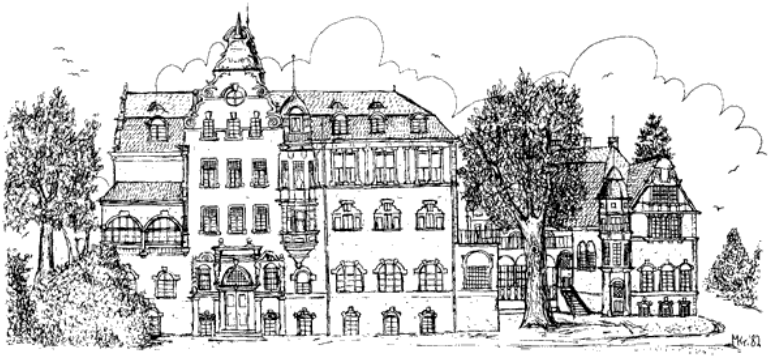
25. Workshop Software-Reengineering und -Evolution

WSRE 2023

der GI-Fachgruppe Software-Reengineering (SRE)

8.-10. März 2023

Physikzentrum Bad Honnef



25. Workshop Software-Reengineering und -Evolution

der GI-Fachgruppe Software Reengineering (SRE)

Die Ausnahmezeit der Corona-Pandemie wirft weiterhin ihre Schatten auf den traditionsreichen WSRE: Nachdem wir im Vorjahr wieder regulär im Mai in Bad Honnef zusammenkommen konnten, hat es uns dieses Jahr wegen nachzuholender Veranstaltungen der Deutschen Physikalischen Gesellschaft (DPG) in den März verschlagen. Trotz aller Unruhe und organisatorischer Herausforderungen ist es (hoffentlich) wieder gelungen, ein interessantes Programm zusammenzustellen. Schließlich ist dies das 25. Treffen dieser Art, das es gebührend zu feiern gilt! Als der WSRE (damals noch WSR) 1999 von Jürgen Ebert und Franz Lehner ins Leben gerufen wurde, hätte vermutlich niemand mit einer solchen Kontinuität und Erfolgsgeschichte gerechnet.

Ziel war es damals, ein deutschsprachiges Diskussionsforum zu allen Aspekten rund um das Thema Reengineering zu schaffen. Durch aktive und gewachsene Beteiligung vieler Personen aus Forschung und Praxis hat sich der WSRE inzwischen als zentrale Reengineering-Konferenz im deutschsprachigen Raum etabliert. Viele Teilnehmer*innen haben ihn als Student*in oder Doktorand*in kennen und schätzen gelernt und bleiben ihm auch in ihrem späteren Berufsleben treu. Dabei wird der Workshop weiterhin als Low-Cost-Workshop ohne eigenes Budget durchgeführt. Bitte tragen Sie dazu bei, den WSRE weiterhin erfolgreich zu machen, indem Sie interessierte Kolleg*innen und Bekannte darauf hinweisen.

Auf Basis der erfolgreichen WSR-Treffen der ersten Jahre wurde 2004 die GI-Fachgruppe Software-Reengineering (https://fg-sre.gi.de/) gegründet, deren Leitungsgremium den WSRE seitdem organisiert und auch bei anderen Aktivitäten rund um das Thema Reengineering mitwirkt. Als GI-Mitglied laden wir Sie ein, der Fachgruppe beizutreten, um dieses Thema zu stärken.

Die Themen des WSRE erstrecken sich auf die Bereiche Software-Reengineering, Software-Wartung und -Evolution. Darunter verstehen wir prinzipiell alle Aktivitäten rund um die Analyse, Bewertung, Visualisierung, Verbesserung, Migration und Weiterentwicklung von Software-Systemen. Im Vordergrund steht der Austausch zwischen Interessierten, insbesondere auch der Austausch zwischen Forschung und Praxis. Aus dem WSRE sind in den vergangenen 25 Jahren viele Kooperationen zwischen Forscher*innen, zwischen Forscher*innen und Praktiker*innen, aber auch unter Praktiker*innen hervorgegangen. Viele Beiträge des WSRE erzählen von diesen Erfolgsgeschichten.

Beim diesjährigen WSRE sind folgende Programmpunkte vorgesehen:

- **Vorträge**: Einblick in sowie Rückblick und Ausblick auf interessante Arbeiten und Ergebnisse rund ums Software-Reengineering.
- **Best Student Paper Award**: Wir prämieren den besten studentischen Beitrag (Paper und Vortrag), bewertet durch eine Jury aus Wissenschaft und Praxis.
- **Keynote**: Elmar Jürgens (CQSE GmbH, München) spricht über Softwareanalyse: „Vom Wiegen allein wird die Sau nicht fett – Was bedeutet das für unsere Analysewerkzeuge?“
- **Impulsvorträge und Podiumsdiskussion** zum Thema „The Future of Reengineering“: Wir beschäftigen uns mit der Frage, welche aktuellen Trends für die Disziplin Software-Reengineering von besonderer Bedeutung sind und welche Themen aus Sicht der Forschung und/oder Praxis wichtig sein werden.
- **SREBOK**: Erstellung einer Wissensbasis zum Software-Reengineering. Dabei sind alle herzlich eingeladen, Beiträge zu leisten.
- **Fachgruppensitzung** der GI-Fachgruppe Software-Reengineering.
- **Networking**: Vernetzung zwischen den Teilnehmenden in den Pausen und beim gemütlichen Zusammensein am Abend.
- **Social Event**: Erkundung der örtlichen Weinkultur in und um Bad Honnef.
- **Virtuelle „Hall of Fame“**: Anlässlich des Jubiläums möchten wir verdiente Workshopteilnehmer besonders ehren.

Die Organisatoren danken allen Beitragenden für ihr Engagement – insbesondere den Autor*innen, den Vortragenden sowie den Teilnehmer*innen und Juroren des Best Student Paper Awards und allen Workshop-Teilnehmer*innen für die lebhaften, kontroversen und interessanten Diskussionen. Vielen Dank auch an das Team des Physikzentrums Bad Honnef, das im Hintergrund dafür sorgt, dass wir hier drei angenehme Tage verbringen können. Insbesondere danken wir auch den Sponsoren des Best Student Paper Awards: Caruso GmbH, Ismaning; Delta Software Technology GmbH, Schmallenberg; itemis AG, Lünen und S&N Invent GmbH, Paderborn, die es durch ihre finanzielle Unterstützung ermöglichen, den Finalist*innen einen Reisekostenzuschuss zu gewähren und für den besten Beitrag ein Preisgeld auszuloben.

Für die Fachgruppe Software-Reengineering:
Jochen Quante, Bosch Research (Sprecher)
Marco Konersmann, RWTH Aachen
Stefan Sauer, Universität Paderborn
Daniela Schilling, Delta Software Technology
Sandro Schulze, TU Braunschweig

Von JOBOL zu JAVOL – Refaktorisierung migrierter Java-Programme

Nils Bauer, Christian Becker, Uwe Erdmenger, Felix Graßler, Denis Uhlig
pro et con Innovative Informatikanwendungen GmbH, Reichenhainer Straße 29a, 09126 Chemnitz
nils.bauer,christian.becker,uwe.erdmenger,felix.grassler,denis.uhlig@proetcon.de

Abstract

Die Firma pro et con realisiert Softwaremigrationsprojekte, bspw. von COBOL nach Java, als 1:1-Migration. Die konvertierten Programme enthalten i. d. R. COBOL-typische und kundenspezifische Codemuster, welche sich im Laufe der Wartung als störend herauskristallisieren können. Dieser Bericht beschreibt Werkzeuge zur Refaktorisierung solcher Muster, welche in einer separaten Phase am Ende eines Migrationsprojektes oder auch Jahre später noch eingesetzt werden können.

1 Motivation

pro et con realisiert Softwaremigrationsprojekte toolgestützt mit der eigenentwickelten *pecBOX*, einem Werkzeugkasten, der verschiedene Migrations- und Metawerkzeuge sowie Codegeneratoren zusammenfasst. Eines der Migrationswerkzeuge ist der COBOL to Java Converter (*CoJaC*), welcher einen Automatisierungsgrad von über 98% erreicht und bereits in mehreren, kommerziellen Migrationsprojekten zum Einsatz kam. Nach Projektabschluss werden die migrierten Java-Programme gewartet und weiterentwickelt, woran pro et con zum Teil beteiligt ist. Erfahrungen aus verschiedenen Projekten und der Wartung migrierter Systeme haben gezeigt, dass der Quelltext zwar gut lesbar ist, allerdings migrationsbedingt dennoch Elemente enthält, welche in Java unüblich sind und die Lesbarkeit erschweren können. Diese sind:

1. technischer Code, der für die korrekte Funktionsweise notwendig ist,
2. COBOL-typische Codemuster, welche durch die Ursprungssprache vorgegeben sind,
3. kundenspezifische Programmiermuster, basierend auf dem Stil der ursprünglichen Entwickler.

