Master research Internship

Bibliographic report

State of the art in code and test co-evolution

Domain: Software Engineering - Computer Aided Engineering

Author:
Quentin Le Dilavrec

Supervisor:
Djamel Eddine Khelladi
Arnaud Blouin
DiverSE

Abstract: In this study, we will try to establish the state of the art in the co-evolution of code and tests. Software is everywhere, if not in a product it is used to produce or design it. Software is used in every field of research and industry. Due to the concept of separation of concerns, software systems are split into different types of artifacts. Such as tests, are able to detect bugs and help fixing them. But as the rest of the application evolves tests break. So to avoid rewriting the tests every time, we co-evolve tests.

Contents

1 Introduction

Due to the concept of separation of concerns, which is able to reduce complexity, improve reusability, and make evolution simpler [15, 32, 34], software systems are split into different types of artifacts, each targeting particular domain concerns. Ensuring the quality of those artifacts is thus of the utmost importance. To gain a clear overview we can distinguish various types of software artifacts. Most commonly found in software projects are code, application programming interface (API), tests, models, scripts, etc. For example, the class model and the API can be completely integrated into the functional implementation. Moreover, an artifact can also be partially or completely synthesized or generated from others. As an example, it is possible to extract an API from some code or models, it is also possible to generate tests from models or functional implementations. As artifacts share common concepts, when one artifact evolves, other artifacts may be impacted and may need to be co-evolved. In this report, we will focus on the scenario of when code evolves and tests must be co-evolved. For example, moving a method from one class to another makes calls to this method invalid, but most importantly with the right contextual information, it is possible to fix those calls and to co-evolve tests by moving related tests to the proper place while fixing some other contextual differences. However, unfortunately, tests co-evolution remains mainly a manual task for developers, which is tedious, error prone, and time consuming. In particular, when hundreds of developers collaborate together, and where those who maintain tests (testers) are not necessarily those who evolve the code [1].

In the internship we will address the problem of co-evolving tests using information available in the rest of the code and its evolution. While in this article we will establish a state of the art on the co-evolution of code and test. Other survey have targeted the co-evolution meta-models and models [13], the co-evolution of mutant and tests [16], and also the generation of tests [2, 3], but to the best knowledge they were no survey or state of the art on the co-evolution of code and test. This article fills this gap, in preparation of this internship.

The rest of the article is presented as follow: Section 2 presents a short background. Section 3 gives the methodology used to construct this state of the art. Section 4 illustrates the results of the categorization presented in the methodology. Finally section 6 presents the conclusion and initial research perspectives.

2 Background

This first section presents a background on testing and co-evolution. Listing 1 shows a basic example of code and tests.

2.1 Software Testing

Tests allow us to detect bugs to solve them [10]. It is also a way to specify functionalities and constraints. It is not as exhaustive compared to symbolic analysis, but it is often easier to implement. Compared to declarative specifications, it facilitates the specification of complex functionalities while allowing flexibility in the implementation, by sticking to common concepts of imperative programming.

[1] `https://github.com/microsoft/onnxruntime`.

```
export class Counter {
  constructor(
    private x : number) {}
  count(cb?:(n:number)=>number){
    if (cb){
      this.x=cb(this.x);
    }else{
      return this.x++;
    }
  }
}
```

```
test('trivial 1', () => {
  const init = 0;
  const e = new Counter(init);
  expect(e.count()).toBe(init+1);
});

test('trivial 2', () => {
  const e = new Counter(3);
  expect(e.count(x=>x-2)).toBe(1);
});
```

Listing 1: example.ts (left) / example.test.ts (right)

Quantifying software quality is also a major concern, which is addressed by software testing, e.g., mutation testing [38], or by comparing tests and field behaviors [22, 18]

Software testing can take many forms. Each form focuses on particular aspects of software and serve different goals. a) **Unit Tests** are the most known kind of tests, they can detect bugs early in development, they run fast, automatically and help at finding causes of bugs. In Listing 1 on the right, we can see some unit tests targeting the piece of code on the left. Like its name indicate the *class* on the left is a counter, its constructor instantiate the `x` attribute, while its method `count` take a function as an optional parameter, this function modify the `x` attribute by a certain number otherwise `x` is incremented by one. Both unit tests on the right test the `count` method, the first one initialize the counter at 0 then check the result of `count` called with the default parameter, the second one initialize the counter at 3 then check the result of `count` called with a given lambda function. b) **System tests** allow to asses the validity of a program in particular use cases, but contrary to unit tests they are slow and might need human intervention in addition to not helping much at finding causes of bugs. Compared to unit tests in Listing 1 system tests would be much larger and span over many classes at once.

Multiple uses of tests also exists depending on some additional concerns, such as mock testing, regression testing, performance testing, etc. For example, a) **mock testing** allows to abstract from dependencies and focuses on small and very controlled parts of programs, while b) **regression testing** allows to compare different versions of a program to facilitates incremental improvements.

