Using Deep Learning to Identify Technical Debt

Levereging Self Admitted Technical Debt

Master's Thesis submitted to the
Faculty of Informatics of the *Università della Svizzera Italiana*
in partial fulfillment of the requirements for the degree of
Master of Science in Informatics
Major in Artificial Intelligence

presented by
Simone Giacomelli

under the supervision of
Prof. Dr. Gabriele Bavota
co-supervised by
Dr. Csaba Nagy

January 2021

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Simone Giacomelli
Lugano, 1 January 2021

Dedication

Abstract

Composed of four parts:
a: context
b: problem
c: proposed solution
d: results

max 400 words. I will write it once the intro has been approved

Acknowledgements

Ack

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Technical Debt (TD) is a metaphor between the financial concept and software development. A TD is contracted when a workaround or shortcut is taken during code implementation. Choosing an easy and quick solution over a slower and more correct one gives the benefit to save time and deliver the artifact faster.

The downsides of incurring in TD are many, one of the most intuitive is that further work on the affected parts will be more expensive and time consuming than on clean and healthy code. There are also indirect effects to contemplate: the software could misbehave in the domain it is operating, causing costs and damages as the effect of unexpected output or wrong behavior [22].

TD do not only appear in software projects but can also be found in many layers of a technology stack: for example, delaying an hardware upgrade or a maintenance can give an immediate benefit of less downtime or financial savings, but an increased cost for future unexpected downtime or failures [1].

Developers or managers can choose to incur in TD because of strict deadline, limited resources available or just plain laziness [15, 1]. Cunningham, who coined the technical debt metaphor, writes in "The WyCash portfolio management system" [5]: "A little debt speeds development so long as it is paid back promptly with a rewrite". Cunningham implies that one could benefit from a small amount of TD but its should be paid back as soon as possible.

TD can arise from intentional and unintentional decision, e.g. an inexperienced person could contract it without being aware of it [15]; In both cases it's often done for saving the limited available resources and shortening time-to-market [21]. For example, startups are highly pressured to quickly test products and ideas in order to save capitals and be faster than competitors. Besker at al. studied how startups incur in TD, which are the factors, challenges and benefits of intentional acquisition of TD; two of the regulating factors found by the authors are the experience of the developers and the software knowledge of the founders [3].

In contrast to the beneficial viewpoint of TD, Ron Jeffries argues that the metaphor

could be "perhaps too gentle", because it highlights the wise aspect of the choice of contracting a debt; the problem is that people also takes debt unwisely. Technical debt benefits a software project as long as it is handled before the bigger long term cost is realized [11].
******
It is useful to list some aspects of TD that form the basis for a tractable model of the phenomenon; many of these aspects comes from the already well known attributes of financial debt. The *principal* is the amount that needs to be payed to 'make things right'. The *interest amount* is the added cost in addition to the principal;

In order to better analyze and create a tractable model of TD, Alves et al. identifies three variables [18]:

Identified variables (3) Tracking and managing is important Few proposed tracking and managing examples

there is a cost paying the debt and there are interest costs when efforts are wasted coping with non optimal code.

Why detecting it is important

the satd is there
it's undocumented
the team is not aware of it

Tom et al. in their systematic literature review studied, among others, the benefits and drawbacks of incurring in TD [21]; the part we are now interested in is the drawbacks:
Increasing costs over time, such as the amount of effort required to deliver a certain amount of functionality
Work estimation becomes difficult
Developer productivity is negatively impacted
Becomes increasingly difficult to repay as decisions are affected by existing debt
Increased risk involved in modifications to the system
Change becomes prohibitively expensive to the point of bankruptcy, and a complete rewrite and new platform may become necessary
Decreased quality in the end product

Ci sono team che risolvo as-you-go, altri hanno strategie per rilevare, identificare e fixare. It has been studied that TD [Investigating the Impact of Design Debt on Software Quality] E' importante rilevare e monitorare i TD perche' questi aumentano il costo di un progetto software, costo dovuto dall'interesse che si deve pagare in relazione al TD.

1.1 Objectives and Results

Once the context is clear, I will explain the goal of the thesis and summarize the achieved results The goal of the thesis is to exploit SATD to train a model that can acquire the capacity to distinguish between TD-free code and code affected by TD. Using the comments in open source projects I will identify class methods noted as SATD. Through the vcs commit history I will identify when this comments disappear; the assumption here is that when the comment is removed from the code the SATD has been fixed. Many cases are excluded to minimize the probability of keeping a false positive, i.e. the SATD is not fixed but the comment is removed. For example, the simplest case excluded is when the code is exactly the same and the only change is the SATD comment removal. Another reason of exclusion is when the code is changed too much; in such case it would not be prudent to keep the sample in the dataset.

The achieved results tell us that it is difficult to predict with high accuracy on methods with big bodies. As the train dataset is limited to shorter and shorter method size, the accuracy grows.

1.2 Structure of the Thesis

A simple bullet list saying "Chapter 2 presents the state of the art, bla... Chapter 3 ..."

- Chapter 2 presents the state of the art
- Chapter 3 explains how to leverage SATD to train a Deep Learning model to detect TD
- Chapter 4 Empirical Study design
- Chapter 5 Results discussion
- Chapter 6 Threats to validity
- Chapter 7 Conclusion

Chapter 2

State of the Art

The following sections describe existing approaches to detect different types of technical debt.

2.1 Automatic Identification of Software Bugs

This section is about bugs and tools that are capable of detecting them automatically. We divide bugs in three different categories:

- Memory related bugs, e.g. null pointer violation, memory leak and buffer overflow.
- Concurrency related bugs, e.g. critical race conditions, deadlock and unsafe concurrent data access.
- Semantic related bugs, when the fault arise from a contradiction to the intention of the programmer or the original design.