Punkt 1 kann durch eine *technische* Refaktorisierung verbessert werden. Dazu zählt bspw. die Umstellung von migrationsbedingt langen Zugriffsketten auf verkürzte Bezeichner [1]. Hierbei erfolgt keine inhaltliche Veränderung der Programme und der Programmlogik. Die geänderten Programme sind vor und nach der Refaktorisierung im Prinzip identisch.

Bei den folgenden beiden Punkten handelt es sich dagegen um potentiell *inhaltliche* Verbesserungen. Die Punkte 2 und 3 werden ebenso wie Punkt 1 in einer separaten Reengineering-Phase nach der Migration bearbeitet.

Allerdings muss diese nicht direkt im Anschluss an die Migration erfolgen. Die Entwickler des Kunden benötigen erfahrungsgemäß eine gewisse Einarbeitungszeit, um sich an den migrierten Code und den neuen Entwicklungsprozess nach der Migration zu gewöhnen. Erst mit steigender Erfahrung im Laufe der Wartung kristallisieren sich häufig Codemuster heraus, die neben einer verbesserten Les- und Wartbarkeit vor allem wegen einer besseren Akzeptanz durch Java-Entwickler überarbeitet werden sollten.

Die Erkenntnisse aus verschiedenen Migrationsprojekten fließen stetig als Verbesserung in die Migrationswerkzeuge ein, wovon jedoch die Kunden bereits abgeschlossener Projekte i. d. R. nicht profitieren. Aus diesem Grund wurde eine Reihe von Refaktorisierungswerkzeugen entwickelt, welche den migrierten Code von bereits abgeschlossenen Migrationsprojekten unter Beachtung von COBOL-typischen und kundenspezifischen Codemustern automatisiert verbessern können. Sie arbeiten unabhängig voneinander, sind optional und können je nach Kunde und dessen Anforderungen beliebig miteinander kombiniert werden.

2 Arbeitsweise und Architektur

Der Ablauf einer Refaktorisierung findet in mehreren Schritten statt. Als erstes wird ein Java-Package oder ein vollständiges Java-Projekt mit allen darin befindlichen Java-Sourcen analysiert. Mit Hilfe des Eclipse Java Development Toolkit (JDT) wird für jedes Sourcefile ein Java-Syntaxbaum erstellt. Dabei werden die zu refaktorisierenden Klassen in den Syntaxbäumen selektiert. Dies geschieht anhand von zuvor definierten Kriterien, bspw. einer bestimmten Elternklasse, des Typs oder eines implementierten Interfaces. Im nächsten Schritt wird der Syntaxbaum mit dem Visitor-Pattern traversiert. Einzelne Java-Statements (als Teilbäume mit Knoten) werden in Ausdrücke zerlegt. Es wird geprüft, ob ein solcher Teilbaum einem bestimmten Muster entspricht. Als Beispiel dient die von COBOL nach Java migrierte Anweisung `someVar.compute(someVar.getValue()+1)` zum Inkrementieren einer Variablen um den Wert 1. Deren Teilbaum wird wie folgt ausgewertet:

```
1|  tree == expression1.call(parameter)
2|  parameter == sum(expression1.call2, '1')
3|  call  == 'compute'
4|  call2 == 'getValue'
```

In Zeile 1 wird geprüft, ob der Teilbaum dem Muster `expression1.call(parameter)` entspricht. Die Zeilen 2 bis 4 enthalten weitere Prüfungen, ob die restlichen Ausdrücke dem gesuchten Muster entsprechen. Sind alle Bedingungen erfüllt, dann ist ein relevanter Kandidat für eine Refaktorisierung gefunden. Die Nutzung von Syntaxbäumen bietet den Vorteil, dass Kontextinformationen in einer strukturierten Form nutzbar sind, wodurch falsch positive Kandidaten minimiert werden können. Es entsteht so eine Liste von zu refaktorisierenden Kandidaten und aller vorzunehmenden Änderungen in den Sourcen. Letztere können Ergänzungen, Verschiebungen, Löschungen oder sonstige Umformungen sein. Im letzten Schritt erfolgt die Änderung durch Bildung neuer Teilbäume:

```
1|  call3 = 'increment'
2|  tree  = expression1.call3
```

Für das obige Beispiel ergibt sich nach der Refaktorisierung das folgende Ergebnis: `someVar.increment()`. Jede dieser Umformungen ist als separates Modul konzipiert, so dass verschiedene, aufeinander aufbauende Kombinationen möglich sind.

3 Anwendung in bestehendem Projekt

Die Refaktorisierungswerkzeuge wurden u. a. auf ein bereits erfolgreich abgeschlossenes COBOL-Java-Migrationsprojekt mit 210 Programmen und 1.400.000 Lines of Code (Java) angewendet, bei dem pro et con an der Wartung und Weiterentwicklung beteiligt ist. Daraus ergaben sich in den letzten Jahren neue Erfahrungen bezüglich der Codemuster, die aufgrund der Ursprungssprache COBOL oder dem Programmierstil der originalen Entwickler aus Java-Sicht schwer lesbar bzw. unüblich sind. Nachfolgend zwei Beispiele dazu:

Negation in Schleifenbedingungen (COBOL-typisch): Schleifenbedingungen werden in COBOL in einer negierten Form angegeben (s. Zeile 1 `PERFORM`):

```
1|  PERFORM UNTIL a > b
2|    COMPUTE a = a + 1
3|  END-PERFORM
```

Durch die 1:1-Migration entsteht folgender Java-Code:

```
1|  while(!(a > b)) {
2|     a.compute(a.getValue()+1);
3|  }
```

Die Negation der Bedingung `!(a > b)` ist unnötig schwer lesbar und für Java-Code unüblich. Das Refaktorisierungswerkzeug analysiert die Vorkommen solcher Codemuster und entfernt die Negation durch Umstellung des inneren Operators >. Das Ergebnis (inkl. des Inkrement-Beispiels) sieht wie folgt aus:

```
1|  while(a <= b) {
2|     a.increment();
3|  }
```

Diese relativ einfache Umformung sorgt bereits für eine bessere Lesbarkeit und erleichtert das Codeverständnis solcher häufig vorkommenden Schleifenbedingungen, da die unnötige Negation entfällt. Im Beispielprojekt wurden ca. 1.700 Schleifenbedingungen und 9.000 Increment-Anweisungen umgeformt.

Zuweisung 88er Stufen (kundenspezifisch): 88er Stufen in COBOL sind sogenannte Bedingungsnamen mit einem bestimmten Wert. Diese können in `IF`-Anweisungen anstelle von `Magic Numbers` genutzt werden:

```
1|  01 SYSTEM-AKTIV PIC X.
2|   88 AKTIV       VALUE 1. * siehe IF
3|
4|  MOVE 1 TO SYSTEM-AKTIV   * Variante 1
5|  SET AKTIV TO TRUE        * Variante 2
6|
7|  IF AKTIV THEN ...
```

Im obigen Beispiel besitzt die Variable `SYSTEM-AKTIV` (Zeile 1) den Bedingungsnamen `AKTIV` (Zeile 2). Der Wert von `SYSTEM-AKTIV` kann über `MOVE` (Zeile 4) oder `SET` (Zeile 5) zugewiesen werden. Hat `SYSTEM-AKTIV` den Wert 1, dann ist die Bedingung im `IF` in Zeile 7 erfüllt. Das Setzen von Werten für Bedingungsnamen wie in Zeile 4 ist fehleranfällig, da auch ungültige Werte zugewiesen werden können. Diese Variante ist auch schlechter lesbar, da die Entwickler ggf. nicht wissen, was `MOVE 1 TO SYSTEM-AKTIV` für eine fachliche Bedeutung hat. Die Variante 2 ist eigentlich bereits in COBOL zu bevorzugen, wird jedoch nicht immer verwendet. Nach der 1:1-Migration entsteht für das aktuelle Beispiel folgender Java-Code:

```
4|  systemAktiv.setValue(1); // Variante 1
5|  systemAktiv.setAktiv();  // Variante 2
6|
7|  if( systemAktiv.isAktiv() ) { ...
```

Die Refaktorisierungswerkzeuge unterstützen die Umformung der Variante 1 in die Variante 2. Dies ist ein komplexerer Anwendungsfall, da hier sowohl der Typ der Variable `systemAktiv` als auch alle Zuweisungen (`systemAktiv.setValue(...)`) und alle zugewiesenen Werte mit den möglichen Bedingungsnamen (`setAktiv()`) abgeglichen werden müssen. Dieses Muster ist kundenspezifisch und die Anzahl der Vorkommen hängt vom Programmierstil der ursprünglichen COBOL-Entwickler ab. Im Beispielprojekt wurden ca. 900 Stellen refaktorisiert.