2.2 Co-evolution in generality

With a rough look at most software engineering systems, there are at least a few types of artifacts that are easy to discern like an API, a functional implementation of this API, a model —or specification— of the application and tests to check the implementation against some constraints. But there are many more software artifacts like traces, binaries, metadata, comments, etc. As a matter of fact, there is no clear boundary between each artifact. For example, the class model and the API can be completely integrated into the functional implementation. An artifact can also be partially or completely synthesized or generated from others. As an example, it is possible to extract an API from some code or models, it is also possible to generate tests from models or functional implementations. Moreover in the same way those artifacts are overlapping, depend on each other to work properly and changing one might impact another negatively and hence requires co-evolution.

Definition 1: Co-evolution is the process of modifying a given impacted artifact A, in response to evolution changes of another artifact B.

The co-evolution scenario we will focus on is code evolution and tests co-evolution. The co-evolution of tests can be split in **amplification** and **repair**. The amplification of tests can be seen as the continuation of tests generation in the context of co-evolution as it consider preexisting tests in relation to evolution in the code. One of the difficulties of amplification is the readability of generated tests. Whereas repairing tests with co-evolution using code, considers changes to the code as a way of detecting and fixing tests broken by code changes. Here the major challenges is to keep tests correct. Looking at Listing 1 if we rename the method `count` of the class `Counter` as `update`, calls to the member `count` of instances of `Counter` would also need to be renamed. Similarly, if we make the parameter of method `count` mandatory, we would need to generate a default value for empty calls to method `count`. And as a last one, if we move the method `count` to another class, tests of `count` should be moved to a more appropriate place and the constructors pointing to `Counter` would need to be renamed.

3 Methodology

This section presents our methodology. We propose criteria to categorize approaches that handles the co-evolution of tests. Thanks to those criteria we will be able to classify the literature and to choose better suited techniques depending on particular concerns. Figure 1 illustrates those criteria as a feature model. Another major focus will be to expose relations between objects of studies and solving methods. In the end, it will allow us to find still existing gaps in test co-evolution and to identify research questions and future works perspectives.

This state of the art took inspiration from the survey [13] from Hebig et *al.* on the co-evolution of models. The bibliographical research started with a set of articles given by my supervisors. Then alternating between searches on mainly *google scholar* with keywords from previous papers in addition to following the most relevant references from papers ("snowballing" technic) that I read.

Here we would like to look at co-evolution as a 2 step process, where the first step would be to detect and categorize evolution in the implementation of some program, the second step would be the co-evolution of tests. We first present criteria that are common to both steps.

3.1 Degree of automation

One of the first criterion to consider is the **degree of automation** of the co-evolution. It quantifies the amount of involvement needed by a developer in the process of co-evolution. In case of a full automation one might only have to confirm co-evolution, otherwise in a semi-automated co-evolution one might need to choose between possible resolutions to apply or even create a custom transformation, capable of handling some domain-specific evolution. We consider *manual*, *semi-automated* and *fully-automated* approaches.