Different tools use different methods to detect bugs; the first big distinction to be made is between dynamic and static analysis.

Static analysis is performed with no execution of the program and relies only on the source code or on some object code.

Dynamic analysis executes the program and inspect its behaviour at run-time. It can be implemented in many flavours, for example: with instrumentation of the executable, with a virtual processor or taking the form of a scheduler. Two are the main factors that are often taken under observation, one is the performance loss in the execution and the second is the extension of the run-time monitor (both in quality and quantity).

We can define another distinction on how bugs are identified; many tools use rules to detect if some violation has occurred. The rules themselves are of two types: programming and statistical rules.

Programming rules are usually clear and squared, e.g. they derive from axioms, mathematical models or are manually defined.

Statistical rules are defined through a statistical analysis on multiple samples. Usually there is a training phase where observations are collected and correct rules (invariants) are refined.

Another means of detecting bugs is *model checking* where the tool verify correctness in a usually finite state system. This verification is performed using formal methods on typically formally specified system.

The last category for bugs finding techniques is the *annotation-based* tools. Those requires the annotation of programs to extract semantics and verify consistency and correctness.

What follows is the description for each selected tool.

PREfix [4] A static analyzer for finding dynamic programming errors

PREFix is a source code analyzer that detects a broad range of errors in C and C++ code. Its goal is to detect many runtime issues on real world programs, without dynamic analysis and instrumentation; only the source code text is used. PREfix can detect defects efficiently through a model that abstracts functions and their interactions; the analyzer traces execution paths handling multiple language features: pointers, arrays, structs, bit field operations, control flows statements and so on.

The method used by the tool is based on the simulation of individual functions. It employs a virtual machine that simulates the actions of each operator and function call. With the detailed tracking it can report defects information to the user so to easily characterize the detected error. The tool can be applied both to a complete program source or only a subset. This bottom up approach is particularly useful when the source code is not fully available (e.g. in the case of a thirty part library).

RacerX [6] RacerX: effective, static detection of race conditions and deadlocks

RacerX deals with complex multithread systems. It detects race conditions and deadlock using static analysis. It can infer from the source code the object lock that is assigned to a particular code block. It detects code that is used in a multi-threaded context; it also detects when the code endeavours in dangerous shared access.

RaceX uses annotations only to mark the code that deals with lock acquisition; this requirement keeps the burden on the user to the minimum to increase ease of adoption. The authors of RaceX report their experiences on the biggest problem about race detection: in large codebase there are massive amounts of unprotected variable access; the key point is to report only those that can actually cause problems. There is emphasis on two aspects:

- The first is to minimize the impact of reporting false positive in order to avoid the users to discard the use of the tool; to achieve this, RaceX employs specific techniques to lower the impact of analysis mistakes.

- The second is the speed of the tool: the authors keep the time of execution for the analysis under deep scrutiny; they claim that for a codebase of 1.8 LOC the time required is between 2-14 minutes.

The tool has been found capable of finding severe problems in huge projects like Linux, FreeBSD and also in a large closed source commercial software.

Purify [13] Fast Detection of Memory Leaks and Access Errors. Winter 1992 USENIX Conference

Purify is a dynamic analysis tool for software testing and quality assurance . It instruments the object files generated by the compiler (the software dates back to 1992 and the supported platform is Sun Microsystem's SPARC); the process that acts on the object files include also third-party libraries. Purify detects multiple errors: memory leaks, access error, reading uninitialized memory. The injected instructions check every read and write memory operations; the slow down of the target is under three times in respect to the non-instrumented execution time. Enabling the use of Purify is as simple as adding a word in the makefile; the generated overhead is inside the limit of tolerance of developers and it allows to detect bugs early in the development cycle.

Valgrind [20] Valgrind: a framework for heavyweight dynamic binary instrumentation

Valgrind is a framework available for many Linux, Android and Darwin architecture. It is the foundation where many tools are built upon. All these tools, several are already included with the standard distribution, help to build more correct and faster program and are categorized as dynamic analysis tools.

The eight supplied tools can be divided in the following groups: memory error detector, cache profiler, thread error detector, heap profiler. There are also two additional tools that are provided to illustrate how to use the framework works and how to use the core low level infrastructure to implement instrumentation.

Basically, the core implements a synthetic CPU that asks the selected tool how to instrument the code and then continues and coordinates the execution. All the instructions are simulated and the memory access is sandboxed; this includes also the third party library linked into the executable. There are ways to manage and suppress every output generated to avoid clutter and unwanted error reports.

It is advised to enable the debug info into the executable; without them Valgrind is unable to determine which function is the owner of a specific instruction, as such it will produce almost useless error and profiling messages. It is also advised to use minimal compiler optimization to avoid incurring into false positive error reports.

The slowdown of the execution depends on the specific instrumentation of the selected tool, and it is roughly between four to fifty times of the original speed.

CP-Miner [16] CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code

CP-Miner stands for Copy and Pasted code Miner. Zhenmin et al. found that a significant portion of source code in many widely used open source projects are duplication made with copy and paste. This practice introduces bugs, the main reason being that programmers leave identifier untouched instead of renaming them consistently to match the new code context. When the label do not exists in the new place it will be detected with a compilation error; on the other hand, if the identifier exists in the new context it will not be detected by the compiler and it will introduce a hidden bug very hard to detect.

CP-Miner tolerates modification to the pasted code. In order to be detected, the segments do not need to be identical; they can also contain insertions or modifications. The tool is capable of detecting duplicated code but not all detection are true positive; nonetheless it is able to report many significant duplicates with hidden issues. The authors detected 28 copy-paste related bugs in the Linux kernel code base and 23 in FreeBSD; these bugs were reported and most of them where previously unknown to the project team. The analysis is done infra-project and targeted the following large projects: Linux, FreeBSD, PostgreSQL and Apache HTTP server.