Neben diesen Beispielen werden bisher insgesamt 13 einfachere und komplexere Refaktorisierungen unterstützt, welche sich optional je nach Kundenwunsch anwenden lassen. Der Test erfolgt äquivalent zum Test regulärer Erweiterungen im Rahmen der Wartung. Der Kunde nutzt hierfür eine Testsuite, die i. d. R. im Rahmen eines Migrationsprojektes entsteht und im Laufe der Wartung erweitert wird.

4 Schlussfolgerungen

Die Erfahrungen aus den letzten Migrationsprojekten zeigen, dass die Optimierung der migrierten Programme eine wichtige Phase in einem Migrationsprojekt darstellt. Dabei stehen neben offensichtlichen Gründen wie der Verbesserung der Les- und Wartbarkeit der Quelltexte auch die Steigerung der Akzeptanz durch neue Entwickler im Vordergrund. Die Refaktorisierungswerkzeuge, welche die Sourcen inhaltlich ändern, können auch Jahre nach dem Migrationsprojekt und mit steigender Erfahrung der Entwickler angewendet werden. Dadurch ist es auch möglich, dass Kunden bereits abgeschlossener Projekte von den in nachfolgenden Projekten gewonnenen Erfahrungen und daraus resultierenden Verbesserungen der Migrationstechnologie profitieren. Der Aufwand für die manuelle Umsetzung solcher Refaktorisierungen in dieser Größenordnung und Häufigkeit in großen Projekten wäre i. d. R. wirtschaftlich nicht gerechtfertigt. Projekte dieser Größenordnung sind nur mit Hilfe automatisierter Werkzeuge möglich.

Literaturverzeichnis

[1] Graßler, F.; Uhlig, D.: Software-Migration ist keine Endstation. Softwaretechnik-Trends, Band 40, Heft 2, August 2020

Generierte Anwendung modernisieren – Reenginering für Fortgeschrittene?

Daniela Schilling
dschilling@delta-software.com
Delta Software Technology GmbH

Abstract

Generatoren haben in der Software-Entwicklung eine lange Historie. Bis heute werden Generatoren aus den 1970er und 80er Jahren eingesetzt, um große und komplexe Anwendungen zu erzeugen. Durch ihre lange Lebensdauer haben auch generierte Anwendung Modernisierungsbedarf. Allerdings ist es hierfür notwendig, Reengineering-Methoden auf den Generator anzuwenden, statt auf die generierte Anwendung, wodurch neue Herausforderungen an den Ansatz entstehen.

1 Bekanntes Problem, andere Ebene

In den 1970er und 80er Jahren kamen einige Generatorsysteme für COBOL auf den Markt. Mit diesen Systemen haben vor allem Banken und Versicherungen ihre großen und komplexen Kernanwendungen entwickelt und pflegen diese Anwendung unter Nutzung der Generatorsysteme bis heute.

Die Anwendungen verrichten immer noch zuverlässig ihren Dienst. Allerdings wurden sie im Laufe der Zeit von Entwickler zu Entwickler weitergeben, an neue Anforderungen angepasst und für neue Aufgaben erweitert. Und so gilt für diese Anwendungen das Gleiche, was auch für native Anwendungen nach vielen Jahrzehnten im Einsatz gilt:

- über die Zeit haben sich technische Schulden angesammelt, die die Wartung des Generators erschweren,
- Laufzeit und Performance entsprechen nicht mehr den Anforderungen
- Teile der Anwendung sollen neugeschrieben oder in Standardprodukte ausgelagert werden, was jedoch die aktuelle Architektur der Anwendung nicht zulässt.

Kurz gesagt ein Reengineering ist erforderlich. Damit der Generator auch weiterhin eingesetzt werden kann, reicht es jedoch nicht aus das Reengineering auf der generierten Anwendung auszuführen, stattdessen muss der Input für den Generator modernisiert werden.

2 Beispiel Delta ADS

Seit Mitte der 1970er Jahre bietet Delta das Generatorsystem ADS für die Generierung von COBOL- und PL/I-Anwendungen. Das System setzt sich zusammen aus Prozessoren und Standard-Macros. ADS bietet eine DSL zur Spezifikation der Kundenanwendungen. Programme spezifizieren die Abläufe und die Anwendungslogik, mittels Macros können wiederkehrende Anteile spezifiziert werden.

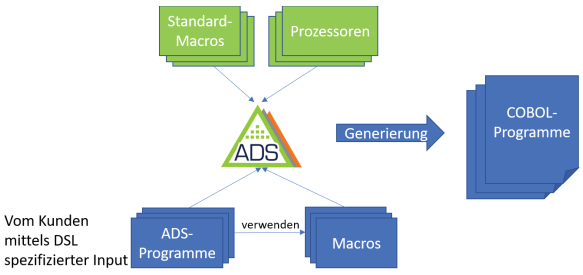


Figure 1: Delta ADS

Bis heute werden mit ADS mehr als 100 Mio Lines of Code generiert.

3 Herausforderung Reengineering von Generatoren

Beim Einsatz von Generatoren gilt häufig, dass der Input für den Generator eine möglichst hohe Les- und Wartbarkeit aufweisen soll, während der generierte Code eher in Bezug auf Performance etc. optimiert wurde. Dennoch kann es bei einem Reengineering notwendig werden, auch den generierten Code zu betrachten. Soll der Generator aber weiterhin eingesetzt werden, reicht es nicht aus, nur den generierten Code zu überarbeiten. Denn, bei einer erneuten Generierung würden alle Änderungen verloren gehen.

Welche Anteile betrachtet werden müssen, hängt ganz entschieden von der Aufgabe ab. Hat die Effizienz bei der Wartung nachgelassen, so muss der Generator-Input überarbeitet werden. Besonders spannend ist jedoch der Fall, dass die generierten Programme nicht mehr den Anforderungen entsprechend und einem Redesign unterzogen werden sollen. In diesem Fall gilt es, die Programme zu analysieren und die Ergebnisse auf den Generator-Input zurück zu führen, um dort die notwendigen Änderungen vorzunehmen.

4 Gemeinsam zum Ziel

Um eine generierte Anwendung modernisieren zu können, ist unterschiedliches Wissen erforderlich. Zum einen muss bekannt sein, welche Modernisierungen vorgenommen werden sollen und an welchen Stellen der generieten Anwendung diese beispielhaft anzuwenden sind. Zum anderen wird Wissen über den Generator benötigt, insbesondere darüber welche Artefakte für die Generierung von bestimmtem Code verantwortlich ist und wie ggf. für die Generierung notwendige Parameter belegt waren.

4.1 Modernisierung einer generierten Anwendung

Eine Kunde möchte seine mit Delta ADS generierte COBOL-Anwendung modernisieren. Dazu haben der Kunde und Delta ein gemeinsames Vorgehen entwickelt.

Der Kunde hat das notwendige Fachwissen und legt fest, was modernisiert werden soll. Beispiel: Im generierte Programm P1 sollen die Zeilen 27-40 überarbeitet werden. Im ersten Schritt ist zu prüfen, aus welchen Artefakten die Zeilen generiert wurden. ADS erzeugt dafür bei der Generierung Back-Referenzen, d.h. Informationen im generierten Sourcecode, aus welchem Artefakt und welcher Zeile die jeweilige Zeile generiert wurde. Damit kann die relevante Stelle gefunden werden. Delta hat, als Hersteller von ADS, zusätzlich die Möglichkeit auch die Belegung der Generierungsparameter auszugeben. Im Beispiel stammen die Zeilen aus dem Macro M1.

Aber Achtung: Macros dienen der Wiederverwendung, d.h. sie können beliebig oft zur Generierung angezogen werden, sowohl innerhalb des Programms P1 als auch in weiteren Programmen. Bevor also das Macro modifiziert werden kann, muss festgestellt werden, welche weiteren Codestellen und Programme von der Änderung betroffen sind. Um dies feststellen zu können, wurde ADS so erweitert, dass bei der Generierung für jedes Macro festgehalten wird, für welche Programme das Macro verwendet wird und welche

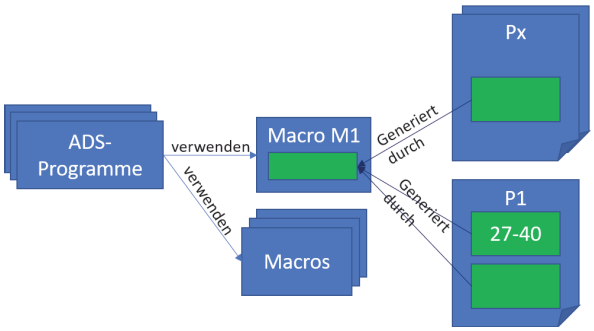


Figure 2: Woher stammt der Code? Und welche weiteren Programme sind betroffen?

Zeilen aus dem jeweiligen Programm durch das Macro erzeugt werden. Damit diese Aussage vollständig ist, muss die gesamte Anwendung generiert werden.

Dann gilt es festzustellen, welche Auswirkung eine Änderung des Macros auf alle anderen Codestellen und Programme hat. Wenn sichergestellt wurde, dass die Änderung auch für alle anderen Stellen unkritisch ist, kann das Macro geändert und die Anwendung neu generiert werden.