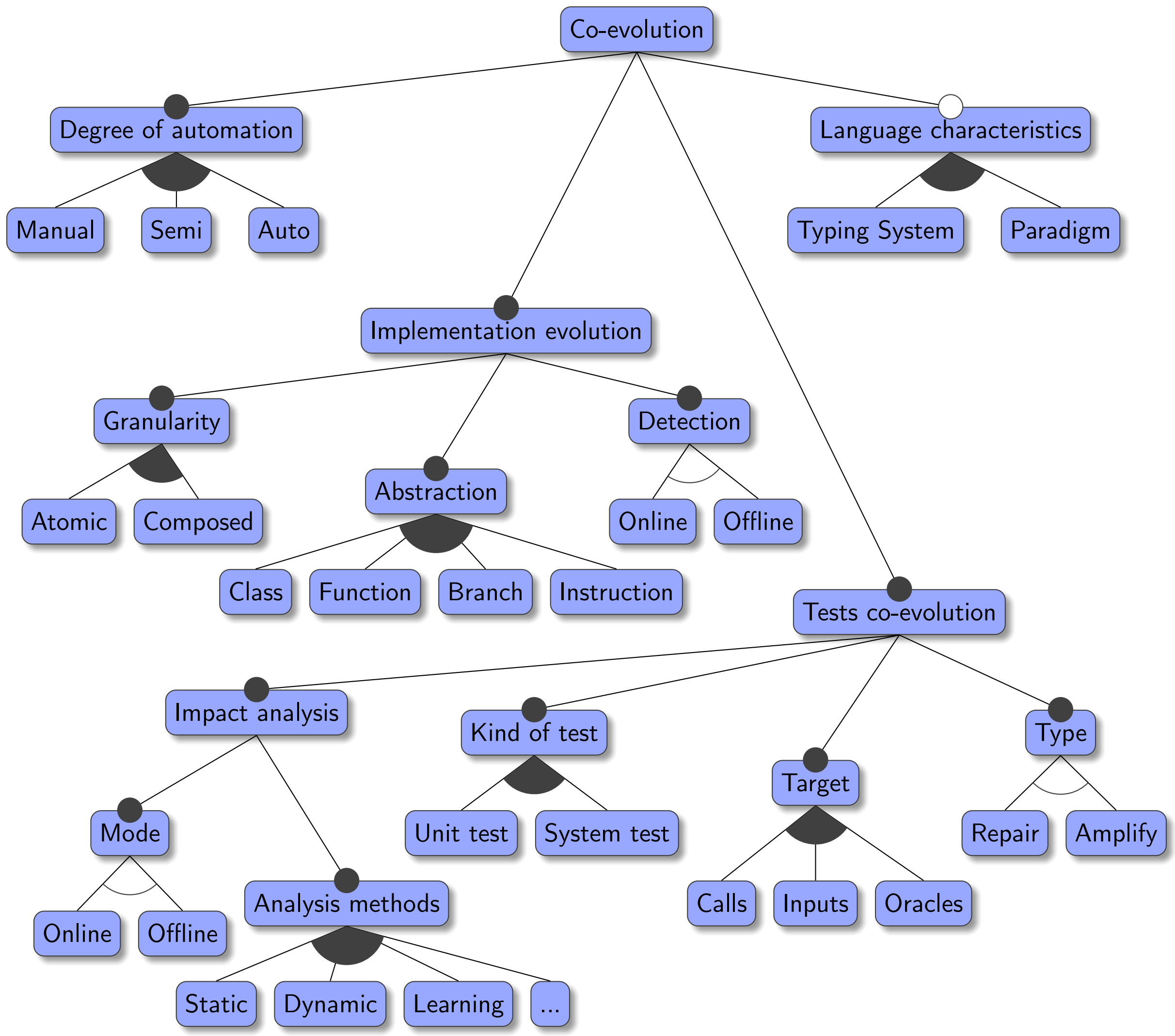

Figure 1: Feature model of co-evolution of code and test [▲ : or, △ : xor, ● : mandatory, ○ : optional]

3.2 Language characteristics

The systems that can be co-evolved possess different characteristics. Those characteristics can particularly be observed through the language point of view. Most software projects use some framework and use a multitude of languages. This multitude of languages might possess common characteristics. We mainly consider the language paradigm like the *Object oriented* (with the Class construct), *Imperative* or *Declarative* paradigms and the type system like *strongly* or *weakly* typed languages.

3.3 Detection and classification of evolutions

Detecting and classifying evolution is the first step in any co-evolution of code and test. Each major criterion composing this step of co-evolution are explained in the following 3 paragraphs.

3.3.1 Granularity

The **granularity of evolution** is very important to the automation of the co-evolution. The simplest kind of evolution is an *atomic* change, while it is very simple to detect simple changes, it does not contain much information. Additions and deletions are the most simple atomic changes, and often the only atomic changes considered. It is possible to combine atomic changes into *composed* changes. For example, moving a method from one class to another is composed of a deletion and addition. Another example of complex change is renaming a method, it is also composed of a deletion and addition, but here the change is much more localized.

3.3.2 Level of abstraction

In every software analysis, the **level of abstraction** reflect the trade-off made between precision and performance. For example, the *file* abstraction can be considered as high abstraction to detect changes in a codebase, the file abstraction is what most compilers for procedural languages are using to avoid recompiling unchanged files. There is also the *class* abstraction, it is one of the most used, as it syntactically and statically presents a large quantity of semantic information. In facts, methods are carrying the behaviors of object, and behaviors can be shared through inheritance. But this abstraction requires the analyzed language to be object oriented and possibly have class, prototypes and an inheritance system. To establish measurements of impacts from changes it is necessary to look at calls, this abstraction is a *call graph.* Finally looking at the level of *flow graphs*, i.e., blocks of instructions linked by branches might be necessary for some analysis but it requires a lot of effort and processing power to compute.

3.3.3 Detection

The **detection of changes** can be done online by logging operations made on files or *offline* by comparing states of files between versions. Detecting changes through *online* logging is more precise but is also more intrusive than offline detection. Online detection can be brittle in case of unlogged changes. Thus all external tools modifying the code would need to provide the set of applied changes.

3.4 Co-evolution of tests

Here, we will look at the particular aspects that concern the actual the co-evolution of tests.

3.4.1 Impact analysis

The **impact analysis** of code changes on tests need to be quantified to propose relevant co-evolution. It allows to locate tests that need to be co-evolved and to provide some more contextual information on tests dependencies

Two modes of impact analysis can be discerned. *Offline* impact analysis is computed when the developer is done with his current set of changes. While *online* impact analysis is computed interactively whenever a change happens.

Many possible analysis methods are preceding to impact analysis depending of the language characteristics of the co-evolved artifacts. The main points of analyzing code here is to measure the impact of changes, and to extract useful information from programs. being capable of measuring code allow to find tests that need to be repaired or relaunched. Analyzing code can also be useful to harvest data and patterns [14] that will allow to better amplify tests. In addition to static analysis, using the history of changes and the behavior of the program during test might prove to allow improvements to the precision and performance of programming assistants.

