloud Platform offers services that can be used together with Cloud Dataflow like their cloud machine learning solutions [7].

These two commercial options are clearly separated from the other presented frameworks since they are proprietary products and run in the cloud of their respective providers. Choosing one of these solutions is particularly advantageous if the other services provided in the cloud can be used as well. Overall, using a fully managed product can have a lot of advantages, but also leads to restrictions in customizability and independence. As such, a clear difference has to be made between commercial solutions and the open source frameworks.

5. SPECIAL SYSTEMS

The previously mentioned systems are all primarily used programmatically. Two systems stand out of the plethora of data processing frameworks by providing sophisticated graphical editors. These have the advantage to enable non programmers to interact with the system and increase the accessibility.

5.0.1 Node-RED

Node-RED is a platform used to wire IoT devices and services together [11]. It was originally developed by IBM before being contributed to the *OpenJS Foundation*[11] in 2016. It runs on *Node.js* and provides a visual editor running in the browser. This editor is used to implement a flow-based program, called *flow*, which consists of nodes passing message objects around. Node-RED is a system which fills the gap for small use cases or private users which do not have or require a large computation cluster. A Node-RED instance is light weight enough to run on a Raspberry Pi.

An example flow is provided in Figure 5. The data propagates from the outputs on the right side of the node to inputs on the left side of connected nodes. The *flow* has an HTTP endpoint, which passes the received payload to a parser. The resulting object is then logged to a file and send via email if it passes the filter function.

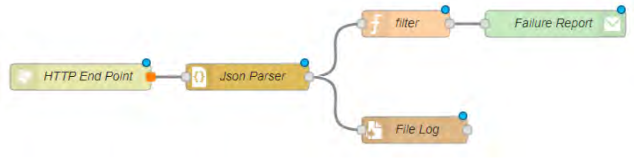


Figure 5: A Node-RED *flow*. It is providing an HTTP-endpoint where health reports can be send to, which are then parsed and logged. If a filter criterion is fulfilled an email alert is send.

Node-RED has a community maintained repository of nodes for a plethora of uses cases and systems which are easily installed with the package manager NPM. This sharing of functionality allows for fast development of prototypes and easy integration into other systems since many required functionalities exist already and do not have to be implemented repeatedly.

[11]Formerly known as *JS Foundation* before it merged with the *Node.js Foundation* in 2019.

A limitation of Node-RED is horizontal scaling for increased performance, since a flow always runs on a single *Node.js* instance. Node-RED does exploit *Worker Threads* to parallelize I/O operations but not for the computation of individual nodes. This implies, there is no mechanism to parallelize the execution of a node in the flow to compensate for inhomogeneous computational complexity. Ways to mitigate this depend on the method the flow digests data from its environment[12]. If it provides, e.g., an HTTP-API, a load-balancer can distribute requests over a cluster of instances, which are all running the same flow. For MQ Telemetry Transport (MQTT)[13], which is a publish/subscribe messaging protocol, each instance could listen on a different topic and incoming messages would be distributed on those topics by the messaging broker. Additionally, Node-RED handles state of nodes only within a single flow. For horizontal scaling, it is required to implement this feature by external means, like a database, which all instances have to write to and read from. In all cases, the architecture of the system has to be adapted to the program that a user wants to run. This is tightly coupling the business logic with cluster design and maintenance. Thus, we do not expect *Node-RED* to be suitable for heavy load data processing. This is backed, by an anecdote of the original developer, OLeary, where he compares using Node-RED to manage a million message per second with crossing the Atlantic in a bathtub. Theoretically possible but not reasonable [13].

5.0.2 Nifi

Apache Nifi[14] is a system, which allows the creation of dataflow graphs [**Sarnovsky.2017**]. They consist of nodes, called *processors*, which consume data from either external sources or a queue within the graph. It provides comprehensive monitoring features including, e.g., the number of queued messages, highlighting for processors that are the bottleneck of a dataflow or the consumed processor time of each node. Nifi is extensible by implementing additional custom processors, which are deployed to the system. Data is propagated through the graph as *Flow Files*. The configuration of processors reaches from simple things like assigning a name up to details like the amount of concurrent threads executing the node, the scheduling time and back pressure thresholds for congestion control. All processors run within a single JVM. Horizontal scaling is realized by spawning multiple instances, which all execute the same tasks but on disparate subsets of data. The consequence of this is, that dataflows of different users are run on the same machine and only separated by rights management. The only way to increase isolation is to run exclusive clusters for each of them.

The focus of the system is to combine process design, control and monitoring within a single application. Thus, it is possible to change configuration in the UI while the processing is running and updating it ad-hoc.

[12]Google Groups discussion on horizontal scalability, including the original Node-RED developer Nicholas O'Leary: https://groups.google.com/forum/#!msg/node-red/Nx1WWqBeLbI/xjZBkWRaAAAJ (visited on 07/28/2019)

[13]http://mqtt.org/ (visited on 11/23/2019)

[14]https://nifi.apache.org

6. COMPARISON

A short and conclusive full-on comparison of the above frameworks would be hardly possible, since the purpose and the implementation of the frameworks differ greatly. With their distinct features, it would only be possible to compare select frameworks in a much smaller context which is not the goal of this paper. Instead we try to guide through the decisions to make when starting with data processing. The information which are presented in the previous sections (Section 3, Section 4, Section 5) are the bases for the following guide. In addition, the key facts about the presented systems are aggregated in Table 1 which can serve as a quick lookup.

6.0.1 Processing Mode

At the very first it must be decided if one needs a system which handles stream data or not. If only batch processing is required most stream processing systems are still sufficient since they have means to serialize the data and then use their native processing mode to consume the data. This goes not apply vice versa. If one has continuously produced data which has to be processed live, this is not natively possible with a batch processing system. The micro batching approach offers a compromise but should be considered carefully since it brings its specific category of problems. For batch processing we would recommend Spark since it is well integrated with other technologies and its streaming extension allows to cover a lot of additional scenarios. For stream processing the most versatile system is Heron with its focus on modularity.

6.0.2 Resource Management

The second decision to make is how resources are managed. If one has dedicated hardware reserved only for data processing purposes no constraints are applied to the framework selection. This is different if the infrastructure has to be shared. Then, some kind of technology is needed to protect different deployed applications from overzealous resource requirements of other running services. It is possible to manually tweak this but we would not recommend to do so but instead use a resource manager like Mesos or one of the many alternatives. This limits the frameworks to Spark, Flink and Heron which all bring resource manager support. If one would strongly prefer another system we want to point out third-party support for, e.g. Mesos, exists at least for Storm and Hadoop MapReduce.

6.0.3 Language Support

Another factor to consider is the need to integrate with legacy code bases or other demands for a specific programming language. The dominating language in the data processing system environment is Java and its related JVM compatible languages. For other languages the system needs either support for writing wrappers or a generic adapter protocol. The all rounders in this category are Hadoop, Spark and Storm which all implement a protocol based on std-in and std-out streams used to integrate arbitrary processes into their computation. The outlier in the presented systems is Node-RED which has native JavaScript support.

6.0.4 Use Cases and Recommendations

We want to briefly describe three use cases and give a recommendation and it's justification.

1) *A small scale private data processing endeavor where we install a couple of dozens of sensors and want to aggregate the information.* We would recommend Node-RED since it can be deployed on a recent *Raspberry Pi*, has an accessible UI and a big library of processing steps. Its limitations in raw computational power can be overcome to some degree by just deploying additional instances (e.g., on additional Raspberry Pis) and deploying a load balancer.

2) *A business level real time data processing project developed by a small team.* We would recommend Storm despite its drawbacks due to the fact that it is easily deployed without dependencies to other systems. In addition it has a stand-alone mode which is interesting in particular during development where a small light weight instance can be spawned on a developers machine for testing.