5 Generierte Anwendung modernisieren – Reengineering für Fortgeschrittene!

Auch generierte Anwendungen müssen modernisiert werden. Die Besonderheit liegt nun aber darin, dass es nicht ausreicht die Anwendung zu überarbeiten, sondern die Änderungen am Generator selbst vorzunehmen, damit dieser auch weiterhin eingesetzt werden kann.

Ist der Hersteller des Generatorsystems in den Modernisierungsprozess involviert, so besteht die Möglichkeit den Generator so anzupassen, dass alle notwendigen Informationen bei der Generierung erzeugt werden. Eine Modernisierung generierter Anwendungen verläuft in diesem Fall zwar anders als bei einer manuell codierten Anwendung, ist aber eine machbare Aufgabe.

Was aber, wenn der Hersteller nicht involviert ist? Ein Vorteil von Generatoren besteht darin, dass sie Code immer gleich erzeugt. Wie kann man das ausnutzen? Lassen sich Methoden wie Mustererkennung oder Clone-Detection so anpassen, dass man mit ihnen nicht nur gleiche oder ähnliche Codestellen in der generierten Anwendung aufdecken kann, sondern auch im Generator-Input? Welche anderen Reengineering-Methoden könnten angepasst wiederverwendet werden? Es scheint, die Modernisierung von generierten Anwendungen ist Reengineering für Fortgeschrittene!

Towards a Model-Based Software Reengineering Approach with Explicit Behavior Descriptions: Chances and Challenges

Marco Konersmann, Bernhard Rumpe
Software Engineering
RWTH Aachen University, Germany
http://www.se-rwth.de/

Introduction Model-based software reengineering (SRE) uses a horse-shoe process style to modernize original systems [4]. A model of the original system is constructed, adapted, and the target system code is partly or as a whole generated. These translations can introduce faults and quality issues. When the model is incorrect with respect to the original system, the target system might have missing functionality or bad quality. We can compare the target system to the original system, but this does not show error sources or how to fix them: is the issue in the model extraction, the model adaptation, or the code generation?

A major kind of models used in model-based SRE are software architecture (SA) models. SA is often expressed as informal boxes and lines, alongside textual documentation for communication. While this provides a large flexibility for the modelers to communicate, these models cannot be processed automatically for analysis or construction of the system. The Unified Modeling Language (UML) is broadly used for SA and design. It provides a rich syntax for modeling architectures and some degree of formality for automated processing. However, the mapping of UML elements to an implementation or a runtime behavior is not clearly specified. E.g., there are many potential implementations of UML components or their interconnection, depending on the technical domain and intended level of abstraction.

Several approaches exist for creating suitable architecture models of a system, e.g., using static or dynamic analysis [2]. However, since there is no universally accepted SA language [6], reuse is limited to related projects. Using a modular SA language with an explicit behavior description, that is suitable for the diverse concerns of SA, can help increasing reuse. In this paper we sketch a reengineering process utilizing a model-based approach with explicit behavior descriptions, and discuss the chances and challenges.

Horse-Shoe Model using Software Architecture Models with Behavior Descriptions Figure 1 shows a model-based SRE process model that allows for simulation due to SA models with explicit behavior descriptions. The original system comprises the code, run time data such as logs and performance annotations, and further artifacts, such as design decision documents and their rationale. First, we create an architectural model of the original system alongside trace links from the code to the model elements. The approach requires an architecture modeling language that defines behavior of components and their interconnection, and the data processing within the components, in a processable way. The architecture model is used to simulate the system and the quality properties are tuned until the simulation results resemble the original run time data sufficiently. It can then be adapted to resemble the intended target architecture, and simulated to evaluate the run time quality compared to the architecture model of the original system. Then, code is generated from the target architecture model. The process maintains trace links in each step. Therefore the relation of original system artifacts to the target system can be traced. The SRE process might introduce the need to update or refactor these artifacts, e.g., when non-architectural code needs to be adapted or design decisions need to be revised.

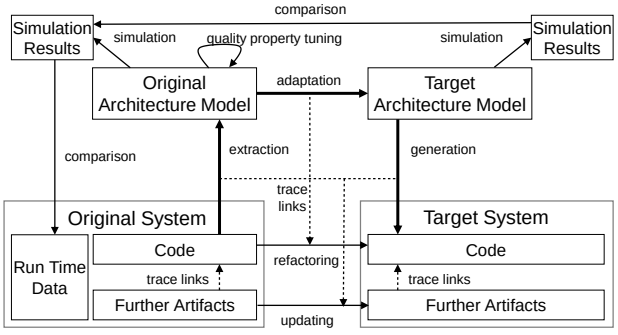


Figure 1: Horse shoe model for software reengineering with model-based software architecture

Chances We see the following chances in applying the approach: First, using a model-based approach allows to automate the SRE process: This includes to automate the creation of architecture models from code, model transformations, and code generation. We require languages that provide a clear interpretation of their behavior description, that holds beyond individual projects or even organizations. They therefore allow to define reusable transformations, e.g., to

replace function calls with event-based communication or to split or merge components. These reusable transformations can be provided alongside the architecture language and reused in multiple SRE projects.

Second, an explicit behavior description enables the simulation of the modeled architecture, for comparing it against the original system. This can increase the quality of the target architecture model simulation.

Third, an architecture modeling language with explicit behavior descriptions allows to analyze the data and control flow through the modeled system. This can help understanding the original system and how to improve the architecture for a target system.

Finally, as the process builds trace links between the code and the architecture model, trace links of existing other artifacts to the original code can be translated into trace links between these artifacts and the target code. These can be used to reference artifacts that need to be updated in the SRE process.

Challenges There are some challenges regarding the approach, that we need to address: First, architectures today have to cope with a large diversity, e.g., in styles, platforms, or implementation languages, which is one of the reasons why no SA language exists, that is suitable for all projects [6]. The diversity also imposes a large complexity on the automated model extraction. Hence, an implementation of this approach needs to carefully choose the architecture language. An implementation should take into account that architectural concepts and technologies evolve. We plan to define fine-grained requirements for architectural languages, model-extraction and code-generation mechanisms to enable efficient reuse or adaption of existing languages and tools.

Second, the simulation of architectures requires a suitable level of abstraction. Finding the right level of abstraction depends on the purpose of the modeling, and thus on the purpose of the reengineering project. E.g., if the purpose is only to wrap the original system in new interfaces, then the level of abstraction might be higher, than if an original system should be reimplemented in a different paradigm. This requires the architecture language, the model extraction mechanism, and the code generator to handle different levels of abstraction, e.g., projects in a repository, classes, or deployments, within the same reengineering project.

Third, creating an architecture model with the quality properties of the original system requires explicit behavior descriptions and information about the modeled system. Among others, this depends on the quality properties to validate, the platform, and the code. One possibility to tune the quality parameters of the model is the use of reinforcement learning.

Fourth, creating traceability is a challenge in all but trivial projects. In practice, trace links require considerable effort to maintain [1]. Automated model extraction and code generation can automatically create trace links. Automation can use naming conventions or other external knowledge to improve the degree of automation for further artifacts.

Finally, refactoring existing, non-architectural code and updating other artifacts automatically is a challenge. As code is formal, elements can be referenced and translated using automated processing. Other artifacts—e.g., design decisions documents—are less formal. Hence there is a greater challenge for automated updates. The presented approach can point to documents or parts therein to reconsider by following the trace links, to decrease the manual effort.

To evaluate the functional suitability, we plan to apply a demonstrator to information system use cases. While we consider the approach domain-agnostic, the extraction mechanisms and code generator need to cover domain-specific frameworks and coding styles. We propose to use MontiArc [3] as a SA language for this approach, because it has explicit behavior descriptions, allows to model different levels of abstractions, an can easily be extended with domain-specific refinements. We can then provide libraries and extensions to represent common technologies and concepts. Codeling [5] is a tool for architecture model extraction and code generation, which considers diversity in the technologies and concepts of architectures. Codeling automatically creates trace links when extracting architecture models. Further trace links are considered an input to this approach, meaning that better trace links can lead to better results. We plan to adapt Codeling to handle MontiArc models and to handle diversity within single projects.

Conclusion We presented an approach for software reengineering using architecture models with explicit behavior descriptions, and discussed its chances and challenges. Future work can include specifying a reference architecture for the approach and providing an implementation to showcase the functional suitability.

References

[1] Jane Cleland-Huang, Brian Berenbach, Stephen Clark, Raffaella Settimi, and Eli Romanova. Best practices for automated traceability. *Computer*, 40(6):27–35, June 2007.

[2] Stephane Ducasse and Damien Pollet. Software architecture reconstruction: A process-oriented taxonomy. *IEEE Transactions on Software Engineering*, 35(4):573–591, 2009.

[3] Arne Haber. *MontiArc - Architectural Modeling and Simulation of Interactive Distributed Systems.* Aachener Informatik-Berichte, Software Engineering, Band 24. Shaker Verlag, September 2016.