The main contributions of the paper are: scalability, bugs detection and statistical study of the copy-pasted code.

- *Scalability* is a strong point of CP-Miner because the technique used in it allow to quickly and efficiently scan large projects including operating system code. For example, it took 20 minutes to find 150,000-190,000 copy-pasted segments respectively in the FreeBSD and Linux kernel; such fragments account for roughly one fifth of the code base. At the time, both projects had more the three million lines of code.

- *Bugs detection* was found to be very effective because of the positive response from the open source project maintainers; most of the reported bugs were not found by static or dynamic analysis detection tools.

- The authors conducted a *statistical study* of the copy-pasted code to give an overview of the phenomenon; the majority of the copied code is between 5 and 16 statements. Around 50 percent of the code has only two copies but around 7 percent has more than eight duplicates. Roughly 12 percent of the copy-pasted code segments are whole functions. Kernel modules are affected with different concentration, depending on the module under analysis: drivers, arch and crypt modules have high copy and paste segments than other parts of the project.

D. Engler's [7] Bugs as Deviant Behaviour: A General Approach to Inferring Errors in Systems Code

The approach taken by Engler et al is to extract from source code beliefs and properties that must or could hold. Through static analysis it is automatically detected two different types of beliefs: MUST beliefs and MAY beliefs. The first one is something that must certainly hold, for example the dereference of a pointer holds the credence that the reference is not null and must be valid. The second one is statistical by nature; the code is observed and searched for patterns that suggests beliefs, for example a call to function "x" followed by a call to function "y". The probabilistic nature of the MAY belief comes when validating it: at first it is treated as a MUST belief then a search yields all uses in the source code of such belief (i.e. the use of "x" and "y"). If it turns out that such pattern is respected most of the times then the belief is probably valid, otherwise it is treated as a coincidence and, as such, discarded. MUST beliefs bear no doubt about their validity and represent internal consistency. All contradictions from both MUST beliefs and valid MAY beliefs are reported as errors (i.e. bugs). The authors leverage their prior work [8] where they used static analysis to fix manual defined rules for specific system; for example a call to spin_lock(l) must be paired with a following spin_unlock(l). Such patterns were previously specified by hand; with the current work they enable a system to infer the same (and more) rules automatically. They reported that the automatic system is able to detect all the manual rules plus a considerable additional amount that goes from ten to one hundred patterns more. The general idea underling this paper, is that the source code contains intrinsic information about what is correct; finding errors in real system means exploiting what is intended as "correct". Sadly, most of the times these rules are not documented, not formalized or if they are available, they are present in informal and unusable format. With this work the authors use static analysis to extract the beliefs that the programmers infused in the source code, without the need of a priori knowledge. Manually performing the task of extracting correct behaviours and rules from source code is usually a hard, difficult and daunting experience, particularly in view of multiple releases and big code bases. Engler et al show of being able to apply this techniques to complex systems as Linux and OpenBSD operating systems. The results are hundreds of detected contradictions (i.e. errors or bugs) reported; many of them have been assessed and resulted in kernel patches.

DIDUCE [12] Tracking Down Software Bugs Using Automatic Anomaly Detection

The paper introduces a tool written by the authors called DIDUCE: Dynamic Invariant Detection ∪ Checking Engine. It is based on instrumentation of Java programs so to observe their behaviour at run-time. During the lifetime of the target Java process, DIDUCE gather information and collects hypothesis of invariants; those are the rules that the program should obey. As violation to the invariants are encountered the tool relaxes the hypothesis allowing for different behaviour. Every invariant has a confidence level, the process previously described updates it; then the user can go through all the anomalies reported ordered by their rank. The tool is intended to be used in the discovery of the root cause of bugs and as an aid in better understanding the program under analysis. The query of this ranking can be done, for example, just before a crash occurs: inspecting what DIDUCE detects as anomalies often lead to the discovery of the root cause of the problem.

The paper describes also the findings during the application of DIDUCE to four Java real-life programs, one of which is the JSSE Library (Java Secure Sockets Extension); the bug under analysis was found during the development of a proxy server to be applied to the JSSE Library. The problem was that using the proxy server triggered unexpected behaviour in the JSSE internals. Thanks to DIDUCE the programmer was able to find the issue in the core of the library: it was reported with high confidence that method read() returned a different value than usual. This method returns the number of bytes read from the stream and Java specs clearly define that the method can also return with a buffer not fully filled. A common Java programmers pitfall is to ignore this result and skip the loop on read() to obtain all expected data. DIDUCE reported a violated invariant with high confidence because the result from the method was always 74 and in one instance (just before the Exception) the number was smaller; this information made apparent that using the proxy triggered a bug present in JSSE Library. The authors' experience suggests that discovering bugs using their proposed methodology is simple across many different kind of programs, shown in the paper with four compelling cases.

AccMon [26] AccMon: Automatically Detecting Memory-related Bugs via Program Counter-based Invariants

The contributions of this paper are two innovative ideas: the first is a novel statistical approach to detect bugs in memory related issues, the second is a novel architectural extension to decrease the overhead of the monitoring process.

The first is called PC-based invariant detection (PC stands for program counter); it leverages the observation that most programs access memory location mostly from the same instructions. Being probabilistic in nature, it can detect memory access anomalies that deviate from the baseline; they usually are the causes for bugs, stack overflows and many other memory-related issues. We can see that there are tho phases: one where the statistical data are gathered and the baseline rules (e.g. invariants) are formed; the other phase put to use the collected rules and check for violations that will be reported. These two phases are intended to be used in multiple runs but also in the same single long-running execution.

The second contribution is called Check Look-aside Buffer (CLB); it aims to lower the burden of the dynamic process monitor activities to decrease the overhead needed. The authors report their experiments on performance: in the worst case analyzed the loss of speed of the process is less than 3 times. In other tools, this slowdown can be of one order of magnitude greater.