3) *A new enterprise level data processing project on regularily arriving sets of batch data.* We would recommend Spark due to its integration with other systems like HDFS which is a common backbone for data lakes. Its support for shared infrastructure helps with the integration in existing clusters and does not hinder the deployment of future applications. Furthermore, it supports possible future stream processing scenarios via micro-batching which gives it a great amount of flexibility.

7. CONCLUSIONS

The amount of data processing solutions prohibits to give a complete overview in any reasonable way. In this paper we gave a short introduction into a set of wide spread systems and compiled a list of central criteria to consider if selecting a system for a particular use case. Due to the wide variety of advantages and drawbacks of different systems and the wide variety of possible use cases, from a small home IoT project over scientific machine learning, up to a full scale Industry 4.0 production facility it is not possible to give a general recommendation.

Nonetheless, the comparison should give a first direction when selecting a system with the most important criteria being highlighted. Selecting a framework that supports the right processing mode is clearly crucial, since this is the key aspect of every framework. Making the choice of a stream processing framework for a project working with constant input data of sensors or choosing a batch processing framework when working with strictly static historic collections of data should now be clear. In addition to that, options for smaller-scale projects up to frameworks for large-scale operations have been presented. As an example, Storm allows for a relatively easy setup for single projects, because no large stack has to be necessarily installed. Special frameworks like NiFi and Node-RED have also been discussed, which allow for visual coding instead of complex programming, making data processing very accessible.

In conclusion, this paper provided a general overview of a select list of frameworks, enabling the reader to make the right decision based on the three discussed main criteria: the processing mode, the architecture, and APIs and language support.

References

[1] Safaa Alkatheri, Samah Abbas, et al. “A Comparative Study of Big Data Frameworks”. In: *International*

Category	Hadoop	Spark	Storm	Heron	Flink	NodeRed	Nifi
Processing mode	Batch	Batch & Stream*	Stream	Stream	Batch & Stream	Stream	Stream
Stream processing		micro-batching	native	native	native	native	native
Processing guarantees	exactly once	exactly once	configurable at-least/at-most/exactly once	exactly once	exactly once	N/A	exactly once
Supported Languages	Java & std-io	Java, Python & std-io	Java & std-io	Java	Java, Scala and Python	JavaScript	Java
Supported Languages	Java & std-io	Java, Python & std-io	Java & std-io	Java	Java, Scala and Python	JavaScript	Java

Table 1: Comparison of the presented frameworks according to different feature. (*Streaming via Spark Streaming)

Journal of Computer Science and Information Security, (Jan. 2019), p. 8.

[2] Apache Software Foundation. *Apache Flink Documentation.* Version 1.9. URL: https://ci.apache.org/projects/flink/flink-docs-release-1.9/ (visited on 12/18/2019).

[3] Apache Software Foundation. *Apache Hadoop Documentation.* Version 3.2.1. URL: https://hadoop.apache.org/docs/r3.2.1/ (visited on 12/18/2019).

[4] Paris Carbone, Asterios Katsifodimos, et al. "Apache flink : Stream and batch processing in a single engine". In: *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36.4 (2015). URL: http://sites.computer.org/debull/A15dec/issue1.htm.

[5] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Commun. ACM* 51.1 (Jan. 2008), pp. 107–113. ISSN: 0001-0782. DOI: 10.1145/1327452.1327492.

[6] Rainer Drath and Alexander Horch. "Industrie 4.0: Hit or Hype?" In: *IEEE Industrial Electronics Magazine* 8.2 (2014), pp. 56–58. DOI: 10.1109/MIE.2014.2312079.

[7] Google. *Google Cloud Dataflow Documentation.* URL: https://cloud.google.com/dataflow/docs/ (visited on 12/18/2019).

[8] IBM. *IBM Streams Documentation.* Version 4.3.0. URL: https://www.ibm.com/support/knowledgecenter/SSCRJU_4.3.0/com.ibm.streams.welcome.doc/doc/ibminfospherestreams-introduction-overview.html (visited on 12/18/2019).

[9] Harriet Jarlett. *Breaking data records bit by bit.* 2017. URL: https://home.cern/news/news/computing/breaking-data-records-bit-bit (visited on 11/02/2019).

[10] Sanjeev Kulkarni, Nikunj Bhagat, et al. "Twitter Heron: Stream Processing at Scale". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data - SIGMOD '15.* Ed. by Timos Sellis, Susan B. Davidson, et al. New York, New York, USA: ACM Press, 2015, pp. 239–250. ISBN: 978-1-4503-2758-9. DOI: 10.1145/2723372.2742788.

[11] M. Lekić and G. Gardašević. "IoT Sensor Integration to Node-RED Platform". In: *17th International Symposium INFOTEH-JAHORINA*. INFOTEH. 2018, pp. 1–5. DOI: 10.1109/INFOTEH.2018.8345544.

[12] Hamid Nasiri, Saeed Nasehi, et al. "Evaluation of distributed stream processing frameworks for IoT applications in Smart Cities". In: *Journal of Big Data* 6.1 (2019). ISSN: 2196-1115. DOI: 10.1186/s40537-019-0215-2.

[13] Nicholas O'Leary. *A Tale of Getting Stuff Done when You're 1 in 379,593.* 2016. URL: https://youtu.be/Bbg1017amZs?t=1650 (visited on 11/02/2019).

[14] David Reinsel, John Gantz, et al. *The Digitization of the World from Edge to Core.* 2018. URL: https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf (visited on 11/02/2019).

[15] Ankit Toshniwal, Siddarth Taneja, et al. "Storm @Twitter". In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data.* SIGMOD '14. New York, NY, USA: ACM, 2014, pp. 147–156. ISBN: 978-1-4503-2376-5. DOI: 10.1145/2588555.2595641.

[16] Matei Zaharia, Mosharaf Chowdhury, et al. "Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing". In: *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation.* NSDI'12. Berkeley, CA, USA: USENIX Association, 2012, p. 2. URL: http://dl.acm.org/citation.cfm?id=2228298.2228301.

[17] Matei Zaharia, Tathagata Das, et al. "Discretized Streams: Fault-tolerant Streaming Computation at Scale". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles.* SOSP '13. New York, NY, USA: ACM, 2013, pp. 423–438. ISBN: 978-1-4503-2388-8. DOI: 10.1145/2517349.2522737.

How Resource Management of Kubernetes, Yarn and Mesos Affects Different Batch Job Workloads

Bernd Schoolmann
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
bernd.schoolmann@rwth-aachen.de

Johannes Leurs
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
johannes.leurs@rwth-aachen.de

ABSTRACT

In the last years, the containerization of applications has played a leading role in cloud computing because of resource efficiency and scalability benefits. To assign workloads to hardware in compute clusters there exist resource management frameworks, such as Kubernetes, Yarn, and Mesos. However, when deciding which of them to employ, there is no overview which compares them directly and evaluates strengths and weaknesses. To help with a consultation, this paper compares Kubernetes, Yarn, and Mesos with respect to batch job processing. First, a general overview of the architectures is provided, with respect to scheduling, storage, networking, resilience, and control interface. After that their features are compared. An in-depth comparison of the three frameworks follows in the aspects of data locality, inter container communication, resilience, set-up overhead, and specific resource requirements. Finally, there is a comparison in terms of scalability. We conclude that it is not a trivial choice, which resource negotiator to employ, but it depends on several requirements which the user places on the cluster: While Kubernetes offers simple operation for less complex workloads, Yarn supports easy mechanisms to achieve data locality and Mesos is the most flexible one when complex requirements are placed on resources.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.2.9 [**Software Engineering**]: Management—*productivity, programming teams, software configuration management*

Keywords

Kubernetes, Hadoop, Yarn, Mesos, Data Processing, Batch Processing

1. INTRODUCTION

In an increasingly data-driven world, mass parallel processing of data on compute clusters becomes more and more relevant. Because clusters consist of many compute nodes, assigning programs to compute resources is usually managed by a central instance, a resource manager. Some examples for these managers include Kubernetes, Mesos, and Yarn.