[4] R. Kazman, S.G. Woods, and S.J. Carriere. Requirements for integrating software architecture and reengineering models: Corum ii. In *Proceedings Fifth Working Conference on Reverse Engineering (Cat. No.98TB100261)*, 1998.

[5] Marco Konersmann. *Explicitly Integrated Architecture - An Approach for Integrating Software Architecture Model Information with Program Code.* phdthesis, University of Duisburg-Essen, March 2018.

[6] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Trans. Software Eng.*, 26(1):70–93, 2000.

Multi-Modell-Wissensgraph zur niederschwelligen datengestützten Entscheidungsunterstützung bei der Identifizierung von unwirtschaftlicher Variabilität

Vasil L. Tenev Raphael Martin
Fraunhofer IESE, Kaiserslautern
{vasil.tenev, raphael.martin}@iese.fraunhofer.de

Zusammenfassung

Die Realisierung konfigurierbarer Systeme erfordert einen zusätzlichen Aufwand auf allen Stufen des Lebenszyklus, von der Entwicklung bis hin zur Wartung. Daher muss bei der Umgestaltung von änderungsintensiven Systemen auch die Wirtschaftlichkeit berücksichtigt werden. Die für diese Evaluierung erforderlichen Daten sind jedoch oft unvollständig und müssen aus unterschiedlichen Quellen extrahiert, aufbereitet und analysiert werden.

Um diese Herausforderungen anzugehen, wird ein Ansatz zum Neugestalten bestehender Daten und Werkzeugketten vorgestellt. Der Kern dieses Ansatzes ist ein integratives Wissensmodell, das es ermöglicht, bestehende sowie neue Analysewerkzeuge anzubinden und dadurch die benötigten Auswertungen zu ermöglichen und eine faktenbasierte Entscheidungsunterstützung bereitzustellen.

1 Einführung und Anwendungsfall

Dieser Beitrag basiert auf dem Anwendungsfall eines Herstellers von Produktionsanlagen. Es handelt sich um Stanz-Laser-Maschinen mit einem hohen Grad an Konfigurierbarkeit sowohl in der Hardware als auch in der Software. Die Konfigurierbarkeit von anpassbaren Systemen bietet eine Reihe von Vorteilen, darunter die Möglichkeit, mit einem oder wenigen Produkten einen größeren Marktanteil zu erreichen. Daher werden zusätzliche Konfigurationsmöglichkeiten seitens der Geschäftsstrategie häufig bevorzugt. Gleichzeitig sollen Aufwand und Kosten für die Fertigstellung minimiert werden.

Die Realisierung einer Konfigurationsmöglichkeit jedoch erfordert einen zusätzlichen Aufwand auf allen Stufen des Lebenszyklus. Insbesondere fallen bei der Architektur, der Realisierung und bei der Qualitätssicherung Kosten an, die von der Entwicklung im Bezug zur Geschäftsstrategie gerechtfertigt werden müssen. Daher muss die Wirtschaftlichkeit von Konfigurationsoptionen bei der Gestaltung der Produktfamilie berücksichtigt werden. Die für diese Evaluierung erforderlichen Daten sind jedoch oft unvollständig und müssen aus unterschiedlichen Quellen extrahiert, aufbereitet und analysiert werden.

Die Integration mehrerer Fragestellungen und zugehöriger Metriken ist für die Adressierung von Geschäftszielen in ihrer Komplexität unerlässlich [1]. In diesem Bericht stellen wir ein konzeptionelles Modell zur Ausrichtung der Geschäfts- und Entwicklungsstrategien vor. Darauf aufbauend wird ein Multi-Modell-Wissensgraph entwickelt, der die Integration von Analysewerkzeugen sowie die Nachvollziehbarkeit von Ergebnissen ermöglicht.

2 Konzept: $Strategy = \frac{\partial^3 WKI}{\partial G\, \partial Q\, \partial M}$

Ein wichtiger Aspekt der Entscheidungsunterstützung in komplexen Anwendungsfällen ist die Abstimmung von Geschäfts- und Entwicklungsstrategien. Dies ermöglicht einen strategischen Umgang mit Variabilität, der organisationsübergreifende Synergien nutzt und letztlich dem aktuellen Bedarf an Konfigurationsmöglichkeiten gerecht wird [2]. Die Angleichung zwischen der Geschäftsstrategie ("Was") und der Entwicklungsstrategie ("Womit") kann durch ein gemeinsames Bindewerk, den strategischen Ansatz ("Wie"), erreicht werden. Diese drei Aspekte bilden die Gesamtstrategie. Hier stellen wir die Hypothese auf, dass die Verbindungen zwischen den drei Aspekten der Gesamtstrategie mit Hilfe des GQM-Modells [1] zusammen mit der Wissenspyramide (auch DIKW-Hierarchie) [3] erklärt werden können. Dies geschieht in zwei Schritten:

1. Zunächst klärt eine Top-down-Analyse entlang des GQM-Modells die Geschäfts- und Entwicklungsziele. Die Identifizierung konkreter Fragen bildet den Kontext für die Interpretation der Ziele. Durch die Definition von Messgrößen werden die Antworten quantifiziert.
2. Als Nächstes wird durch eine Bottom-up-Analyse entlang der DIKW-Hierarchie (s. Abbildung 1) ein Überblick über die vorhandenen Datenquellen (D) und die notwendigen Analysen für das Reengineering von Information (I), Wissen (K) und Weisheit (W) geschaffen.

Aus den Ergebnissen dieser Schritte lässt sich die Entwicklungsstrategie aus den vorhandenen In-

Abbildung 1: Hierarchische Übersicht über Wissensmanagement und Analyseaktivitäten

formationen in Bezug auf die Metriken ableiten: Entwicklungsstrategie $= \frac{\partial \text{Information}}{\partial \text{Metriken}}$. Analog dazu ist der strategische Ansatz $= \frac{\partial \text{Wissen}}{\partial \text{Fragen}}$ und die Geschäftsstrategie $= \frac{\partial \text{Weisheit}}{\partial \text{Ziele}}$.

3 Multi-Modell-Wissensgraph

Das Konzept von Abschnitt 2 ist eine Verallgemeinerung des Analyseansatzes zur Ermittlung der Variabilität in [4]. Dort wurden vier Aktivitäten entlang der Wissenspyramide [3] vorgestellt, um das Management von Konfigurationswissen zu erleichtern. Hier stellen wir einen Multi-Modell-Wissensgraphen vor, der die Analyseaktivitäten unterstützt und eine datengestützte Entscheidungshilfe bei der Identifizierung unwirtschaftlicher Variabilität ermöglicht. Für jede Ebene der Wissenspyramide (s. Abbildung 1) erstellen wir ein Modell:

Datenmodell. Die vorhandenen Artefakte werden zunächst in einem Wissensgraphen so abgebildet, dass die "physische", d. h. ursprüngliche Form der Daten wiederhergestellt werden kann. Das Ergebnis sind Sammlungen von schemalosen Dokumenten und dazwischen liegenden typisierten Kanten. Wenn die Daten beispielsweise in Excel-Dateien gespeichert sind, werden die Dateien, die Arbeitsblätter und die einzelnen Zellen als Dokumente dargestellt. Enthalten-in-Kanten stellen die syntaktische Struktur dar.

Informationsmodell. Semantische Struktur und Bedeutung werden aus der Datendarstellung abgeleitet und parallel dazu aufgebaut. Während der Ableitung werden Kanten für die Rückverfolgung zwischen dem Datenmodell und dem Informationsmodell gebildet. In unserem Anwendungsfall werden zum Beispiel Konfigurationsmatrizen von verkauften Produktinstanzen pro Arbeitsblatt gespeichert. Aus diesen wird das Informationsmodell mit Produkten, Merkmalen und deren Ausprägungen für jede Produktinstanz gefüllt. Aus anderen Arbeitsblättern werden Stücklisten und Auswahlregeln für die Teile in Bezug auf die Merkmale extrahiert. Dabei sorgen die Kanten im Modell für die Abbildung der semantischen Beziehungen.

Wissensmodell. Basierend auf dem Informationsmodell können weitere Analysen den Multi-Modell-Wissensgraphen mit einem Wissensmodell anreichern. Kontexte aus der Graphdarstellung der Informationen und Attribute werden ausgewertet und Regeln werden interpretiert. In unserem Beispiel werden Abhängigkeiten aus den Auswahlregeln abgeleitet. Der so erweiterte Wissensgraph enthält Wissen über die Zusammensetzung der einzelnen Produktinstanzen und Realisierungsbeziehungen zwischen Merkmalen und Teilen aus den Stücklisten.

Weisheitsmodell. Das gesammelte Wissen kann nun gezielt ausgewertet werden (s. Abschnitt 2). Das Ziel, unwirtschaftliche Variabilität zu identifizieren, wird erreicht, indem (a) die Kosten pro Merkmalausprägung geschätzt werden und (b) pro Merkmalausprägung der Gewinn in Relation zu seiner Bepreisung bewertet wird.