In the general case, analyzing programs is difficult. The whole stack from an algorithms to run is complex and diverse. Indeed many programming languages use different paradigms. For each language many parsers and compilers exist. There is also many runtime and intermediate representations. It is thus important to find points in this stack where analysis are the most efficient.

The static analysis is often the first choice when one want to analyze a particular program or project. In the best case scenario a static analysis can prove properties of a program for any given inputs. Most domains of science and industry that needs to prove properties use language with rich types systems. But annotating programs can be tedious and lead to bugs. That is why analysis tools make heavy use of type inference to lighten the burden of type annotating. Yet type inference have its limits as uncertainties lower the quality of types through the program. Refining those uncertainties is a major point to improve software quality.

Even if rich type systems are very useful for analysis, programs heavily constrained by types are less flexible, demand more code and use more complex artifacts to alleviate types overhead. There is an obvious trade-off between development flexibility and ease of analysis. Making use of runtime can disambiguate uncertainties through programs and ensure properties with more precision. Combining both static and dynamic analysis offer the possibility to further improve code quality while improving flexibility.

Static analysis It requires type information (annotated or inferred). It can check properties on infinite domains in an exhaustive way. Prove to be efficient on simple programs but able to accept a large number of inputs. Type systems can be languages that don't have explicit annotated types, it is nonetheless possible to use type rules e.g. mono-type in C with everything is an int, can check for null dereferencing (that is dereferencing 0). To improve robustness and flexibility most analysis tools have types that match all types and types that match no types. In practice, it allows incremental typing and type inference.

Many tools exist to analyze programs statically, most of them only work on one language (typescript, compCert, spoon) while some try to be more agnostic (llvm, semantic, pandoc). Focusing on one language allow finer analysis but might not scale to multilanguage projects. While tools handling multiple language might work better on multilanguage projects, to leverage the quantity of work for each language such tools need an intermediate representations of programs.

Static analysis work with semantic models such as class diagram,type system, and so on.

Dynamic analysis It is particularly suitable for highly dynamic and not very typified languages. but it cannot provide absolute guarantees on an infinite domain. Event is it tries to be as close as possible to the actual behavior of the program. Dynamic analysis can be effective on potentially complex programs while accepting fewer inputs than static analysis.

Dynamic analysis woks with functional models such as finite state machines, memory behavior, and so on.

Hybrid analysis Supports static analysis by providing information that is easily accessible to the runtime. It also supports dynamic analysis by directing it to the sensitive points detected during static analysis. Use tests to collect information at runtime and improve inferences from static analysis. Use static analysis to detect pieces of sensitive programs and test and instrument them to better understand them and detect bugs.

3.4.2 Kind of tests

The **kind of tests** targeted by a tests' co-evolution methods could be relevant as *system tests* are much bigger and take longer than *unit tests*. In a way the kind of tests handled by co-evolution methods should give a lead on the scalability of the approach.

3.4.3 Target

The **target** of the co-evolution can be the *calls*, the *inputs* of calls or the expression of *oracles*. Take the example from the background, the value given to the class constructor is an input, while the value in the `toBe` method is part of an oracle. A value can also be used both as an input and as a part of an oracle, like the constant `init`. From another point of view an input value go through what we want to test, while an oracle value avoid passing though what we want to test.

Calls Reproducing functional behaviors observed in production is one of the first requirement to synthesize units test from in field executions. There are many proposed techniques in the literature capable of producing a skeleton of calls for test cases.

Inputs (caution inputs of test or inputs of calls) From an existing test or a skeleton of calls, there are many tools to produce complete tests (almost, in the case of the calls skeleton oracle should also be generated).

Oracles They are assertions to compare input and output values of the tests to detect if those tests pass or fail. Assertions are tricky to repair and generate as it part of the program specification. So the challenge is to mine those from somewhere.

For example, in the first test of Listing 1, a *call* to the constructor of `Counter` is made with the `number` 3 as an *input*, then a *call* to the method `count` is made with a function as an *input*. Finally the *oracle* checks that the value return by the previous call is equal to the number 1.

3.4.4 Type

Given some evolutions two **types of co-evolution** are possible. *Amplification* co-evolution creates new tests from other tests by various exploratory methods (genetics, regression, etc.). *Repair* co-evolution modifies existing tests to make it pass the compilation, or the runtime checks.

4 Classification of Approaches

In this section, we will present the state of the art on co-evolution of code and tests following the classification given by the feature model in Figure 1. We will present some of our results in Table 1 regarding the classification of approaches that detect and classify evolution in software artifacts, mostly code. Then in Table 2, regarding the actual co-evolution of software artifacts, mostly tests. It should be noted that approaches mentioned in one table but not in the other, either only detect evolution or only improve tests without considering evolution.