The effectiveness of the authors' contribution are tested through AccMon, a tool they developed in order to implement the idea of a PC-based invariant detection; such experiments shows that the proposed novel ideas are sometime capable of finding more bugs in respect to other tools such as Purify or CCured. Then, in conjunction with previous work from the same authors (iWatcher) it is demonstrated the effectiveness of CLB in lowering the burden of the overhead.

The authors report many other advantages in using AccMon in respect to other tools:

- the analysis is not done on the values of the variables, thus It can detect also bugs that do not violate value-based invariants.
- in the current form, it uses source code to achieve compiler-based optimization but it can directly work with binaries without the need of compilation.
- it does not need type information; given it's statistical nature it can detect anomalies just using abstract memory pointers.
- it's possible to switch off the monitoring activities dynamically at runtime, with almost no overhead. The authors states that AccMon can be used in production runs.

Liblit's [17] Bug isolation via remote program sampling

The underling observation that pave the road for this paper is that often the user community of a program has more raw throughput of running and executing it than the developers. In other words, the number of executions that the team responsible for the program can apply for testing is dwarfed by the number of executions that the community can or will bring up to bear. The authors propose an infrastructure for gathering data from the user's execution to a central information store; then they propose a process, called automatic bug isolation, to analyze gathered data in order to provide information to the developers to help find and fix bugs. This infrastructure shows multiple benefits:

- Use a vast amount of data that is generated by the execution of the program by the user community; it is usually discarded and do not contribute to better the quality and the experience for the user itself.
- Enabling the collection of information helps to draw a clearer picture on the effective use of the program and drive better decisions about development roadmap
- Map and define feature usage statistics
- Avoid issues related to manual feedback generated from an user intervention: usually the user is unsophisticated and non technical; it's ability to have a positive impact on the bug reporting is limited. The benefits of automatically gather this information are many and varied.

Designing an infrastructure that was able to scale was a non-trivial process; there are two main issues to address.

One is to make the lowest impact on the performance on the program execution. It is very important to be respectful of the resources used in gathering debug information. To achieve this goal, the authors employ sampling; they also address a technique to conduct fair sampling.

The other one is a craftiness in gathering and periodically sending data to the central system: even collecting a small amount of information have an huge impact on scalability.

The authors then focus on the data analysis phase. They propose three different application with increasing level of sophistication:

- They show how to share the burden of assertion across the use base so to inflict upon each user only a small fraction of the checks.

- They show how to start from a large set of predicates (predicate guessing) and shrinking it down over time to reveal the smallest set that can deterministically predict a bug.

- They show how to use linear regression to isolate non-deterministc bugs; in other words they shrink the set of predicates that has the highest correlation with the failure.

ESC/Java [9] Extended Static Checking for Java

This tool perform static analysis on Java source code looking for errors and warnings to report. It provides specific Java annotation to formally express design decisions. ESC handles and warns about multiple common programming errors, e.g. null dereference, index out of bounds, types errors; it also warns about concurrency errors, like race conditions and deadlock. Aided by the custom annotations, ESC employs an engine to decode the semantics of the program and apply techniques to automatically prove theorems; doing so it is able to report potential bugs that are not detected by the type system and that are detected at runtime.

One core requirement imposed on ESC is the modularity of checking; in other words, it can work on pieces of code (i.e. methods) in isolation to the rest of the code. This restriction was chosen for scalability reasons even if the downside is the need of custom annotations. The authors argue that the cost of using the annotations are not an hard overhead: when developers are engaged in manual code review they need information that usually come from unstructured sources (e.g. natural language comments) and they already sustains the burden of gathering additional knowledge not present in the raw code.

It is evident throughout all the paper the importance the authors put on the tradeoff between the cost of the annnotation process and the benefit of true errors feedback; this sentence taken from the paper's introduction sums the core of this tradeoff: "if the checker finds enough errors to repay the cost of running it and studying its output, then the checker will be cost-effective, and a success". Infact, they enumerates two important features needed by an ideal static checker :

- soundness - if the program has errors, it will find some
- completeness - every error reported is a true positive

ESC do not seek to honor these two features for the very belief quoted above. The authors observe that the alternative processes used to achieve software quality (testing and code reviews) do not possess any of the features of the ideal static checker.

At paper time of writing, ESC was used for two years on multiple kind of programs and was proven effective in finding meaningful bugs. The performance was adequate for interactive use on most methods. Even if it was proven of real usefulness, the users' feedback suggest that the cost of annotating was high and the number of warnings was excessive. It must be said that the annotation were added after the development of the project and not during the evolution of it; this is commonly know by developers to be a dreaded task.
At this point it is unclear if the tool delivers a positive tradeoff between cost and benefit; the experience of the authors during internal use of the tool was encouraging. They believe that ESC is already a valid tool to be used in classrooms: it enforces good design, modularity and verification.

VeriSoft [10] Model checking without a model: An analysis of the heart-beat monitor of a telephone switch using VeriSoft

The goal of VeriSoft is to detect problems in a concurrent reactive system (CRS) through the exploration of its state space. The *reactive* word stands to describe the continuous interaction of the system with the environment. The issues detected are: deadlocks, assert violations, livelocks. A CRS is composed of two parts: a finite number of processes written in arbitrary code (e.g. C, C++, Java, tcl, and so on) and a finite number of communication objects (TCP connections, semaphores, shared memory, and so on). The need of such tool arise from the difficulty of writing a robust and reliable CRS; it is well accepted that concurrent systems are prone to unexpected issues, difficult to track, test and to reproduce.

What VeriSoft does is a systematic state space exploration; it defines the state space as a directed graph: the nodes are the global states and the edges are the transitions between states. It follows that each global state should be uniquely identified and this is one of the core issue that VeriSoft solves with an original combination of algorithms; the author calls it "an efficient state-less search".