Kubernetes is developed mainly by Google and profits from years of experience that Google developers gained when using their proprietary solutions Borg and Omega, which they developed for Google internal resource management[13]. Yarn is the resource manager of the Apache Hadoop ecosystem. Hadoop was first developed for executing MapReduce jobs on compute clusters, but because of the lack of support for other workload types, Hadoop 2.0 introduces Yarn to manage arbitrary workloads. Mesos is also a project of the Apache Software Foundation. Mesos features powerful control for clients of deciding which resources their programs should run on.

Resource allocation that serves the needs for all possible workloads, like single applications and complex sets of collaborating applications, is a non-trivial task, as modifying the resource allocation architecture to increase support for some workload types may make resource allocation less flexible regarding other workload types.

Since workloads have various runtime requirements, a solution to this must take many constraints into account. Handling this and yet keeping a small architecture is a challenge, however, it is beneficial to scalability [24].

The aforementioned frameworks approach this challenge in different ways. Therefore it is interesting to ask which of these frameworks to employ, in order to serve application needs such as low latency communication, data locality, resilience, and scalability. We answer this question by comparing Mesos, Kubernetes and Yarn. We chose these because they are used by big companies [29, 11, 19] and are open source, showing that they are relevant for many users. After obtaining an overview of the frameworks architectures we analyze how good they serve the aforementioned workloads' needs. Because many containerized workloads scale easily, we finally analyze how they scale in Mesos, Yarn, and Kubernetes.

Regarding the examined workloads we restrict ourselves to batch job processing, even though all resource managers also support other types of jobs, like services. Our definition of batch job processing is a data processing program, that can be distributed into program fragments running in parallel,

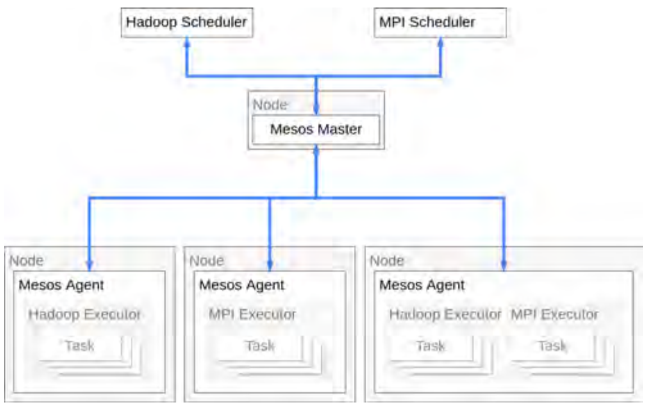


Figure 1: Cluster Architecture of Mesos

that we call tasks. Batch jobs do not interact with other users or systems once started. This also applies to data needed for the programs: it has to be available on startup. No interaction with anything also means that no real-time requirements can be imposed such as responding to external events. In conclusion, batch jobs just run to completion on a given data set, if not interrupted.

In section 2 we describe each resource management framework's architectures and afterwards compare their unique properties. In section 3 we investigate different requirements of batch jobs and how the different resource management frameworks aid in fulfilling these. In section 4 we compare the clusters in terms of scalability of both workloads and the underlying infrastructure.

2. FRAMEWORK ARCHITECTURES

In this section the architectures of compute clusters - as understood by the frameworks and the interaction with them - are described, taking into account: what parts they consist of, what to do to deploy jobs on the cluster, what cluster users can do to assure that their jobs are being scheduled in a way that matches specific conditions (e.g. on specific types of nodes) and what additional features they provide that are useful for batch processing. Additionally we examine how resilient they are.

2.1 Mesos

Mesos' approach to resource management is lightweight. At the core Mesos only bundles information about what resources in a cluster are available and deploys workloads on available resources when instructed to do so by cluster users. It does not make decisions where to schedule these workloads. This is implemented by frameworks that interact with Mesos [25].

2.1.1 Scheduling

The low-level layer of Mesos consists of *agents* and the *Mesos master*. The Mesos master manages the resource distribution by publishing resource-offers to frameworks they can use to schedule their work on. The agents are daemons that run on every node to monitor tasks and resource consumption. They regularly pass this information to the master, so that it knows what resources are available for new programs.

The high-level layer of Mesos consists of a set of framework schedulers, which users implement to interact with the Mesos cluster. They subscribe to the master to obtain resource-offers, in order to schedule tasks and control their execution on the cluster. These resource-offers consist of unused resources of the cluster. The framework schedulers can then decide by themselves, if they accept them (or parts of them) to run their jobs on, or decline the offers and wait for better suiting ones. The philosophy behind this is that applications can make better decisions on what resources satisfy their needs than an external scheduler. However, the master can withhold resources, e.g. to ensure that a single framework's workloads do not block the entire cluster.

Which resources are offered to who is controlled by an allocation module in the Mesos master. This allocation module can be exchanged by custom ones to match organizational requirements on resource distribution [6]. When implementing a custom allocator module in the Mesos master, it is also possible for schedulers to send requests to the Mesos master that inform him more specifically about what resource requirements they have. The default allocator will ignore these messages though [10].

When a framework's scheduler accepts resource-offers, it sends tasks to the Mesos master to deploy them. Resources might also be reserved by them for later use. On the node, the scheduler chose the agent then starts a container in which he launches an *executor* which is provided by the framework. The executor sets up and manages the execution environment (including networking stack, storage and setting up environment paths) for the tasks which it launches in their own containers (nested in the executor) and is in charge of monitoring the execution of jobs for the framework. That includes, e.g. checks for malfunctioning containers. The communication between jobs and frameworks is done over a communication endpoint provided by the executor. Because the requirements of different jobs can differ strongly, frameworks are expected to implement their own executor to take responsibility appropriately. However, it is also possible to use a default implementation that simply launches containers. Mesos offers a mechanism called *task groups* that can be used to share resources such as network stacks and storage among tasks. Task groups can also be used to ensure that a group of tasks is launched on the same node. The Executor can launch containers nested inside other containers running on them. The nested containers then adopt their parents' resources. When cluster operators want to ensure that specific containers run on certain nodes, they can launch these as *standalone containers*. No frameworks or executors are needed for that and the Mesos master is bypassed, so operators do not need to rely on resource-offers.

Mesos allows adding information about nodes to refine scheduling decisions. Both types, *attributes* and *resources*, are represented as key-value-pairs. Resources describe information about compute resources that the Mesos master considers in his resource-offers and partitions when needed. CPU and RAM information is provided this way, but e.g. disk space or GPU information might also be considered. Attributes are information that framework schedulers might interpret but are ignored by the Mesos master. Examples might be hardware products used which the schedulers can consider for optimization. Custom information can be added in both ways. Mesos supports two container technologies: While Docker containers [22] are supported, Mesos also provides its own container technology which provides its own resource isolators, customized to meet the requirements of

Mesos.

2.1.2 Storage

Mesos treats storage as a resource and therefore the master is responsible for distributing storage included in resource-offers. When a framework's scheduler accepts an offer, it mounts a volume created on that storage directly into the started container. For that, storage must be on the same node as the launched task. The agent then enforces the resource limitations. Frameworks can grow and shrink the volume at any time if further storage is offered to them.

In addition Mesos supports the Container Storage Interface (CSI) [16]. Thus Mesos can integrate third-party storage solutions by interacting with a plugin written by storage providers. The storage is then handled as native storage. At the moment, only local storage is supported. In the future, external persistent storage will be supported by Mesos using CSI [7]. Another way is to use Docker containers that support external storage. This storage can not be controlled by the Mesos master.