Table 1: Classification part 1, implementation's evolution [?: not mentioned, N/A: not attributable]

Reference			Language	Granularity	Abstraction	Detection	Automation	Analysis
Vcubranic et *al.*	2003	[4]	all	?	metadata, ...	offline	auto	static
Adamapoulos et *al.*	2004	[1]	Fortan-77	N/A	mutant	offline	auto	static, genetic
Jiang et *al.*	2006	[17]	N/A	composed	event	offline	auto	dynamic
Halfond et *al.*	2008	[11]	java, PHP, http,...	composed	calls, data flow	offline	semi[2]	static
Memon et *al.*	2008	[27]	all	composed	event	online	semi[3]	dynamic
Hassan	2009	[12]	C,C++	atomic	pattern, metadata	offline	auto	static
Daniel et *al.*	2010	[7]	java, .NET	composed[4]	instruction	offline	semi[2]	static, symbolic
Dagenais et *al.*	2011	[5]	java	composed[4]	metadata	offline	semi[2]	static
Jin et *al.*	2012	[18]	C	composed	class, flow graph	offline	auto	dynamic
Mirzaaghaei et *al.*	2014	[31]	java	atomic	class	offline	auto	static
Dagenais et *al.*	2014	[6]	java	composed	pattern	offline	semi[2]	static
Khelladi et *al.*	2017	[20]	OCL	composed	class	online	semi[3]	static
Tsantalis et *al.*	2018	[37]	java	composed[4]	instruction, class	offline	semi[2]	static

Table 2: Classification part 2, test's co-evolution [?: not mentioned, N/A: not attributable]

Reference			Language	Impact Analysis	Kind of test	Type	Target	Automation	Analysis
Adamapoulos et *al.*	2004	[1]	Fortan-77	offline	unit	generate	tests inputs	auto	static, genetic
Halfond et *al.*	2008	[11]	java, PHP, http,...	offline	?	repair	calls in general	semi[2]	static
Memon et *al.*	2008	[27]	all	offline	unit[5]	repair	whole tests	semi[3]	dynamic
Thummalapenta et *al.*	2009	[35]	java	offline	unit	generate	tests calls and parameters	semi[2]	static
Daniel et *al.*	2010	[7]	java, .NET	offline	unit	repair	whole tests	semi[2]	static, symbolic
Robinson et *al.*	2011	[33]	java	offline	unit[5]	generate	whole tests	auto	static
Jin et *al.*	2012	[18]	C	N/A[6]	unit	generate	tests calls and inputs	auto	dynamic
Galeotti et *al.*	2013	[9]	java	N/A[6]	unit	generate	tests calls and inputs	auto	static, symbolic
Tonella et *al.*	2014	[36]	all	offline	all	generate	tests event	semi[2]	dynamic
Mirzaaghaei et *al.*	2014	[31]	java	offline	all	repair, amplify	whole tests	auto	static
Fraser et *al.*	2014	[8]	java	N/A[6]	unit	generate	tests calls and inputs	auto	static
Mirshokraie et *al.*	2015	[29]	js	offline	unit	generate	whole tests	auto	dynamic
Mirshokraie et *al.*	2016	[30]	js	offline	unit	generate	whole tests	auto	dynamic
Khelladi et *al.*	2017	[20]	OCL	offline	N/A[7]	repair	whole models and constraints	semi[3]	static
Kampmann et *al.*	2019	[19]	web-python-sql-C stack	offline	unit	generate	whole tests	auto	dynamic

[2]Makes recommendations, on possible co-evolutions.

[3]Might sometimes require human design choices.

[4]Only consider in place compositions.

4.1 Language characteristics

We were able to extract some redeeming characteristics through the different approaches. As shown in Tables 1 and 2 most of the approaches that we found focus on Object Oriented languages. In particular they use the *Class* construct and heavy type systems available statically, like Java,.NET and C++. These approaches seem to correlates strongly with techniques such as static analysis and patterns recognition. Nonetheless some approaches do not rely on particular characteristics of languages in themselves, like class and static types but they rely on the runtime behavior of the program. These approaches use events at some points with dynamic analysis to produce behavioral models [18, 19, 27]. We also found a few approaches working Declarative constraints systems [20], or on database language paradigms [41, 19].

4.2 Degree of automation

The criterion of the degree of automation will be discussed several times in the next sections under other points of view. So we will be quick here with only a general comment on the tables classifying approaches. In both Tables 1 and 2, the automation criterion refers to the approach in general and not only on evolution or co-evolution, as it was very difficult to distinguish both without trying to reproduce experiments.

4.3 Evolution of the Implementation

4.3.1 Granularity

Table 1 shows a correlation between the granularity of changes and the automation of the corresponding approach, where approaches using composed changes require more manual intervention. The cause of this correlation seems to be that approaches using composed evolution are more complex although they can handle a greater variety of evolution.

All approaches considering evolution use some degree of composed changes, Dagenais et *al.* use some basic compositions in [5, 6], for example, renaming is composed of the deletion of a name and the addition of a new one, here the authors consider the case of in-place renaming, so it is easier to infer the relation between the deletion and addition.