The paper reports the analysis conducted with VeriSoft on a software owned by Lucent Technologies: "Heart-Beat Monitor" (HBM). Such software monitors the status of a telephone switch elements and determines their state based on the propagation delays of messages sent through those elements. HBM is an important piece of software because it has a big impact on the switch performance due to its influence on the switch routing.

The experience of the authors is reported as successful: they were able to find errors in the documentation and in the software itself. Subsequently they modified the code to strengthen some properties and tested them again with VeriSoft; after another run, as desired, VeriSoft reported satisfactory results. The development team of HBM decided to integrate the code changes from the authors for the next commercial release.

VeriSoft acts as a scheduler and has complete control over non-determinism so it can reproduce any interesting scenario (i.e. those that during the automatic tests led to errors and issues). One other benefit is that there is no need to describe the model with specific languages: it relies on the exercise of the actual code. One downside is that it cannot detect cycles in the graph of the global spaces and, as such, it can only detect violation of safety properties.

JPFinder [14] Applying Model Checking in Java Verification

Java PathFinder (Jpf) is a prototype translator from the Java language to Promela (Process Meta Language). Promela runs on SPIN (Simple Promela Interpreter). SPIN is a general tool to find concurrency problems and verify the correctness of a system.

It is reported that Jpf is not the first attempt of Java-to-Promela translator; in addition to this other work, Jpf can handle a significant number of features of the Java language. At the same time, many other are missing. The paper describe the issues due the impedance between Java and Promela; they come in two flavors: performance issues and missing feature issues.

Jps provides the programmer with Java static methods to annotate the source code with assertions; those assertions will be checked with the SPIN model checker.

The authors used a Chess game server written in Java as the test subject for finding synchronization bugs. They did not use the original source code but wrote a simplified abstraction in Java composed with 16 classes and roughly 1400 lines of code; it is reported that this Java program was non trivial and the development was done without thinking about formal verification. Then the authors fed the Java semplification to Jpf and were able to find a bug that was later confirmed.

CMC [19] CMC: A pragmatic approach to model checking real code

Model checking is very hard in practice; it usually involves the use of a specific domain language to describe the model and then a model checker. This common approach to model checking is very hard to endure in practice: it exposes the age-old dualism of having two parallel systems. On one hand we have the actual implementation, on the other hand we have the abstraction that represents the model of the implementation. Having two distinct bodies opens the following issues:

- The model could exhibit issues that are not present in the actual implementation.

- The implementation could show bugs that are not present or detected in the model.
- The need to maintain both systems coping with the impedance of two different ways of expressing meaning, behaviour and intent.

Complex systems often hides rare but nasty bugs that arise only after many weeks of continuous run; this is a major issue with such systems. Explicit checkers can help in this scenario; they search a huge state space without wasting the resources for repeated parts of usual testing.

The first contribution reported by the authors is CMC: C and C++ model checker. It works directly on the implementation without the need to create a separate model to be checked. CMC needs some adaptations on the code, some are just good programming practices (e.g. asserts, specifying the environment), other are changes required specifically by CMC: one is for handling the non-determinism and another one is to handle the initialization functions and event handlers. The tool works by directly executing and scheduling the system under analysis. It needs to store and load the whole state space and as such it handles techniques to cope with the *state explosion problem*: simple heuristics help to prune a huge amount of states.

The second contribution is the application of CMC to three implementations of AODV routing protocol. The actual goal of CMC is to check network code implementations, but the ultimate goal is to check a broader range of programs. During the experimentation the authors, through CMC, were able to find 34 errors many of whom were meaningful errors. Actually, their work exposed also a bug with the AODV specification itself (that was later acknowledged in the RFC 3561 citing the first author of this paper).

Network code is of core importance for the stability of a system; it is prone to many issues that undermine correctness, e.g. packet loss, hardware errors, security attacks.

CMC proved to work well, given the results on three different implementation of a routing protocol. For a wider acquisition of CMC it essential for the authors, to lessen the burden of the code adaptation and automate it as much as possible.

2.2 Code Smells and Antipatterns

copy&paste Several techniques have been proposed in the literature to detect code smell instances affecting code components, and all of these take their cue from the suggestions provided by four well-known books: [29], [16], [86], [66].

2.2.1 Four inspirational books description

copy&paste
Webster [86] - todo The first one, by Webster [25] defines common pitfalls in Object Oriented Development, going from the project management down to the implementation.

Riel [66] - todo Riel [66] describes more than 60 guidelines to rate the integrity of a software design

Fowler [29] - todo The third one, by Fowler [29], describes 22 code smells describing for each of them the refactoring actions to take.

Brown et al. [16] - todo Brown et al. [16] define 40 code antipatterns of different nature (i.e., architectural, managerial, and in source code), together with heuristics to detect them.

2.2.2 Last decade proposed approaches

From these starting points, in the last decade several approaches have been proposed to detect design flaws in source code - todo fix copy&paste

Travassos et al. [23] Travassos et al. created a set of techniques for manually identifying defects in order to improve software quality. These practices help individuals to read object oriented code and assess, through a predefined taxonomy, if some problem is present. The authors conducted an empirical study on these techniques and report their feasibility.

van Emden and Moonen [83] todo, fix copy&paste van Emden and Moonen [83] presented jCOSMO, a code smell browser that visualizes the detected smells in the source code. In particular, they focus their attention on two Java programming smells, known as instanceof and typecast. The first occurs when there are too many instanceof operators in the same block of code that make the source code difficult to read and understand. The typecast smell appears instead when an object is explicitly converted from one class type into another, possibly performing illegal casting which results in a runtime error.