2.1.3 Networking

Mesos supports the Container Networking Interface (CNI) [15], which provides flexibility when designing a job's networking architecture. Using CNI, it is possible to assign one IP address (or multiple) to each container. The connection between containers is managed by the CNI, so tasks do not need to take this into account. Features include enforcing a network-wide security policy by separating containers into virtual subnets or inspecting the contents of the packets, but also determining the fastest routes between hosts, ensuring low latency and high bandwidth. The CNI can extend the created virtual network across clusters that are only connected over the internet. Other features, such as traffic shaping, to enforce bandwidth limits and guarantees per application, are also possible. A drawback in the current implementation is that Docker containers in user mode only support one network interface at a time. This is not a problem with Mesos containers.

Mesos provides a concept of regioning called ”Fault Domains” which allows specifying a region and a zone a node belongs to (the name is ”Fault Domain” because different nodes are more likely to fail together when they are nearer in the network topology, e.g. if their common rack fails). While it is proposed that the zone specifies the rack the node is in and the region specifies the cluster, this assignment is not mandatory, because the cluster administrator determines it. This information can be used for scheduling decisions.

2.1.4 Resilience

Mesos offers a high-availability mode for the Mesos master in which there are several masters as a backup for a leading master. In case of a framework scheduler failing, Mesos allows the frameworks to run multiple schedulers with only one interacting actively with the Mesos master [8].

The Mesos master monitors its connection to agents using ping messages and monitoring the state of the TCP connection to the agent daemons. If the ping health check fails, the master considers the agents dead and removes them from the cluster. Frameworks are notified about this so they can react to the loss of the tasks that ran on the cluster. If the connection between the master and the agent is reestablished after that, the master prompts the agent to restart. The agent then restarts and re-registers at the master. Tasks are lost, but persistent volumes survive. A TCP connection failure is handled the same way, except if running frameworks checkpoint the state of their executors. Then the master allows the agent to reconnect until a timeout is reached [8]. If a node gets separated from the Master (meaning that it can no longer be reached), the Mesos master assumes all of the tasks on it as dead and reports this incident to the application frameworks. Mesos also informs schedulers about crashed executors.

If the agent daemon crashes or exits and is restarted, the new instance may take over the responsibility of running executors and tasks, if the agent has checkpointed its state before. If not, running tasks and executors are expected to gracefully exit. However natively, agents are not restarted automatically. For this an external service has to be used.

Mesos delivers a health checking mechanism to monitor if tasks run as expected. Regarding application health, Mesos supports checks on tasks performed by local executors so they do not burden the network. These can check HTTP endpoints or TCP availability, or be arbitrary shell commands. The results of the checks can be forwarded to the scheduler if configured conditions hold. Mesos distinguishes between ”normal” checks and health- checks. While the information collected using normal checks is just forwarded to schedulers, only the interpretations
”healthy” or ”unhealthy” are forwarded by health checks. In addition, executors might kill tasks that consequently fail health checks.

2.1.5 Cluster-Control API

Frameworks interact with the Mesos master over a REST API, although there exist libraries for Java and C++, too. It is also possible for a framework to communicate with its executors. For that, a REST call is provided in the master's REST API, which forwards messages to the framework and to the executors.

Administrators can manipulate storage volumes, set resource reservations, changing maintenance schedules or configure resource providers (that are used in CSI). Moreover, information about the cluster state and its components can be retrieved.

It has to be noted, that in general, messages between framework components are not transmitted reliably. E.g., the request to schedule a job on offered resources can be lost on the way from framework to master. Thus, communication partners should monitor responses until a timeout, on which they take appropriate action. An exception is status updates about tasks or operations that the agents, executors, and master send. They include important information about the task state (e.g. failed/pending/finished), their health and failure, including the reason, or –if requested– about the state of operations invoked. Thus frameworks can track job progress and react to critical events, such as failed tasks. To make the transmission of status updates reliable, schedulers must send an acknowledgment of these messages.

2.2 Kubernetes

Kubernetes was developed with a focus on running web services. Thus a lot of components to support web services, like load balancers, already exist. However, it is also possible to run batch jobs on Kubernetes. defew A Kubernetes cluster can be configured by setting a *desired state* of the cluster. Configurable options are which jobs and how many instances of them should run, which limits apply to the jobs and more. Kubernetes always tries to bring the *current state* of the cluster to the desired state by applying certain actions, like replacing malfunctioning containers with new ones. In order to deploy workloads, the user modifies the desired state as a goal for Kubernetes to move the cluster towards.

2.2.1 Scheduling

In Kubernetes, the smallest deployable unit is called a *pod*. A pod is a group of at least one container. All of these containers run on the same physical node and share a network stack. This is useful if there are multiple tasks that should be run in different containers, but are still tightly coupled and need high bandwidth inter-communication.

A Kubernetes cluster consists of a control plane, managing the state of the cluster, and nodes, responsible for executing the tasks in the cluster. The control plane (also called master) is responsible for changing the current cluster state to the desired state. It consists of several components: The *API-server* handles queries about the clusters state and requests to change the desired state. For example, the command line tool kubectl communicates with the API-server to manage the cluster and the jobs running on it. The *scheduler* then takes action to bring the cluster's current state to the desired state, like assigning unscheduled jobs to nodes. In Kubernetes the scheduler is exchangeable with ones that are more appropriate for specific workloads. Multiple schedulers at the same time are also supported. If multiple schedulers are running, it is possible to select which scheduler to delegate the assignment of a pod to. *Kube-scheduler* is the default scheduler in Kubernetes, capable of filtering and ranking nodes for a job. This is usually done using a labeling mechanism: *Lables*, which are key-value pairs, can be attached to nodes to add arbitrary information about them, for example, that an SSD is used. This information can then be utilized to specify placement constraints for a pod, which are used to filter feasible nodes. User-defined constraints are optional. Default ones – like verifying that nodes' resources are sufficient to fit the pod – are always checked. After filtering, the scheduler scores the remaining nodes in a similar way where labels can again be used to set preferences towards nodes with certain properties. The pod is then assigned to the node with the highest score. Kubernetes supports all container technologies that implement the Container Runtime Interface (CRI). Besides that, Docker and rkt [21] are supported without CRI.

2.2.2 Storage

As Mesos, Kubernetes supports the Container Storage Interface, which can be used to provide persistent storage to a container. There are local plugins, which keep the data on the node, providing high speed access and a high amount of storage operations per second, compared to the alternative: remote plugins, which decouple the storage layer from the compute infrastructure. A CSI in Kubernetes consists of a node plugin and a controller plugin. The controller plugin handles general requests like creating or deleting a volume. The node plugin handles the operations interacting directly with a volume like reading and writing data.

2.2.3 Networking

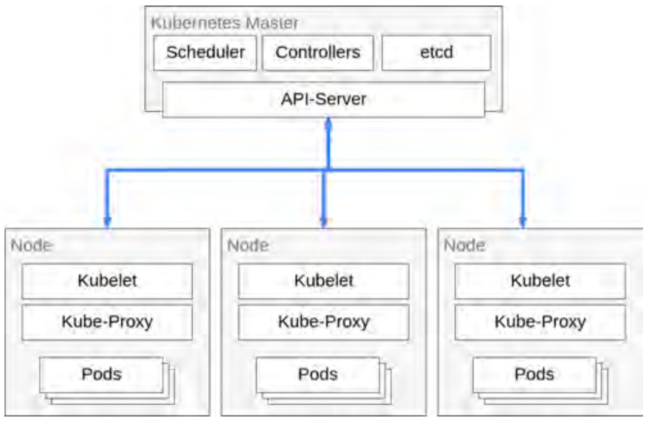


Figure 2: Cluster Architecture of Kubernetes

As Mesos, Kubernetes supports the CNI. There is however a wider variety of CNIs available for Kubernetes. In Kubernetes, the CNIs integrate very well with Kubernetes' annotation system. For example, a limit can be set for ingress and egress bandwidth for a pod just by applying an annotation in the pod's specification.