It should be noted that some security approaches which uses dynamic analysis and finite state machine inference are doing a special kind of change detection using a fixed initial state [17]. They actually try to detect behavioral changes from the runtime.

4.3.2 Abstraction

Both Tables 1 and 2 show that the abstraction of choice is the class construct.

Some approaches make use of metadata to mine patterns in Content Versioning Systems (CVS). Zaidman et *al.* in [39, 40] mine co-evolution patterns in SVN commits, while Martinez et *al.* in [26] mine co-evolution patterns in git commits.

[5]In the context of regression testing.

[6]Do not use changes to generate tests.

[7]Co-evolve OCL constraints.

We were also able to find some studies classifying changes. Here they use statistics and learning algorithms to predict the type of changes [25, 23, 24] . They combine the class abstraction with metadata from CVS that are analyzed through Natural Language Processing (NLP).

4.3.3 Detection

With the exception of Khelladi et *al.* in [21, 20] who are able to detect changes online because models design has been historically supported in many graphical interfaces. Just as shown in Table 1, most of the approaches that we found use offline detection. These approaches deal with contents that can be edited in many ways, making it difficult to change each editing mode. Thus, these articles rely either on file metadata and file diffs to detect changes [31, 7, 11], metadata of CVS and blob differences [26, 12, 5, 4, 37], or behavioral differences [27, 17].

4.4 Co-evolution of tests

In this section we found many article doing tests co-evolution. We also found approaches that were not exactly co-evolving tests but are still relevant to consider. They do not call their approach co-evolution but they share many tools and algorithms. This increased variety of approaches could be beneficial to the internship.

4.4.1 Impact Analysis

Analysis mode All the articles retained in the state of the art are doing offline impact analysis. There is therefore either no need for test co-evolution during code changes, or the current test co-evolution techniques are too expensive to react to each change.

Analysis methods We found different methods of impact analysis.

Static analysis is the most wide spread type of analysis here, as shown in Tables 1 and 2. The causes of this distribution seem to be due to the large amount of semantic and structural information available in strongly typed object-oriented languages such as Java,

On the contrary, **dynamic analysis** does not appear to be very common, in fact, dynamic analysis is particularly suitable for highly dynamic and weakly typed languages, such as Javascript and Perl. But it requires to go down to the runtime of the program which causes a performance penalty and an increase in complexity. Nonetheless in [19] Kampmann at *al.* use dynamic analysis to synthesize unit tests from system tests through the use of behavioral models and fsm inference algorithms.

Mirshokraie et *al.* in [28] then in [29, 30] combine dynamic analysis and **mutation testing** to improve tests of Javascript programs.

Hybrid analysis seem to be in many future works [3] but we did not find approaches explicitly claiming it in the context of co-evolution.

4.4.2 Kind of tests

We found no approaches claiming to be able to repair or generate system tests. So the hypothesis on the computational complexity of these approaches does not seem invalid.

Kampmann et *al.* in [19] use system tests to generate new unit tests. Mirshokraie et *al.* in [30] also use system tests but in the from of GUI tests to generate new unit tests.

Memon et *al.* in [27] generate unit tests in the particular case of regression testing. Here the regression testing allows the creation of oracles from the program current behavior.

Khelladi et *al.* [20] do not co-evolve tests but a very close artifact. In fact, they co-evolve OCL, a declarative constraint language on models such as class diagrams. Here specifying constraints is very similar to specifying oracles.

4.4.3 Target

Tonella et *al.* [36] and Jiang et *al.* [17] use traces and fsm to construct a functional behavioral model of an application, then they generate new tests as skeletons of **calls** from paths in the fsm. Halfond et *al.* [11] detect parameter mismatch in multi-languages systems. Dagenais et *al.* [5, 6] recommend alternatives for broken calls and for general references. Fraser et *al.* [8] produce tests composed of calls and inputs from java generics. Daniel et *al.* [7] compute new **inputs** for tests that maximize coverage through symbolic execution. Adamopoulos et *al.* [1] amplify inputs through mutation testing and genetic algorithms. Zhang et *al.* [41] amplify tests for database systems through the use of symbolic execution and genetic algorithms. Table 2 shows that fully automated approaches generating (non regression) unit tests are not producing tests with **oracles**. With Kampmann et *al.* and Mirshokraie et *al.* as exceptions in [19] and [29, 30] because these approaches borrow oracles from system tests (such as GUI tests) to generates unit tests. combine dynamic analysis and **mutation testing** to improve tests of Javascript programs. In facts, oracles are part of the application specification, thus there can not be automatically generated. To overcome this restriction, Mirzaaghaei et *al.* [31] use oracles from other tests, Kampmann et *al.* [19] run system tests with the same inputs as unit tests to reduce false positives triggered by oracles in unit tests (if an oracle from an unit test fails, the corresponding system test should also fail). Khelladi et *al.* [20] repair constraints which is very similar to repairing oracles.

4.4.4 Type

Robinson et *al.* [33] use static analysis to generate tests then mutation testing to refine generated tests. Kampmann et *al.* [19] synthesize unit tests from system tests.