Simon et al. [] todo, fix copy&paste Simon et al. [72] provided a metric-based visualization tool able to discover design defects repre- senting refactoring opportunities. For example, a Blob is detected if different sets of cohesive attributes and methods are present inside a class. In other words, a Blob is identified when there is the possibility to ap- ply Extract Class refactoring.

Marinescu [50] todo, fix copy&paste Marinescu [50] proposed a metric-based mechanism to capture deviations from good design principles and heuristics, called "detection strategies". Such strategies are based on the identifica- tion of symptoms characterizing a particular smell and metrics for measuring such symptoms. Then, thresholds on these metrics are defined in order to define the rules

Lanza and Marinescu [44] todo, fix copy&paste Lanza and Marinescu [44] showed how to exploit quality metrics to identify "disharmony patterns" in code by defining a set of thresholds based on the measurement of the exploited metrics in real software systems. Their de- tection strategies are formulated in four steps. In the first step, the symptoms characterizing a smell are defined. In the second step, a proper set of metrics measuring these symptoms is identified. Having this information, the next step is to define thresholds to classify the class as affected (or not) by the defined symptoms. Finally, AND/OR operators are used to correlate the symptoms, leading to the final rules for detecting the smells.

Munro [53] todo, fix copy&paste Munro [53] presented a metric-based detection tech- nique able to identify instances of two smells, i.e., Lazy Class and Temporary Field, in the source code. A set of thresholds is applied to some structural metrics able to capture those smells. In the case of Lazy Class, the metrics used for the identification are Number of Meth- ods (NOM), LOC, Weighted Methods per Class (WMC), and Coupling Between Objects (CBO).

Moha et al. [51] todo, fix copy&paste Moha et al. [51] introduced DECOR, a technique for specifying and de- tecting code and design smells. DECOR uses a Domain-Specific Language (DSL) for specifying smells using high-level abstractions. Four design smells are identified by DECOR, namely Blob, Swiss Army Knife, Functional Decomposition, and Spaghetti Code.

Tsantalis and Chatzigeorgiou [79] todo, fix copy&paste Tsantalis and Chatzigeorgiou [79] presented JDeodor- ant, a tool able to detect instances of Feature Envy smells with the aim of suggesting move method refactoring op- portunities. For each method of the system, JDeodorant forms a set of candidate target classes where a method should be moved. This set is obtained by examining the entities (i.e., attributes and methods) that a method accesses from the other classes. In its current version JDeodorant8 is also able to detect other three code smells (i.e., State Checking, Long Method, and God Classes), as well as opportunities for refactoring code clones.

Ligu et al. [47] todo, fix copy&paste Ligu et al. [47] introduced the identification of Refused Bequest code smell using a combination of static source code analysis and dynamic unit test execution. Their approach aims at discovering classes that really want to support the interface of the superclass [29]. In order to understand what are the methods really invoked on subclass instances, they intentionally override these methods introducing an error in the new implementation (e.g., division by zero). If there are classes in the system invoking the method, then a failure will occur. Otherwise, the method is never invoked and an instance of Refused Bequest is found.

2.2.3 Code smell detection formulated as an optimization problem:

Kessentini et al. [36] todo, fix copy&paste Kessentini et al. [36] as they presented a technique to detect design de- fects by following the assumption that what significantly diverges from good design practices is likely to represent a design problem. The advantage of their approach is that it does not look for specific code smells (as most ap- proaches) but for design problems in general. Also, in the reported evaluation, the approach was able to achieve a 95% precision in identifying design defects [36].

Kessentini et al. [37] todo, fix copy&paste Kessentini et al. [37] also presented a cooperative parallel search- based approach for identifying code smells instances with an accuracy higher than 85%.

Boussaa et al. [13] todo, fix copy&paste Boussaa et al. [13] proposed the use of competitive coevolutionary search to code-smell detection problem. In their approach two populations evolve simultaneously: the first generates detection rules with the aim of detecting the highest possible proportion of code smells, whereas the second population generates smells that are currently not detected by the rules of the other population.

Sahin et al. [68] todo, fix copy&paste Sahin et al. [68] proposed an approach able to generate code smell detection rules using a bi-level optimization problem, in which the first level of optimization task creates a set of detection rules that maximizes the coverage of code smell examples and artificial code smells generated by the second level. The lower level is instead responsible to maximize the number of code smells artificially generated. The empirical evaluation shows that this approach achieves an average of more than 85% in terms of precision and recall.

2.2.4 Non binary classification (smell/clean)

The approaches described above classify classes strictly as being clean or anti-patterns, while an accurate analysis for the borderline classes is missing [40]

Khomh et al. [40] todo, fix copy&paste Khomh et al. [40] proposed an approach based on Bayesian belief networks providing a likelihood that a code component is affected by a smell, instead of a boolean value as done by the previous techniques.

Oliveto et al. [58] todo, fix copy&paste This is also one of the main characteristics of the approach based on the quality metrics and B-splines proposed by Oliveto et al. [58] for identifying instances of Blobs in source code

2.2.5 Usage of historical data for code smells

Ratiu et al. [65] todo, fix copy&paste Ratiu et al. [65] proposed to use the historical information of the suspected flawed structure to increase the accuracy of the automatic problem detection.

Palomba et al. [60] todo, fix copy&paste Palomba et al. [60] provided evidence that historical data can be successfully exploited to identify not only smells that are intrinsically characterized by their evolution across the program history – such as Divergent Change, Parallel Inheritance, and Shotgun Surgery – but also smells such as Blob and Feature Envy [60].

2.3 Self-Admitted Technical Debt

Potdar and Shihab [30] Potdar and Shihab [30] pioneered the study of SATD by mining five software systems to investigate (i) the amount of SATD they contain, (ii) the factors promoting the intro- duction of the SATD, and (iii) how likely is the SATD to be removed after its introduction.