Kubernetes provides two mechanisms for service discovery: DNS and environment variables. A service, in Kubernetes, hides the networking endpoints of specific pods and provides a single IP address to communicate with if a specific type of service is desired. To find a service, the pod can request the IP-addresses of the domain *service.namespace*. Using this relatively simple mechanism, different programs can find each other and communicate.

2.2.4 Resilience

In the control plane, there is a set of controller managers. They run different controllers including the ones responsible for maintaining the correct number of pods, the ones responsible for reacting to failure events and other ones.

Pod failures are detected by health checks and probes, running on the nodes. If a pod fails them, its state will become failed. The restart policy is an object in Kubernetes which determines, if and when a pod is restarted or rescheduled. The scheduler then responds according to the pod's restart policy. The latter might be useful if a restart might cause data loss because of an inconsistent starting state left behind by the first run [20].

If a node fails, it might be drained by a cluster administrator or controller running in the cluster. Draining it evicts all pods from the node and makes it unavailable for new pods. The scheduler then sees that the desired state contains the pods that are evicted, but the current state does not, so it reschedules the missing pods [17].

The Kubernetes control plane can be configured to be in high availability mode. By default, there is only one master, whereas in high availability mode there is a set of masters that all work in parallel. In high availability mode, the cluster's state is held in a distributed etcd cluster (a key-value store). In a high availability cluster, both the master applications and the etcd instances holding the control plane's state should be distributed across different physical machines. In the event one master fails, operation should continue as normal since the state is still available on the other etcd instances. A cluster operator should make sure to remove failing masters and add new ones if the number of healthy masters is too low [18]. Without high availability, the loss of etcd would mean the entire cluster state is lost and needs to be rebuilt. If only an application master fails, it can be restarted and no loss of data or state should occur.

2.2.5 *Cluster-Control API*

Application developers communicate with the cluster using a REST API. Usually, this is wrapped by the command line tool kubectl.

If configured, jobs can communicate with the master over kubectl or the REST API in the same way as other cluster users. Using this API the desired cluster state can be set, jobs can be added, updated, or deleted and even nodes can be edited.

2.3 Yarn

The Hadoop framework is the oldest of the resource management frameworks we examine in this paper. It natively supports complex distributing data processing workloads like MapReduce by employing the Hadoop Distributed File System (HDFS) in order to accelerate data loading before starting computations. The initial version of Hadoop had the downside that resource management and execution control of running jobs were tightly coupled into the same component (the *job tracker*), which hindered scaling since these components could not be scaled independently. Execution control did not scale with the workloads at all because of the monolithic structure of the job tracker. Yarn attempts to fix this by splitting resource management and execution control into separate daemons [30].

2.3.1 *Scheduling*

On a Yarn cluster, users manually and frameworks automatically submit jobs to the *resource manager*. The resource manager then deploys an *application master* for the job on a node in the cluster. The application master is an application written by the user to manage the execution of jobs. For that, the application master negotiates the resources needed for the job's execution with the resource manager which also is the central broker between application masters and the nodes regarding resource allocation. On each node, there is a daemon called *node manager*. Once the application master is granted resources, it communicates with the node managers resources' nodes in order to deploy the tasks belonging to the job. The application master then monitors the execution and watches for failures of the tasks.

Node attributes can be used to specify the properties of these nodes. They can be used by application masters to specify constraints on resources they request. The resource manager will then attempt to fulfill these constraints. Another type of constraints are *affinity constraints*. These can be used to evaluate which placement is better suited for tasks compared to others. *Anti affinity constraints* do the opposite.

Yarn natively supports Java applications as containers. In addition, Docker containers are supported.

2.3.2 *Storage*

Yarn integrates natively with the distributed file system of Hadoop, HDFS. HDFS spreads the blocks, which the files consist of, over multiple nodes called *data nodes*. These are

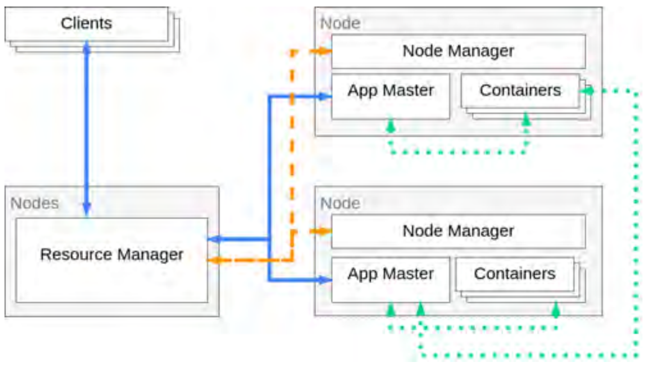


Figure 3: Cluster Architecture of Yarn

usually the same ones that are used for computations. One master component, the *name node*, indexes the locations of the blocks. When attempting to access a file, a request is sent to the name node which replies with the locations of each data block the file consists of. The requester can then communicate directly with the data nodes which hold these blocks. In this setup, no data blocks are routed through the name node, because this would be a bottleneck for the system and overload the name node. HDFS is built for redundancy and therefore replicates blocks across several data nodes.

A few other storage providers can also be used, such as Amazons S3, OpenStack Swift and Azure Blob storage.

2.3.3 *Networking*

By default, there is no network abstraction layer that can provide a virtual network between containers. However, the application master has open communication channels to the worker containers. This means that all containers on one node share the same IP address (except when using Docker containers) and that applications need to take care of the networking layer themselves. A service discovery mechanism can be used to resolve DNS records to containers. Using a service record, the hosts IP and port to the container can be stored and retrieved by other services later on [4].

2.3.4 *Resilience*

The state of the Yarn cluster is kept in a Zookeeper cluster. During a fail of the resource manager, cluster features such as scheduling are unavailable but while restarting the resource manager the cluster state can be recovered. Failover resource managers might be deployed, so that downtime of the cluster's features is prevented by switching traffic to a failover resource manager in the event that the main resource manager fails. Having multiple active resource managers at the same time is currently not supported.

In the event of a node failure, the resource manager updates its list of available nodes. The application masters implementation determines whether it can recover the job and the missing tasks or not. The developer has to implement data recovery and task restarting logic for the application master. Otherwise, the whole job might fail from a node outage or maintenance.

If an application master fails, the resource manager detects this and reschedules another instance of the application master. Recovering from this failure is left up to the application master, which means that the developer has to implement the logic to recover tasks belonging to the application master and recovering into a consistent state [12]. If a container fails, it simply gets restarted by the node manager of the node the container was running on.

2.3.5 Cluster-Control API

For interaction with the cluster, a REST API can be used. Additionally, a command line tool exists for abstraction. It supports job deployment, querying information about the cluster, deleting, updating or creating jobs, but also cluster management commands like getting logs and inspecting states of the nodes.

If the job submitter wants to communicate with the application master, he can request information about it (and the application in general) from the resource manager. Among other information, the address for remote procedure calls is included that can be used for communication [3]. A service discovery mechanism can be used to resolve socket information of containers from URLs. If only tracking of application progress is required, the application master can log information about that (or arbitrary other information) to a server called *timeline server* from which the information may be retrieved by job submitters.

2.4 Feature Comparison

The previous description of the cluster architectures shows a few of key differences between the paradigms which the frameworks follow.