Mirzaaghaei et *al.* [31] amplify and repair unit tests using carefully handcrafted patterns that matches certain evolution. However creating these pattern can be tedious. A partial solution to this problem could come from Khelladi et *al.* in [21], who are combining repairing rules to co-evolve models given changes in metamodels.

Daniel et *al.* [7] repair tests using symbolic execution, more specifically they focus on repairing string literals. Memon et *al.* [27] repair regression GUI tests, more specifically they repair the sequence of GUI events, sometimes it needs manual interventions when the approach does not find an appropriate resolution.

.

5 Related works

The problem of co-evolving software has been tackled by many researchers.

For the co-evolution of models, problems have been extensively investigated, Hebig et al. propose a survey [13]. For the co-evolution of tests, the research is much more sparse, such that there is to our knowledge, no survey on co-evolution of code and tests. Nonetheless, there are some exploratory studies on the co-evolution of code and tests, where the evolution of tests are empirically accessed in software life-cycles [22, 39, 40]. There exists also neighbor works to the co-evolution of code and tests, be it on test generation or mutation testing. In [2], Anand et *al.* surveyed recent test generation techniques, while in [3], Andreasen et *al.* showed the difficulties of test generation in dynamic languages. Mutation testing shares some tools and techniques with the co-evolution of code and tests, in [16] Jia et *al* organize different such tools.

6 Conclusion

In this state of the art we have shown a large variety of approaches to the co-evolution code and tests. We have seen a majority of approaches working on Java and mostly richly typed OO programming. With approaches capable to co-evolve more and more different evolutions. Yet they do not consider complex evolution. Moreover, recent works have tried to tackle more challenging languages constraints such as weakly typed and dynamic language (javascript). In the particular case of test generation. But we did not find any approaches capable of co-evolving such challenging languages.

As future perspectives this state of the art could lead to a survey, as more time would allow a more systematic review of the field along with check the availability of tools. One first objective would be to check the feasibility of the co-evolution of tests in context were static type information is more scarce like in dynamic languages. Some of these languages are very popular for their flexibility, their lack of readily available type information makes it harder to analyze. Part of this difficulty seems to have been mitigated by incremental type systems. Thus, we hope that more incremental approaches to the co-evolution of code and tests would allow making use of tests to further analyze code which will then allow to further improve tests. Another objective would be to address tests co-evolution for real world complex evolution.

References

[1] Konstantinos Adamopoulos, Mark Harman, and Robert M Hierons. How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution. In *Genetic and evolutionary computation conference*, pages 1338–1349. Springer, 2004.

[2] Saswat Anand, Edmund K Burke, Tsong Yueh Chen, John Clark, Myra B Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil Mcminn, Antonia Bertolino, et al. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013.

[3] Esben Andreasen, Liang Gong, Anders Møller, Michael Pradel, Marija Selakovic, Koushik Sen, and Cristian-Alexandru Staicu. A survey of dynamic analysis and test generation for javascript. *ACM Computing Surveys (CSUR)*, 50(5):66, 2017.

[4] Davor Čubranić and Gail C Murphy. Hipikat: Recommending pertinent software development artifacts. In *Proceedings of the 25th international Conference on Software Engineering*, pages 408–418. IEEE Computer Society, 2003.

[5] Barthélémy Dagenais and Martin P Robillard. Recommending adaptive changes for framework evolution. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(4):19, 2011.

[6] Barthélémy Dagenais and Martin P Robillard. Using traceability links to recommend adaptive changes for documentation evolution. *IEEE Transactions on Software Engineering*, 40(11):1126–1146, 2014.

[7] Brett Daniel, Tihomir Gvero, and Darko Marinov. On test repair using symbolic execution. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 207–218. ACM, 2010.

[8] Gordon Fraser and Andrea Arcuri. Automated test generation for java generics. In *International Conference on Software Quality*, pages 185–198. Springer, 2014.

[9] Juan Pablo Galeotti, Gordon Fraser, and Andrea Arcuri. Improving search-based test suite generation with dynamic symbolic execution. In *2013 ieee 24th international symposium on software reliability engineering (issre)*, pages 360–369. IEEE, 2013.

[10] Péter Gyimesi, Béla Vancsics, Andrea Stocco, Davood Mazinanian, Árpád Beszédes, Rudolf Ferenc, and Ali Mesbah. Bugsjs: A benchmark of javascript bugs. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 90–101. IEEE, 2019.

[11] William GJ Halfond and Alessandro Orso. Automated identification of parameter mismatches in web applications. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 181–191. ACM, 2008.

[12] Ahmed E Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering*, pages 78–88. IEEE Computer Society, 2009.

[13] Regina Hebig, Djamel Eddine Khelladi, and Reda Bendraou. Approaches to co-evolution of metamodels and models: A survey. *IEEE Transactions on Software Engineering*, 43(5):396–414, 2016.

[14] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 837–847. IEEE, 2012.