Storey et al. [35] Storey et al. [35] explored the use of task annotations (e.g., TODO, FIXME) in code comments as a mechanism to manage developers tasks. Their findings highlight several activities that are supported by such annotations (e.g., TODOs are sometimes used to communicate by asking questions to other developers).

Guo et al. [18] Guo et al. [18] tracked the lifecycle of a single delayed maintenance task (i.e., a technical debt instance) to study the effect of this decision on the project outcomes. Their data confirm the harmfulness of technical debt, showing that the decision to delay the task resulted in tripled costs for its implementation.

Klinger et al. [21] Klinger et al. [21] interviewed four technical architects at IBM to understand how decisions to acquire technical debt are made within an enterprise. They found that technical debt is often acquired due to non-technical stakeholders (e.g., due to imposed requirement to meet a specific deadline sacrificing quality). Also, they highlight the lack of effec- tive communication between technical and non-technical stakeholders involved in the technical debt management

Kruchten et al. [23] Kruchten et al. [23] built on top of the technical debt definition provided by Cunningham [14] to clearly define what constitute technical debt and provide some theoretical foundations. They presented the "technical debt landscape", classifying the technical debt as visible (e.g., new features to add) or invisible (e.g., high code complexity) and highlighting the debt types mainly causing issues to the evolvability of software (e.g., new features to add) or to its maintain- ability (e.g., defects).

Lim et al. [25] Lim et al. [25] interviewed 35 practitioners to investigate their perspective on technical debt. They found that most of participants were familiar with the notion of technical debt—"We live with it every day" [25]—and they do not look at technical debt as a poor programming practice, but more as an intentional decision to trade off competing concerns during development [25]. Also, practitioners acknowledged the difficulty in measuring the impact of technical debt on software projects. Similarly, Kruchten et al. [24] reported their understanding of the technical debt in industry as the result of a four year interaction with practitioners.
Kruchten et al. [24] Kruchten et al. [24] reported their understanding of the technical debt in industry as the result of a four year interaction with practitioners.

Zazworka et al. [41] Zazworka et al. [41] compared how technical debt is detected manually by developers and automatically by de- tection tools. The achieved results show very little overlap between the technical debt instances manually and auto- matically detected.

Spinola et al. [34] Spinola et al. [34] collected a set of 14 statements about technical debt from the literature (e.g., "The root cause of most technical debt is pressure from the customer" [32]) and asked 37 practitioners to express their level of agreement for each statement. The statement achieving the highest agreement was "If technical debt is not managed effectively, maintenance costs will increase at a rate that will eventually outrun the value it delivers to customers".

Alves et al. [2] Alves et al. [2] proposed an ontology of terms on technical debt. Part of this ontology is the classification of technical debt types derived by studying the definitions existing in the literature.

Maldonado and Shihab [15] Maldonado and Shihab [15] also used the classification by Alves et al. to investigate the types of SATD more diffused in open source projects. They identified 33K comments in five software systems reporting SATD. These comments have been manually read by one of the authors who found as the vast majority of them (∼60%) reported design debt.

Wehaibi et al. [37] Wehaibi et al. [37] analysed the relation between SATD and software quality in five open source systems. Their main findings show that: (i) the defect-proneness of files containing SATD instances increases after their introduction; and (ii) developers experience difficulties in performing SATD-related changes

2.4 TD and machine learning

–this section will reference the paper from CSABA (techdebt conference 2020 related paper)–
(c&p) Several detection techniques and tools have been proposed in the literature. Recently, the adoption of machine learning techniques to detect bad smells became a trend [20]

Khom et al. [27] and [28] Khomh et al. proposed a Bayesian approach which initially converts existing detection rules to a probabilistic model to perform the predictions [27]. Khom et al. [28] extend [27] by the introduction of Bayesian Belief Networks, improving the accuracy of the detection.

Maiga et al. Maiga et al. proposed an SVM-based approach that uses the feedback information provided by practitioners [35, 36]

Amorim et al. [3] Amorim et al. [3] presented an experience report on the effectiveness of Decision Trees for detecting bad smells. They choose these classifiers due to their interpretability [3]. Thus, most of the proposed works focus on only one classifier. They were also trained in a dataset composed of few systems and, consequently, the results may be positive towards their aproach due to overfitting.

Fontana et al. [21] Fontana et al. evaluated different machine learning algorithms on a set of different systems [21]. Their work was later extended and refined, providing a larger comparison of classifiers [20]. The notorious impact of this work was the incredible performance re- ported. Even naive algorithms were able of achieving great results using a small training dataset. This draws attention to possible drawbacks and limitations of their work, which was later reported by Di Nucci et al. [15]

Di Nucci et al. [15] Di Nucci et al. [15]. They replicated the study and verified that the reported performance was highly biased by the dataset and the procedures adopted, such as unrealistic balanced dataset, in which one third of the instances were smelly.

Daniel Cruz at al. –the paper itself- Detecting Bad Smells with Machine Learning Algorithms: an Empirical Study. Bad smells are symptoms of bad design choices implemented on the source code. They are one of the key indicators of technical debts, specifically, design debt. To manage this kind of debt, it is important to be aware of bad smells and refactor them whenever possible. Therefore, several bad smell detection tools and techniques have been proposed over the years. These tools and techniques present different strategies to perform detections. More recently, machine learning algorithms have also been proposed to support bad smell detection. However, we lack empirical evidence on the accuracy and efficiency of these machine learning based techniques. In this paper, we present an evaluation of seven different machine learning algorithms on the task of detecting four types of bad smells. We also provide an analysis of the impact of software metrics for bad smell detection using a unified approach for interpreting the models' decisions. We found that with the right optimization, machine learning algorithms can achieve good performance (F1 score) for two bad smells: God Class (0.86) and Refused Parent Bequest (0.67). We also uncovered which metrics play fundamental roles for detecting each bad smell.