As mentioned, Kubernetes always has a desired state and a current state and the controllers try to transform the current to the desired state. For example, when scheduling 10 pods, with none prior running, Kubernetes sets the desired state to 10 running pods. The scheduler sees this and tries to assign nodes to these pods. Yarn and Mesos, on the other hand, take actions in response to events. In Yarn, the user tells the resource manager to launch a master, which then negotiates resources with the resource manager to launch the containers on. In Mesos, the user tells the framework to launch the containers. Kubernetes takes more effort when it comes to handling unexpected events like malfunctioning containers, e.g. by replacing them with new ones. This takes off work of the frameworks but limits them in their reaction to these events on the other hand. This behavior can be augmented so that frameworks can react appropriately regarding the situation. Using Mesos and Yarn, the frameworks always have to handle such situations themselves. Another difference is that Yarn operates on a less abstracted networking layer compared to Kubernetes and Mesos. While Kubernetes and Mesos provide integration for the CNI and thus can provide IP addresses to each container and can create common virtual networks across different clusters, containers in Yarn share the same address spaces if they run on the same machine and thus are not decoupled from the infrastructure (the node's network stack). One more difference is resource allocation. Yarn and Kubernetes by default both decide which resource a Job is assigned to. Mesos only ever publishes out resource-offers, but the final decision which of these offered resources to run the job on lies with the framework. This has massive implications for flexibility (and scalability). Yarn's and Kubernetes' schedulers are more complex since they need to handle more logic. Mesos' approach is more lightweight since it only acts as a broker between advertised resources of the nodes and requests from frameworks. Additionally, Mesos' approach is more flexible since it allows to easily integrate a custom resource manager in Mesos in the form of a Mesos framework. Replacing the scheduler in Yarn or Kubernetes requires a lot more integration than simply accepting or refusing offers made by Mesos.

Finally, there is a big difference in how jobs are run from a conceptual point of view. Kubernetes does not really recognize jobs that span over multiple pods. It would be possible to imitate a Yarn-like application master with a pod that communicates with the cluster handle the scheduling of pods. Mesos is similar in this regard, however, the framework might be customized to run specific types of jobs in a more general way. For example, a framework can run all MapReduce tasks instead of reimplementing it for specific MapReduce tasks. Yarn is different in this matter. Yarn always requires an application master which knows how to execute, manage and monitor the execution of a more complex job. This is useful when anything interdependent like a MapReduce job is deployed, which comes with a lot of intercommunicating worker containers. Even for simpler jobs, like a build pipeline, this might be necessary in order to ensure the correct order of execution of these steps.

3. HANDLING OF BATCH JOB REQUIREMENTS

Different kinds of jobs benefit in distinct ways from each resource management framework. In this chapter, we investigate different relevant properties, and the effect the resource management frameworks have on each of these properties.

3.1 Data Locality

Data locality describes the property that the data needed for an application is already on the node at application launch (and does not need to be loaded from another source), or at least should be close like in the same rack [23], since bandwidth between nodes is higher. Thus it is sometimes more efficient to move application code to the machine that stores the data needed by it and run it there than vice versa. That is e.g. the case for MapReduce Jobs, where many tiny instances of an application have to run on large data sets where transferring parts of the data through the network would slow down the execution flow, as the bandwidth is the bottleneck. An example application for this job would be doing statistics over user data, like computing the average age of all users. The opposite case, where there is a lot of computation in a workload on a relatively (to the amount of computation) small dataset, does not benefit as much from data locality, as the nodes cannot keep up with the hard drive speed, or even the network speed. An example of this kind of application is machine learning, where a lot of computation is being done. Jobs requiring data locality benefit from HDFS which comes with Yarn included in Hadoop. By communicating with the name node the jobs application master can locate the machines and racks where their data is stored on and request the tasks to run on these machines (or at least on the same racks as the machines). Since this eliminates most of the network bottlenecks present when compared to a separated file system layer, Yarn has an advantage — in workloads benefiting from data locality — as long as the application developer implements interaction with HDFS properly. Mesos and Kubernetes do not include a distributed file system like HDFS natively, however, it is possible to mount local volumes on the nodes. To achieve similar support of data local applications, it is conceivable to implement a scheduler on Mesos, which imitates the features of HDFS. Data nodes could be implemented by using standalone containers to ensure that one instance runs on each node. In Kubernetes, one approach would be first deploying an application (similar to Yarn's application master) which communicates with a name node and then sets the filters/ranking of nodes based on this. However, this is a considerable effort. Both approaches amount to re-implementing an HDFS like file system.

To summarize, data locality is natively supported by Yarn. In Mesos and Kubernetes it is also possible but requires more effort for the application developer while not contributing any benefits compared to Yarn. If data locality is an important property, Yarn is the simplest and most effective solution.

3.2 Inter Container Communication

Communication intensive applications, such as those which use message-passing interfaces, might require a network infrastructure with high bandwidth and low latency in between the compute nodes. At first, this sounds similar to data– locality. However, it tries to achieve higher data-processing rates by keeping related applications close to the data they require. However, in this section, we do not consider data that is stored ahead of time (like in the data locality section), but instead, we consider the live communication between processes. For example, a distributed physics simulation benefits from low latency communication since there are many interdependent data relations that need to be considered. If this simulation is detailed enough (and thus is exchanged information) it will also require high bandwidth since in such a simulation, each time step depends on the results of the previous one. Thus the tasks the simulation consists of benefit from being scheduled near to each other regarding the network topology.

Since Mesos' scheduler gets offers it can decide whether the constellation of allocated resources fit its requirements (in terms of inter resource communication) and then decide to accept the offer. This is very flexible since the region the tasks are scheduled into is dynamically determined. Region information might also be given addressing nodes with attributes that indicate their place in the network topology. This is a more refined approach than using fault domains because fault domains only offer a two-level hierarchy. In Kubernetes, this is handled through Kubernetes' labeling system. A region and zone can be specified. Similar to Mesos, these get attached as labels to the node. A job could be specified to run specifically in a region to improve the communication between the tasks. This, however, is less flexible than Mesos since the region needs to be specified by the cluster's user when deploying the job, instead of being dynamically determined by the scheduler, as there is no offer system. As Yarn is natively rack-aware, the scheduler can be configured to schedule several tasks into the same rack for improved communication. In order for this to work, the Yarn cluster operator has to configure the datacenter's networking architecture into the cluster. The cluster user can specify a delay the scheduler can wait before loosening up its locality requirements. According to Tom White [31], in a typical Yarn cluster, waiting only a few of seconds significantly increases the chances to get scheduled on the same node as another job. If lots of data transfer is involved, the time savings in the data transfers already significantly outweigh this slight delay [2].

In Yarn and Kubernetes a common region to schedule jobs on can only be specified by explicitly naming a specific one for scheduling. The disadvantage is that requesting frameworks can not know what capacity is still free in that region, so the request could lead to a resource allocation that is not optimal: The instances requesting the resources would have to try all possible regions after each other to find a fitting one until the tasks can be assigned. Mesos, on the other hand, allows a more flexible approach. Because frameworks can decide by themselves what resources to use, they can decide on the best set of resources matching their regional constraints that the Mesos master offers them (or wait for an even better fitting one).

3.3 Resilience

With increasing cluster size the likelihood of a machine failing increases. If an application runs a long time (e.g. over weeks) and its assigned node fails, the effort to run the application has been of no use if no measures have been taken to recover from such a solution [2].

In this section, we are going to investigate three types of failures: application failure, node failure and master failure.

The first type of failure to consider is application failure. When a bug in the application causes a task to fail, it has to be detected and restarted. Mesos and Kubernetes offer built-in mechanisms to provide this service in the form of health checks. The cluster user can define health probes that check status provided by the tasks, for example over an HTTP endpoint which reports whether or not the task is healthy. Yarn has no such integration. In Kubernetes, once a task and its pod stop, its restart policy determines if it is automatically restarted by the scheduler or not. By default, the scheduler will automatically restart the pod. However, if it is required that these events are handled manually, handles can be attached to these events in order to restart tasks using custom procedures. Mesos will notify the framework of the task failure. It is the framework's job to decide if and how to reschedule. This allows more flexible recovery mechanisms. For example, if the cluster notices that failures often occur in one region due to disk faults, the framework can decide to reschedule in a different region since the previous region's storage infrastructure is down. In Yarn, the application master has to check the health of its tasks and has to deal with the restart logic itself.