4 Fazit

In diesem Beitrag wird ein Ansatz zur niederschwelligen datengestützten Entscheidungsunterstützung bei der Identifizierung unwirtschaftlicher Variabilität vorgestellt. Der Ansatz verwendet ein neuartiges Konzept, das GQM- und DIKW-Modelle kombiniert, um die Angleichung zwischen Geschäfts- und Entwicklungsstrategien abzuleiten. Darauf aufbauend wird der Multi-Modell-Wissensgraph vorgestellt, um die Analyse und Erklärung der Ergebnisse durch einfaches Traversieren des Graphen zu ermöglichen. Der Multi-Modell-Wissensgraph ist in zweifacher Hinsicht multi-modellhaft. Zum einen bietet er eine Mischung aus schemalosen Dokumenten und Wissensgraphen. Zum anderen spiegelt die Kombination von Modellen auf unterschiedlichen Abstraktionsebenen Aspekte des Model-Based-System-Engineering wider.

Literatur

[1] F. Van Latum, R. Van Solingen, M. Oivo, B. Hoisl, D. Rombach, and G. Ruhe, "Adopting GQM based measurement in an industrial environment," *IEEE Software*, vol. 15, pp. 78–86, Jan. 1998.

[2] D. Morais Ferreira, "Development of an Architecture of an Integrated Analysis Framework for Change-Intensive Systems Based on Industry Needs," Master's thesis, Fraunhofer IESE, Fraunhofer IESE, 2021.

[3] J. Rowley, "The wisdom hierarchy: representations of the DIKW hierarchy," *Journal of Information Science*, vol. 33, pp. 163–180, Apr. 2007. Publisher: SAGE Publications Ltd.

[4] V. L. Tenev and M. Becker, "Mit Feature-Modellen das Komplexitätsmanagement vereinfachen," in *24. Workshop Software-Reengineering und -Evolution (WSRE)*, May 2022.

Towards understanding the impact of requirement evolution on deployment

Florian Schmalriede and Andreas Winter
Carl von Ossietzky Universität, Oldenburg, Germany
{schmalriede,winter}@se.uol.de

Abstract

Distributed and heterogeneous systems, such as IoT systems, enable many different alternative software deployments which lead to different system characteristics. Evolving functional or non-functional requirements might lead to adapting the deployment. This paper shows a case study where changed requirements result in a new deployment, motivating early deployment planning in software evolution.

1 IoT Systems Software Deployments

Internet of Things (IoT) systems integrate real-world things into digital networks, allowing computer systems and users to interact with them without direct physical contact. Computer systems and users analyze properties of things and influence them according to use-case-specific business logic. Processes become fully or partially automated.

IoT systems are realized as distributed and heterogeneous systems in which subsystems with different profiles cooperate across different network technologies to fulfill application-specific goals. Sensor and actuator nodes, directly attached to things or in their environment, digitize and influence properties of things via physical interactions. Servers process gained information and respond accordingly to outcomes automatically. Clients present gained information to users and react to input from users accordingly. Gateways intermediate between environment-dependent and device-specific network technologies and link individual networks of sensor and actuator nodes, servers, and clients into collaborative networks.

Different subsystems in IoT systems vary e.g. in terms of their computing power, storage capacity, power consumption and communication bandwidth due to their environment or profile. Neverless, often all subsystems are suitable to deploy software that provides requested functionality. Accordingly, there is a wide range of alternative deployments in IoT systems.

Depending on software deployment, subsystems of IoT systems have different workloads. This implies that different software deployments affect the characteristics of IoT systems in different ways. For example deployments which will lead to more workload on mobile subsystems is likely to reduce their runtime. Accordingly, deployment of software has impact on the fulfillment of requirements that specify which characteristics need to be met. Vice versa, requirements restrict the range of possible deployments. Changing requirements over time may lead to changing software deployments. If this hypothesis is confirmed, it becomes relevant to consider impacts of requirements evolution on software deployment to address them in a deployment planning phase.

To determine, if evolving requirements will impact the deployment of software, three case-study experiments are performed. The first experiment shows that initial requirements can be fulfilled with an initial deployment. In the second experiment requirements from first experiment evolve and it is shown that the initial deployment from first experiment do not met the new requirements. The third experiment show that evolved requirements can be met with a different deployment.

2 Case Study

Monitoring cold chains (cf. [1, p. 28ff.]) for food during transport is selected as an application scenario for IoT systems. Foods are placed in transport boxes, each equipped with a temperature sensor node, and loaded into cold trucks. For independence, temperature sensor nodes are supplied with mobile power sources. There are fixed gateways in cold trucks, powered by cold trucks battery, to which temperature sensor nodes share food temperature. Gateways establish an internet connection and forward measured temperature values. Additionaly, gateways can interact with drivers in order to notify them in cooling issues. Thus, gateways serve as gateways and clients at the same time. A server receives information about cold chain via internet and stores them for later access. With clients, contractors and customers access information on the cold chain on the server.

Since performing experiments with real cold trucks is very complex, a model[1] was used in [3] to perform the experiments. Each of two temperature sensor nodes are realized by an ESP32 devkit-C32D developer board and a BlueDot TMP117 temperature sensor. Rechargeable batteries with a storage capacity of 65 W are used as power source for the temperature sensor nodes. A Raspberry PI 4B with an attached 4 inch display is used as gateway. Temperature sensor nodes connected via WiFi, to a gateway providing a local hotspot. Only gateway and temperature sensor nodes are considered in the experiments, so server and clients are simulated together on a conventional laptop.

[1] The authors thank Phillip Bühring for building the model and performing experiments in his master's thesis [3].

Each experiment was performed over a period of one hour. Every second, a temperature reading was collected from both temperature sensors. Functional requirements were validated with acceptance tests, where collected temperature values were replaced with simulated temperature values to be comparable across experiments. Non-functional requirements related to the runtime of temperature sensor nodes were calculated using the capacity of mobile energy sources and the energy consumption of temperature sensors nodes (see table 1). Energy consumption was measured by INA226 power sensors which average 256 readings and have an A/D conversion time of 142 µs.

2.1 Experiment 1

Rudimentary cold chain monitoring requires functionality to detect and share cold chain interruptions. It is required that cold chain monitoring functionality is continuously available at least one working week, here Monday 6:00 am to Saturday 6:00 pm. Thus rechargeable batteries of temperature sensor nodes must last 132 hours. Between Saturdays 6:00 pm and Mondays 6:00 am, batteries can be recharged.

A cold chain monitoring software, that compare measured temperature of food continously against hard thresholds, detect cold chain interruptions on sensor nodes. If cold chain interruptions appear, the software shares information about them with other subsystems. Software on gateways inform drivers about cold chain interruptions and forwards information about interrupts to the server. The server stores information about interruptions and gives customers and contructors the possibility to consult them via clients. This system detects cold chain interruptions and prevents spoiled goods from entering the market.

The deployment in `Experiment 1` meets the functional and non-functional requirements. Acceptance tests were successfully passed and batteries run time of 141 hours exceeds the required 132 hours.

2.2 Experiment 2

The cold chain monitoring can be improved by predicting possible cold chain interruptions. If a cold chain interruption is to be expected drivers are informed such they can react. Runtime and charging time of rechargeable batteries remain unchanged to `Experiment 1`.

In order to meet emerging functional requirements, the cold chain monitoring software deployed on the temperatur sensor nodes is extended by complex calculations to predict cold chain interruptions. Software on gateways are extended to inform drivers about upcoming interruptions. Beside detecting and reporting cold chain interruptions, the upgraded system detects and informs about upcoming cold chain interruptions.

The deployment in `Experiment 2` meets the functional but not the non-functional requirements. Acceptance tests were successfully passed, but batteries runtime of 105 hours is below the required 132 hours.

2.3 Experiment 3

Reworking the deployment in experiment 3 improves these disadvantages. The cold chain monitoring software is now deployed on gateways. Thus temperatur sensor nodes do not have to perform complex calculations and save energy for measuring and sending temperature values, only.

The deployment in `Experiment 3` meets the functional and non-functional requirements. Acceptance tests were successfully passed and batteries run time of 149 hours exceeds the required 132 hours.

Experiment	Power consumption per hour
`Experiment 1`	$\approx 460\,\mathrm{mW}$
`Experiment 2`	$\approx 618\,\mathrm{mW}$
`Experiment 3`	$\approx 437\,\mathrm{mW}$

Table 1: Average power consumption per hour of temperatur sensor nodes, taken from [3].

3 Conclusion

By having an initial deployment that fulfill initial requirements and evolving the initial requirements that will not be fulfilled by initial deployment it is shown, that requirement evolution can have impact on deployment. Using an improved deployment which met the evolved requirements, shows that changed requirements call for revised deployments.

In order to avoid subsequent adjustments of deployments, including hardware adaptations, requirement evolution has to be considered already in the initial engineering process. Accordingly, deployment does not change during evolution of considered requirements.