[15] Walter L Hürsch and Cristina Videira Lopes. Separation of concerns. 1995.

[16] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering*, 37(5):649–678, 2010.

[17] Guofei Jiang, Haifeng Chen, Cristian Ungureanu, and Kenji Yoshihira. Multiresolution abnormal trace detection using varied-length n-grams and automata. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 37(1):86–97, 2006.

[18] Wei Jin and Alessandro Orso. Bugredux: reproducing field failures for in-house debugging. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 474–484. IEEE, 2012.

[19] Alexander Kampmann and Andreas Zeller. Bridging the gap between unit test generation and system test generation. abs/1906.01463, 2019.

[20] Djamel Eddine Khelladi, Reda Bendraou, Regina Hebig, and Marie-Pierre Gervais. A semi-automatic maintenance and co-evolution of ocl constraints with (meta) model evolution. *Journal of Systems and Software*, 134:242–260, 2017.

[21] Djamel Eddine Khelladi, Roland Kretschmer, and Alexander Egyed. Change propagation-based and composition-based co-evolution of transformations with evolving metamodels. In *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pages 404–414. ACM, 2018.

[22] Maurizio Leotta, Diego Clerissi, Filippo Ricca, and Paolo Tonella. Capture-replay vs. programmable web testing: An empirical assessment during test case evolution. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 272–281. IEEE, 2013.

[23] Stanislav Levin and Amiram Yehudai. Using temporal and semantic developer-level information to predict maintenance activity profiles. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 463–467. IEEE, 2016.

[24] Stanislav Levin and Amiram Yehudai. Boosting automatic commit classification into maintenance activities by utilizing source code changes. In *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering*, pages 97–106. ACM, 2017.

[25] Cosmin Marsavina, Daniele Romano, and Andy Zaidman. Studying fine-grained co-evolution patterns of production and test code. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, pages 195–204. IEEE, 2014.

[26] Matias Martinez and Martin Monperrus. Coming: a tool for mining change pattern instances from git commits. In *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings*, pages 79–82. IEEE Press, 2019.

[27] Atif M Memon. Automatically repairing event sequence-based gui test suites for regression testing. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 18(2):4, 2008.

[28] Shabnam Mirshokraie, Ali Mesbah, and Karthik Pattabiraman. Efficient javascript mutation testing. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 74–83. IEEE, 2013.

[29] Shabnam Mirshokraie, Ali Mesbah, and Karthik Pattabiraman. Jseft: Automated javascript unit test generation. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10. IEEE, 2015.

[30] Shabnam Mirshokraie, Ali Mesbah, and Karthik Pattabiraman. Atrina: Inferring unit oracles from gui test cases. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 330–340. IEEE, 2016.

[31] Mehdi Mirzaaghaei, Fabrizio Pastore, and Mauro Pezzè. Automatic test case evolution. *Software Testing, Verification and Reliability*, 24(5):386–411, 2014.

[32] Harold Ossher and Peri Tarr. Using multidimensional separation of concerns to (re) shape evolving software. *Communications of the ACM*, 44(10):43–50, 2001.

[33] Brian Robinson, Michael D Ernst, Jeff H Perkins, Vinay Augustine, and Nuo Li. Scaling up automated test generation: Automatically generating maintainable regression unit tests for programs. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 23–32. IEEE, 2011.

[34] Peri Tarr, Harold Ossher, William Harrison, and Stanley M Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No. 99CB37002)*, pages 107–119. IEEE, 1999.

[35] Suresh Thummalapenta, Tao Xie, Nikolai Tillmann, Jonathan De Halleux, and Wolfram Schulte. Mseqgen: Object-oriented unit-test generation via mining source code. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 193–202. ACM, 2009.

[36] Paolo Tonella, Roberto Tiella, and Cu Duy Nguyen. Interpolated n-grams for model based testing. In *Proceedings of the 36th International Conference on Software Engineering*, pages 562–572. ACM, 2014.

[37] Nikolaos Tsantalis, Matin Mansouri, Laleh M Eshkevari, Davood Mazinanian, and Danny Dig. Accurate and efficient refactoring detection in commit history. In *Proceedings of the 40th International Conference on Software Engineering*, pages 483–494. ACM, 2018.

[38] Qianqian Wang, Yuriy Brun, and Alessandro Orso. Behavioral execution comparison: Are tests representative of field behavior? In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 321–332. IEEE, 2017.

[39] Andy Zaidman, Bart Van Rompaey, Serge Demeyer, and Arie Van Deursen. Mining software repositories to study co-evolution of production & test code. In *2008 1st international conference on software testing, verification, and validation*, pages 220–229. IEEE, 2008.

[40] Andy Zaidman, Bart Van Rompaey, Arie van Deursen, and Serge Demeyer. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical Software Engineering*, 16(3):325–364, 2011.

[41] Pingyu Zhang, Sebastian Elbaum, and Matthew B Dwyer. Automatic generation of load tests. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 43–52. IEEE Computer Society, 2011.