To summarize Kubernetes provides the simplest tools for dealing with task failure. All tools provide an option to do smart rescheduling by attaching custom logic to the failures. The application state recovery has to be implemented by the developers and is not handled by the clusters in all cases.

The next type of failure to consider is node outage. Sometimes the worker node is down for one of several reasons. Some of these reasons are hardware failure, maintenance downtime, software crash (in the operating system), network failure, or other failures. In a big cluster, over a long-running job, this is likely to happen. Therefore it would be useful if the cluster automatically detects such a failure of a node and restarts the tasks that are now missing from the cluster. In Kubernetes, a failed node needs to be drained (logically from the cluster's point of view, does not necessarily involve communication with the node) which evicts all the pods from it. As long as the restart policy permits it, the scheduler will automatically reschedule all the pods that were lost. If local storage is used, it is lost. Yarn handles this like a container failure of all containers on the node and notifies the application master. If the node of the application master is affected, the application master is restarted by Yarn (if this is configured). If its state should be recovered, the application master has to implement the logic for this itself. In Mesos, similar to Yarn, a node outage is broadcasted to the affected frameworks if the Mesos master discovers it. The frameworks have to handle rescheduling of all the tasks which have been lost due to the node failure themselves. However, if the frameworks implement checkpointing mechanisms for their tasks, they might just be restarted and their state recovered. It follows that for node failure, Kubernetes follows a very simple concept while Mesos and Yarn by default enable complex logic to handle failures. This is similar to the difference between the frameworks in terms of handling application failures.

The final type of failure to consider is a master outage. Kubernetes has several masters active at a time and keeps the state in a shared store (etcd). The failure of one master does not pose a significant problem since there are more masters running. If only one master is running in the cluster, it should still be able to recover from failure as long as the cluster state (the etcd cluster) is replicated onto other machines. Yarn keeps the cluster's data in a shared distributed store which means in single Master mode it works the same way as Kubernetes. It follows that there can be small outages while the new Master is being set up (by the cluster operator) however no data should be lost. Availability is possible due to the option to add failover resource managers. Similarly, Mesos can have multiple failover masters with one only being active at a time. It also employs a distributed state store just like Kubernetes and Yarn, in order to ensure that it does not lose the cluster's state.

To summarize, Kubernetes, Yarn, and Mesos all provide master availability and do not lose state due to distributed state stores, provided that they are deployed in high- availability mode. They differ slightly in the implementation of failover mechanisms, but in our opinion not enough to matter for a lot of real-world applications. Outages are possible if the frameworks do not have enough master nodes to failover to, however, it should be possible to recover from this since all three frameworks keep the cluster data in a distributed data store.

3.4 Job Deployment

A simple job like a one-off batch job, for example converting a set of video files, can be deployed by just one command with Kubernetes. All a developer has to do is to package the application in a container, upload it to a registry the cluster has access to and then tell the cluster to start the job. In Yarn, the developer needs to write at least the application master which the resource manager deploys onto the node to manage the job's execution. This application master needs to implement the communication logic with the resource manager necessary for deploying the task onto the cluster. In Mesos, a default executor can be used to deploy one-off jobs. However, a framework has to be written that handles at least resource negotiation with the Mesos master. For jobs with more complex requirements, Yarn offers tools to achieve good data locality, while no native tools are available in Mesos and Kubernetes. Regarding complex constraints on resources, which jobs might have, Mesos has the most flexible approach to take this requirement into consideration. Due to their network abstraction layers, Kubernetes and Mesos make deployment of networked jobs easier, since this decouples the underlying infrastructure from the application code. It follows that Kubernetes has the least setup overhead for simple applications. Yarn and Mesos need a complex setup even for trivial tasks which makes them less usable for budget or time-constrained projects but provide features to support more complex job requirements.

3.5 Specific Resource Requirements

When running a job, certain resources or hardware features might be needed to complete the job more efficiently, or even to complete the job at all. Examples include A GPU, special-purpose hardware or certain hardware features like specific hardware-accelerated CPU instructions. For example, machine learning tasks benefit heavily from graphics processing units or even tensor processing units. Therefore it is useful to schedule certain jobs specifically only on nodes with this special hardware, in case not all nodes share the same hardware. Using Kubernetes' labeling system, the cluster operator can advertise these features for nodes by labeling them with the appropriate properties. The cluster user can then specify this label as a filter on the scheduling request and the scheduler will ensure the task has the necessary resources. For GPUs, a device plugin for the GPU kind (currently Nvidia or AMD) has to be installed in the cluster. The user can then specify the amount of GPUs the job requires, and the job will get exclusive access to the GPU. Kubernetes does not support sharing a single GPU over multiple jobs at the moment. Mesos provides a similar labeling system. Depending on label type (attribute or resource) Mesos ignores this information, so they are only interpreted by schedulers or distributes them within its resource-offers [9]. This means that the framework schedulers need to support filtering the job requirements, in order to take advantage of them. The cluster user can then limit themself to only accepting offers that match their specific requirements. For GPUs, the cluster user specifies this as a special resource to schedule the job. At the moment, Mesos only supports Nvidia GPUs which might pose a problem if the cluster operator already possesses hardware from a different manufacturer. Mesos supports sharing the graphics memory of GPUs which means that when changing between jobs the data set does not need to be re-uploaded. There is, however, no way to enforce these limits at the moment, they require the jobs to implement the limits. Yarn can handle specific resources over a similar labeling system, however, this needs to be specifically enabled in the resource manager. With this enabled in the cluster, a node-label expression can be specified, and the job will only be placed on nodes that match this expression. For GPUs, this is not necessary as they are supported by default. However, the same restriction as with Mesos, supporting only Nvidia GPUs, does apply. Yarn does not support sharing GPUs at the moment, but it is planned for the future.

To summarize, all frameworks support scheduling to special hardware through a labeling system. GPU support differs a bit between frameworks however. Kubernetes has a wider support of GPU suppliers. Mesos on the other hand allows to distribute GPUs and other special resources on the same node across different frameworks by considering it in its resource-offers.

4. APPLICATION AND INFRASTRUCTURE SCALING

Scaling is an important challenge when it comes to the evolution of an application. This is also true for batch jobs: Be it a physics simulation that becomes more complex, or more user data that has to be processed, e.g. for statistics. In the context of container technologies, this often means deploying more containers because of the anatomy of mass parallel processes that run in compute clusters. Thus, it is interesting, how the three frameworks handle the increase of work to schedule, both in terms of scaling up the number of jobs running on the cluster and of adding infrastructure to the cluster.

4.1 Workload Scaling

Kubernetes' main bottleneck when scaling workloads to tens of thousands of pods is the scheduler. It is the single point that all scheduling passes through. If a job is to be scheduled, it gets added to the scheduler's queue and waits to be scheduled. This means that when scheduling a lot of pods, the queue needs to be big enough to contain all pods, and the scheduler needs to be fast enough to handle the jobs sequentially. At the moment, at most 150,000 pods are supported and 30,000 pods can already take 2 hours to schedule [26]. An improvement on this is a custom scheduler or deploying multiple schedulers to schedule pods in parallel.

Yarn has a similar issue, as here the resource manager is the single bottleneck. Every application master has to talk to the resource manager in order to get resources allocated for tasks it wants to run. In addition, a lot of lookups on the name node can become an issue for scaling the underlying HDFS. A strategy deployed by Uber for mitigation of this problem is splitting the writing and reading of the name node, so that there is one name node that handles writing, and multiple name nodes that handle read requests [1].

In a Mesos cluster, frameworks have to handle job coordination but also scheduling. The framework receives offers from the master and can choose to accept or reject these. Thus, this overhead is distributed across all different frameworks, so scheduling bottlenecks are eliminated in general.