More generally, the results show that requirements affect the deployment of software in IoT systems. Whereby different deployments may meet the same functional requirements, but may not meet same non-functional requirements. This motivates a more detailed research on the impact of requirements on deployments as well as deployment planning in order to develop requirements-aware IoT systems.

Beyond the results of the experiments, a realistic model for IoT systems was built in [3], which will be used for further research on the impact of requirements on IoT systems deployment. This allows to close gaps in research identified by [2] and will be used as a basis to find a structured way to plan the deployment of functionalities in IoT systems.

References

[1] A. Bassi et al. *Enabling Things to Talk*. Springer Berlin Heidelberg, 2013.

[2] A. Brogi et al. *How to place your apps in the fog: State of the art and open challenges*. Software: Practices and Experience, May 2020.

[3] P. Bühring. *Case studies in IoT-Deployment*. Master's thesis. Carl von Ossietzky Universität Oldenburg, February 2023.

Progress Report for a Software Reengineering Body of Knowledge (SREBOK)

Marco Konersmann
RWTH Aachen University
Aachen, Germany
https://se-rwth.de/

Jens Borchers
Borchers BfI
Hamburg, Germany
jens@borchers-bfi.de

Leif Bonorden
Universität Hamburg
Hamburg, Germany
leif.bonorden@uni-hamburg.de

Andres Koch
Object Engineering GmbH
Uitikon-Waldegg, Switzerland
akoch@objeng.ch

Sandro Schulze
TU Braunschweig
Braunschweig, Germany
sanschul@tu-bs.de

Abstract

The special interest group software reengineering (FG-SRE) of the German Informatics Society (GI e.V.) pursues the goal of discussion and moving forward the state-of-practice and research of software reengineering in the German-speaking countries since 1999. In 2021 we started an initiative to collect and share the knowledge about the state-of-the-art in software reengineering to help practitioners and researchers to get an understanding of the field. In the last WSRE workshop, the team presented the initiative's goals and plans. In this contribution and the associated talk, we report on our recent progress and expected results of this initiative.

1 Introduction

Software reengineering (SRE) is concerned with "the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form" [2]. The field of SRE is very broad, incorporating technical and organizational aspects, which stretch through all phases of the software lifecycle. This broadness makes it difficult to get an overview of the field, e.g., for practitioners to learn about SRE and apply its knowledge, methods, practices and tools to software systems and for researchers to identify valuable points for research.

To tackle this problem, on the Workshop on Software Reengineering and Evolution 2021, the special interest group software reengineering (FG-SRE[1]) of the German Informatics Society (GI e.V.) started an initiative to collect their knowledge and experience and share it with the public. The initiative's goal is to describe the state-of-the-art of SRE as an entry point for interested software engineers to learn about SRE and as a reference to existing further literature. In this contribution and the associated talk, we present the recent progress and planned activities for achieving this goal and how the audience, members of our special interest group and further experts in the field can contribute to the project.

[1] https://fg-sre.gi.de

2 Current State of the Initiative

We are currently working on a citable open access publication that collects and describes the relevant aspects of SRE in the form of a book. The "Software Reengineering Body of Knowledge" (SREBOK) is intended to describe the aspects deeply enough to serve as a self-contained overview and reference document while referencing further reading for details comparable to the Software Engineering Body of Knowledge [1]. In our contribution in 2022 [3], we presented our goals and the state of planning. Since then, we worked on the book structure, thanks to the feedback of the community. We started writing in chapters, contacted potential further co-authors, and deepen our shared understanding in regular meetings.

3 Planned Activities

In the Workshop on Software Reengineering and Evolution, we report on the current state of the document and invited the workshop participants to show their interest in contributing to the document, e.g., by co-authoring a chapter or section. Therefore, we also present an example section to showcase the SREBOK's intended style. As the next steps, we plan to:

1. iteratively sketch and refine further chapters' contents, to progress towards a complete document,
2. further get in touch with potential co-authors for chapters and sections,
3. get feedback from the community,
4. contact potential publishers in parallel,
5. publish when ready.

4 Final Remarks

We are currently actively searching for co-authors for the SREBOK. We call every interested party to join our effort to share our knowledge. If you are interested in joining, please address Marco Konersmann via konersmann@se-rwth.de.

References

[1] Pierre Bourque. *SWEBOK : Guide to the Software Engineering Body of Knowledge*. IEEE Computer Society, Los Alamitos, CA, 2014.

[2] Elliot J. Chikofsky and James H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Softw.*, 7(1):13–17, 1990.

[3] Marco Konersmann, Jens Borchers, Leif Bonorden, Andres Koch, and Sandro Schulze. Towards a Software Reengineering Body of Knowledge. In *Proceedings 24. Workshop Software-Reengineering und -Evolution (WSRE)*, volume 42 of *Softwaretechnik-Trends*, pages 47–48, 2022.

Vom Wiegen allein wird die Sau nicht fett – Was bedeutet das für unsere Analysewerkzeuge?

Elmar Juergens

CQSE GmbH
München, Germany

juergens@cqse.eu

Ich habe vor 15 Jahren als Doktorand zum ersten Mal eigene statische Qualitätsanalysen in der Praxis durchgeführt. Mich hat damals fasziniert, welche Einblicke mir die Ergebnisse in die Entwicklungsprozesse und Probleme in erfolgreichen Organisationen erlaubten.

Oft haben sie mir jedoch gezeigt, dass die Vorstellung, die ich als Forscher von der Praxis hatte, und von der sich die Anforderungen für unsere Analysewerkzeuge abgeleitet haben, bestenfalls unvollständig war. Als wir beispielsweise Clone Detection bei einem unserer ersten Industriepartner eingesetzt haben, haben wir hunderttausende Zeilen Code gefunden, die zwischen mehreren Abteilungen in dreifacher Kopie gepflegt wurden – und bei denen nachweislich kritische Fehler nur teilweise behoben waren. Unser Vorschlag, hierfür gemeinsame Komponenten einzuführen, rief jedoch zu unserer Überraschung wenig Begeisterung hervor: Die drei Kopien waren entstanden, weil die Organisation vorher mit der Pflege dieser Funktionalität in einer gemeinsamen Komponenten gescheitert war. Was dieser Partner wirklich brauchte, waren funktionierende Ansätze für Management von Duplikaten, nicht für die Erkennung. Das stellt aber ganz andere Anforderungen an die Analyse, als die reine Erkennung von Duplikaten.

Heute bekomme ich diese Einblicke öfter. Unsere Firma beschäftigt inzwischen 60 Mitarbeiter, von denen knapp die Hälfte in Software Engineering promoviert hat. Wir alle arbeiten ausschließlich an der Entwicklung oder dem Einsatz von Qualitätsanalysen. Aus unseren Forschungsprototypen ist inzwischen ein kommerzielles Analysewerkzeug geworden, das eine Vielzahl der Analyseansätze aus unserer Community implementiert. Es wird von Entwicklern und Testern weltweit täglich eingesetzt, von KMUs bis hin zu DAX-Konzernen, von Behörden bis zu Unternehmen im Silicon Valley. Auch 15 Jahre später fasziniert mich, welche Ergebnisse und Überraschungen unsere Analysen in der Praxis zu Tage fördern.

Und immer noch zwingen mich die dabei gewonnenen Einblicke regelmäßig, mein Verständnis der Anforderungen an Qualitätsanalysen weiterzuentwickeln. Konkret heißt das oft, dass wir fundamentale Annahmen, die der Architektur unserer Analysen zugrundelagen, über den Haufen werfen müssen. In anderen Worten, wir müssen unser eigenes Werkzeug Re-Engineeren.

In dieser Keynote möchte ich zentrale Erkenntnisse aus dem Einsatz von Qualitätsanalysen teilen und vorstellen, wie sie mein Verständnis verändert haben, was ein gutes Analysewerkzeug ausmacht.

Experiences with Using a Pre-Trained Programming Language Model for Reverse Engineering Sequence Diagrams

Sandra Greiner[1], Nicolas Maier[2], Timo Kehrer[1]
[1]University of Bern, Switzerland
[2]University of Fribourg, Switzerland

Abstract *Reverse engineering software models from program source code has been extensively studied for decades. Still, most model-driven reverse engineering approaches cover only single programming languages and cannot be transferred to others easily. Large pre-trained AI transformer models which were trained on several programming languages promise to translate source code from one language into another (e.g., Java to Python). Thus, we fine-tuned such a pre-trained model (CodeT5) to extract sequence diagrams from Java code and examined whether it can perform the same task for Python without additional training.*

Motivation and Background. Reverse engineering aims at transforming source code to abstractions in the form of models, such as UML-like diagrams, which shall facilitate program comprehension and analysis, serving as starting point for optimizing and modernizing legacy software.