2.5 Summing Up (WAS 2.4)

Here I explain why what I did is different

Chapter 3

Using Deep Learning to Detect Technical Debt

Here I describe the approach.

I'm going to start from a short overview, then I go into details

3.1 Mining SATD Instances and their Fixes

3.2 The Deep Learning Model

3.3 Hyperparameter Tuning

Chapter 4

Empirical Study Design

I’m going to start with a short description about the goal/research questions

4.1 Context Selection

I’ll explain the dataset used for the evaluation

4.2 Data Collection and Analysis

How I compute the results

4.3 Replication Package

A link to a repo with all data and code

Chapter 5

Results Discussion

…results discussion …

5.1 Quantitative Results

Report precision/recall for different confidence thresholds

5.2 Qualitative Results

Discuss interesting cases in which your approach succeeds/fails

Chapter 6

Threats to Validity

. . . Discussion on the limitations of my study . . .

Chapter 7

Conclusion

. . . conclusion . . .

Bibliography

[1] Eric Allman. “Managing technical debt”. In: *Communications of the ACM* 55.5 (2012), pp. 50–55.

[2] Nicolli SR Alves et al. “Identification and management of technical debt: A systematic mapping study”. In: *Information and Software Technology* 70 (2016), pp. 100–121.

[3] Terese Besker et al. “Embracing technical debt, from a startup company perspective”. In: *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2018, pp. 415–425.

[4] William R Bush, Jonathan D Pincus, and David J Sielaff. “A static analyzer for finding dynamic programming errors”. In: *Software: Practice and Experience* 30.7 (2000), pp. 775–802.

[5] Ward Cunningham. “The WyCash portfolio management system”. In: *ACM SIGPLAN OOPS Messenger* 4.2 (1992), pp. 29–30.

[6] Dawson Engler and Ken Ashcraft. “RacerX: effective, static detection of race conditions and deadlocks”. In: *ACM SIGOPS operating systems review* 37.5 (2003), pp. 237–252.

[7] Dawson Engler et al. “Bugs as deviant behavior: A general approach to inferring errors in systems code”. In: *ACM SIGOPS Operating Systems Review* 35.5 (2001), pp. 57–72.

[8] Dawson Engler et al. *Checking system rules using system-specific, programmer-written compiler extensions*. Tech. rep. STANFORD UNIV CA COMPUTER SYSTEMS LAB, 2000.

[9] Cormac Flanagan et al. “Extended static checking for Java”. In: *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*. 2002, pp. 234–245.

[10] Patrice Godefroid, Robert S Hanmer, and Lalita Jategaonkar Jagadeesan. “Model checking without a model: An analysis of the heart-beat monitor of a telephone switch using verisoft”. In: *Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis*. 1998, pp. 124–133.

[11] Yuepu Guo, Rodrigo Oliveira Spínola, and Carolyn Seaman. “Exploring the costs of technical debt management–a case study”. In: *Empirical Software Engineering* 21.1 (2016), pp. 159–182.

[12] Sudheendra Hangal and Monica S Lam. “Tracking down software bugs using automatic anomaly detection”. In: *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*. IEEE. 2002, pp. 291–301.

[13] Reed Hastings and Bob Joyce. *Fast Detection of Memory Leaks and Access Errors. Winter 1992 USENIX Conference*. 1992.

[14] Klaus Havelund and Jens Ulrik Skakkebæk. “Applying model checking in Java verification”. In: *International SPIN Workshop on Model Checking of Software*. Springer. 1999, pp. 216–231.

[15] Konrad Hinsen. “Technical debt in computational science”. In: *Computing in Science & Engineering* 17.6 (2015), pp. 103–107.

[16] Zhenmin Li et al. “CP-Miner: Finding copy-paste and related bugs in large-scale software code”. In: *IEEE Transactions on software Engineering* 32.3 (2006), pp. 176–192.

[17] Ben Liblit et al. “Bug isolation via remote program sampling”. In: *ACM Sigplan Notices* 38.5 (2003), pp. 141–154.

[18] Antonio Martini, Terese Besker, and Jan Bosch. “Technical Debt tracking: Current state of practice: A survey and multiple case study in 15 large organizations”. In: *Science of Computer Programming* 163 (2018), pp. 42–61.

[19] Madanlal Musuvathi et al. “CMC: A pragmatic approach to model checking real code”. In: *ACM SIGOPS Operating Systems Review* 36.SI (2002), pp. 75–88.

[20] Nicholas Nethercote and Julian Seward. “Valgrind: a framework for heavyweight dynamic binary instrumentation”. In: *ACM Sigplan notices* 42.6 (2007), pp. 89–100.

[21] Edith Tom, Aybuke Aurum, and Richard Vidgen. “A consolidated understanding of technical debt”. In: (2012).

[22] Edith Tom, AybüKe Aurum, and Richard Vidgen. “An exploration of technical debt”. In: *Journal of Systems and Software* 86.6 (2013), pp. 1498–1516.

[23] Guilherme Travassos et al. “Detecting defects in object-oriented designs: using reading techniques to increase software quality”. In: *ACM Sigplan Notices* 34.10 (1999), pp. 47–56.

[24] Alberto Villar, Santiago Matalonga, and Montevideo Uruguay. “Definiciones y Tendencia de Deuda Técnica: Un Mapeo Sistemático de la Literatura.” In: *CIbSE*. 2013, pp. 29–42.

[25] Bruce F Webster. *Pitfalls of object-oriented development*. M & T Books, 1995.

[26] Pin Zhou et al. “AccMon: Automatically detecting memory-related bugs via program counter-based invariants”. In: *37th International Symposium on Microarchitecture (MICRO-37’04)*. IEEE. 2004, pp. 269–280.