To summarize, Kubernetes scheduling system is monolithic and therefore might pose problems for deployments with several tens of thousands of jobs. Yarn has a similar bottleneck because it has a monolithic resource manager. However, since the application masters handle job coordination run on the cluster nodes, it is expected to scale better. Mesos is even more flexible in this scenario since it delegates not only the job coordination to the framework but also most of the scheduling by only making offers to it.

4.2 Resource Scaling

With an ongoing increase of work to schedule, the cluster's compute capacity will eventually be used completely. At this point the cluster has to be enlarged to handle increased resource demands. Thus it is important to which point the frameworks scale without noticeable limitations and what are the bottlenecks when doing so. One approach is to increase the power of the individual nodes. This is called vertical scaling. The problem with this approach is that this does not scale indefinitely. Also, the cost per compute power increases exponentially, so it is more feasible to add more compute nodes. Thus, the size of the cluster is theoretically unlimited [27]. Kubernetes provides the least amount of scalability when it comes to the physical layer. At the time of writing, only 5,000 nodes are supported [14]. This means that large data processing jobs might have to be split across multiple clusters to handle the workload. There is no way to logically treat this set of clusters as one big cluster, so the application developer has to find a solution himself. Traditionally, Yarn can only scale up to a few of thousand nodes, due to the monolithic resource manager. However, it is possible to combine several federated clusters into a larger super-cluster, containing several tens of thousands of nodes. This means that scaling beyond a few thousand nodes is easier in Yarn since the application developer can treat this super-cluster as one giant cluster [5]. Due to Mesos' non-monolithic nature and lean resource broker, it scales very well with some clusters (running a modified version of Mesos) reportedly spanning over 80,000 nodes [28]. While there is no federated cluster support like in Yarn, the limit is still in the same range as Yarn. However, this approach is less complex and thus less effort.

To summarize, when it comes to scaling raw compute power, Mesos' scheduler provides the biggest flexibility since it is fairly lean and does not need to handle a lot of logic. Yarn requires some setup and maintenance but can scale up to similar sizes as Mesos. Kubernetes in its current form does not scale as much which means that it is necessary to scale vertically once the node limit is reached.

5. CONCLUSION

To conclude, there is no single perfect resource management framework that solves all problems in an optimal way. Mesos is very customizable but requires a lot of tailoring by the application developer, like writing a custom scheduler and executor, in order to reap the benefits of it. This is useful when deploying workloads with complex resource requirements. Kubernetes makes it easy to deploy non-complex jobs. Yet it is flexible when setting up more complex logic, but lacks proper tooling when it comes to data processing. However a complex job with intra-job dependencies requires a custom scheduler just like Mesos and Yarn. Yarn supports jobs with a need for data locality well, but requires similar involvement of the developer compared to Mesos.

6. REFERENCES

[1] Ang Zhang, Wei Yan. Scaling uber's apache hadoop distributed file system for growth. `https://eng.uber.com/scaling-HDFS/`. Accessed: 2019-11-21.

[2] Apache Software Foundation. Apache placement constraints. `https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/PlacementConstraints.html`. Accessed: 2019-11-21; Version 3.2.1.

[3] Apache Software Foundation. Apache resource manager api. `https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/ResourceManagerRest.html`. Accessed: 2019-12-13; Version 3.2.1.

[4] Apache Software Foundation. Apache service discovery. https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/yarn-service/ServiceDiscovery.html. Accessed: 2019-11-21; Version 3.2.1.

[5] Apache Software Foundation. Apache yarn federation. https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/Federation.html. Accessed: 2019-11-21; Version 3.2.1.

[6] Apache Software Foundation. Mesos allocation modules. http://mesos.apache.org/documentation/latest/allocation-module/. Accessed: 2019-11-21; Version: 1.9.0.

[7] Apache Software Foundation. Mesos container storage interface (csi) support. https://mesos.apache.org/documentation/latest/csi/. Accessed: 2019-11-21; Version: 1.9.0.

[8] Apache Software Foundation. Mesos designing highly available mesos frameworks. https://mesos.apache.org/documentation/latest/high-availability-framework-guide/. Accessed: 2019-11-21; Version: 1.9.0.

[9] Apache Software Foundation. Mesos framework development guide. https://mesos.apache.org/documentation/latest/app-framework-development-guide/. Accessed: 2019-11-21; Version: 1.9.0.

[10] Apache Software Foundation. Mesos scheduler http api. http://mesos.apache.org/documentation/latest/scheduler-http-api/. Accessed: 2019-11-21; Version: 1.9.0.

[11] Apache Software Foundation. Platforms powered by mesos. http://mesos.apache.org/documentation/latest/powered-by-mesos/. Accessed: 2019-11-21.

[12] Arinto Murdopo. Towards high availability in yarn: Motivation and proposed solution. http://www.otnira.com/2013/01/19/ha-in-Yarn-motivation-and-proposed-solution/. Accessed: 2019-11-21.

[13] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes. Borg, omega, and kubernetes. *ACM Queue*, 14:70–93, 2016.

[14] Cloud Native Computing Foundation. Building large clusters - kubernetes. https://kubernetes.io/docs/setup/best-practices/cluster-large/. Accessed: 2019-11-21; Version: 1.16.

[15] Cloud Native Computing Foundation. Cni - the container network interface. https://github.com/containernetworking/cni. Accessed: 2019-11-21.

[16] Cloud Native Computing Foundation. Container storage interface (csi) specification. https://github.com/container-storage-interface/spec. Accessed: 2019-11-21.

[17] Cloud Native Computing Foundation. Disruptions - kubernetes. https://Kubernetes.io/docs/concepts/workloads/pods/disruptions/. Accessed: 2019-11-21; Version: 1.16.

[18] Cloud Native Computing Foundation. Handling master replica failures - kubernetes. https://kubernetes.io/docs/tasks/administer-cluster/highly-available-master/#handling-master-replica-failures. Accessed: 2019-11-21; Version: 1.16.

[19] Cloud Native Computing Foundation. Kubernetes user case studies. https://kubernetes.io/case-studies/. Accessed: 2019-11-21.

[20] Cloud Native Computing Foundation. Pod lifecycle - kubernetes. https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/. Accessed: 2019-11-21; Version: 1.16.

[21] CoreOS, Inc. rkt. https://coreos.com/rkt/. Accessed: 2020-01-17.

[22] Docker, Inc. Docker. https://www.docker.com/. Accessed: 2020-01-17.

[23] Z. Guo, G. Fox, and M. Zhou. Investigation of data locality in mapreduce. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (Ccgrid 2012)*, CCGRID '12, pages 419–426, Washington, DC, USA, 2012. IEEE Computer Society.

[24] W. Hasselbring. Microservices for scalability. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering - ICPE 16.* ACM Press, 2016.

[25] B. Hindman and K. E. al. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 295–308, Berkeley, CA, USA, 2011. USENIX Association.

[26] Hongchao Deng. Improving kubernetes scheduler performance. https://coreos.com/blog/improving-kubernetes-scheduler-performance.html. Accessed: 2019-11-21.

[27] Shekhar Gulati. Best practices for horizontal application scaling - archived. https://blog.openshift.com/best-practices-for-horizontal-application-scaling/. Accessed: 2019-12-09.

[28] Timothy Prickett Morgan. Mesos clusters growing to monster sizes. https://www.nextplatform.com/2016/03/24/Mesos-clusters-growing-monster-sizes/. Accessed: 2019-11-21.

[29] Uber. Uber engineering blog. https://eng.uber.com/tag/hadoop/. Accessed: 2019-11-21.

[30] V. K. Vavilapalli and E. a. Murthy. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.

[31] T. White. *Hadoop: The Definitive Guide, 4th Edition.* O'Reilly Media, Inc., 2015.