Existing model-driven reverse engineering tools are often trimmed to a single programming language, and they typically produce a structured, yet highly generic representation of the source code [3]. For instance, the well-known MoDisco tool [1] transforms Java source code (compatible to Java 1.6) into an instance of its own Java metamodel or the OMG's KDM metamodel[1]. While the Java metamodel abstracts from some details of the Java AST, it still represents the source file contents in a very fine-grained way, lacking the key feature of a model to focus on a dedicated aspect for a well-defined purpose. Both transferring the parser infrastructure to other source languages and extracting higher-level abstractions or domain-specific models require almost prohibitive manual effort.

These downsides call for more flexible reverse engineering tools which can be easily adapted to derive abstractions residing at different levels, and for a language-agnostic approach which can generalize reverse engineering tasks from a specific programming language to support distinct languages.

In sight of large pre-trained AI transformer models, such as GPT-3 and T5, smaller versions of these general-purpose natural language transformer models have been trained to learn different programming languages. These *pre-trained programming language models* (denoted as *AI models* in the sequel) are not restricted to perform analysis tasks in one programming language but promise to handle different ones. For instance, CodeT5 [4] cannot only summarize the behavior of a program's methods in natural language but also translate it into other programming languages. However, reverse engineering capabilities of AI models have not been explored yet [2]. In our preliminary study, we examined to what extent an AI model can perform the reverse engineering task of extracting sequence diagrams from methods written in different programming languages.

Reverse Engineering Task. As reverse engineering task, we desired to extract collaborations at the method-level of object-oriented software. For instance, Figure 1a depicts an excerpt of a Java method comprising three different method invocations, two of them being wrapped by conditional statements. Figure 1b depicts the corresponding sequence diagram that captures the deduced object collaborations.

The reason for eventually choosing this reverse engineering task is twofold. First, on top of Java parsing technology, such as MoDisco, the task can be scripted in a straightforward way, allowing us to easily generate sets of training and validation data. Second, since the input to the AI model is limited to a rather small set of tokens, we expect to not exceeded these limits by the size of the examined methods. In fact, we easily reached these limits in an earlier experiment in which we tried to use an AI model to transform an entire Java code base into an instance of MoDisco's Java metamodel.

Experimental Setup and Results. In our experiments[2], we employed CodeT5 (small[3]), an AI model which outperformed the previous state-of-the-art AI model, PLBART. CodeT5 is trained on seven programming languages (including Java and Python) to perform, for instance, code-to-code translation.

Java To Sequence Diagram For training the AI model on that task, we retrieved more than 350k methods from CodeSearchNet and ran the MoDisco discoverer on them. A script converts the result-

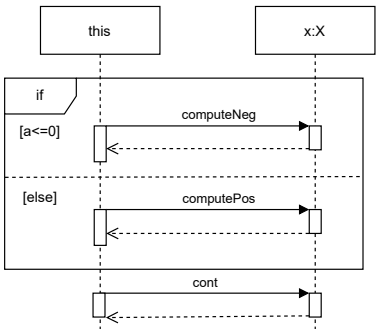
[1]https://www.omg.org/spec/KDM/1.4/About-KDM/

[2]https://doi.org/10.5281/zenodo.7628609

[3]https://huggingface.co/Salesforce/codet5-small

```java
void foo(int a) {
  // ...
  if (a <= 0) {
    x.computeNeg();
  }
  else {
    x.computePos();
  }
  x.cont();
}
```

(a) Java method

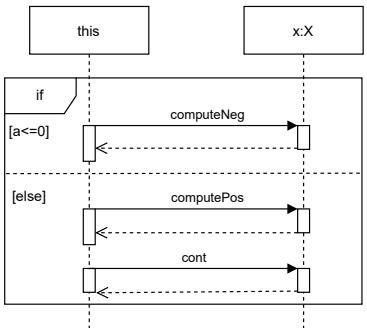


(b) Sequence diagram

Figure 1: Original Java source code and *correctly* reverse engineered sequence diagram.

```python
def foo(a):
  # ...
  if (a <= 0):
    x.computeNeg();
  else:
    x.computePos();
  x.cont();
}
```

(a) Python method



(b) Sequence diagram

Figure 2: Original Python source code and *incorrectly* reverse engineered sequence diagram.

ing XMI-files into JSON-files[4] comprising a custom description of a sequence diagram for each Java `MethodDeclaration`. For fine-tuning the AI model, we split the resulting data set into training, validation, and test sets, consisting of about 366k, 13k, and 21k examples, respectively, and fed them as tokens into the net. The fine-tuned model determined more than 98% of the sequence diagrams correctly.

Python To Sequence Diagram As our goal is to transfer the trained knowledge to other programming languages, we examined whether the fine-tuned model can extract correct sequence diagrams for Python, too. Due to the lack of a ground truth, we manually inspected a sample of selected Python methods. For example, Figure 2a depicts the method shown in Figure 1a defined in Python, and Figure 2b shows the sequence diagram obtained from our fine-tuned model. The figures demonstrate that the sequence diagram is determined incorrectly: The model associates the method invocation `cont()` with the else-branch. Fine-tuning might have caused the model to learn recognizing brackets as delimiters and contradicts our expectations on transfer learning.

Critical Discussion. We briefly summarize the major lessons learned from our experiments as follows:

Technical Limitation The limitations on the number of input tokens hinders exploring the contents of an entire class. As a consequence, the accuracy of the sequence diagram is limited as it may be unclear on which concrete objects the methods are invoked. However, given the knowledge of the input project and with additional fine-tuning of the task, we are confident that at least a post-processing mechanism can extract information about interacting objects and the corresponding classes.

Limited Transfer Learning The model transforms the Java methods into expected sequence diagrams almost perfectly (more than 98% accuracy and 99% CodeBLEU). Thus, fine-tuning the model according to the developers' recommendations worked as expected, and the aforementioned technical limitations are no serious issues.

Our initial results indicate that fine-tuning to convert Java methods may cause the AI model to forget knowledge about relationships among different pre-learned programming languages. If transfer learning between multiple programming languages is hindered by fine-tuning, it remains questionable whether the effort of training the language model pays off.

Conclusion and Outlook. All in all, we demonstrated how to reverse engineer sequence diagrams from Java methods using a pre-trained programming language model. While, after fine-tuning, the learned task was accomplished almost perfectly, our initial motivation of transferring the learning to another programming language did not meet our expectations. However, we did not experiment with different sizes for fine-tuning or with learning multi-tasks, yet. Therefore, future work may explore these two directions as well as the break-even point when the usage of an AI model exceeds conventional manual coding.

[4] Another script can transform the JSON-file into a vector graphic for visualization purposes.

References.

[1] Hugo Brunelière, Jordi Cabot, Grégoire Dupé, and Frédéric Madiot. Modisco: A model driven reverse engineering framework. *Information and Software Technology*, 56(8):1012–1032, 2014.

[2] Changan Niu, Chuanyi Li, Bin Luo, and Vincent Ng. Deep learning meets software engineering: A survey on pre-trained models of source code. In *IJCAI-22*, pages 5546–5555. International Joint Conferences on Artificial Intelligence Organization, 2022.

[3] Claudia Raibulet, Francesca Arcelli Fontana, and Marco Zanoni. Model-driven reverse engineering approaches: A systematic literature review. *IEEE Access*, 5, 2017.

[4] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708. ACL, 2021.

KI-gestützte Modernisierung von Altanwendungen: (Sentiment-) Analysen im Diskurs des Anforderungsmanagements

Andreas Schmietendorf, Sandro Hartenstein, Sidney Leroy Johnson
Hochschule für Wirtschaft und Recht Berlin
andreas.schmietendorf@hwr-berlin.de, sandro.hartenstein@hwr-berlin.de, s_johnson20@stud.hwr-berlin.de

1. Motivation

Die Entwicklung von Software erfolgt zunehmend unter Einsatz von Algorithmen der künstlichen Intelligenz. Zumeist gilt das Interesse der Integration von endnutzerzentrierten KI-Funktionen, welche mit Hilfe von Frameworks und APIs, aber auch cloudbasierten Services realisiert werden. Weniger beachtet wird die Frage, inwieweit die Aufgabenstellungen des Software Engineerings selbst von KI-Ansätzen profitieren können. Bereits vor 35 Jahren waren derartige Überlegungen Gegenstand von Konferenzen (vgl. (Gotwalt 1987)). Aktuelle Erwartungen zum KI-Einsatz im Software Engineering werden zumeist mit einem hohen Automationsgrad wie unter (Henkes 2019) in Verbindung gebracht:

> „Vor allem KI und Machine Learning mischen die Karten am Markt neu, da sie in der Lage sind, die Software-Entwicklung und -Wartung weiter zu automatisieren und zu industrialisieren."

2. Ausgangslage

Die langjährige Pflege, Wartung und Weiterentwicklung bestehender Softwaresysteme verursachen im praktischen Umfeld massive Kosten und Risiken. Gerade bei gewachsenen Softwaresystemen zeigen sich vielfach Schwächen und Inkonsistenzen in Bezug auf existierende Dokumentationen und eingesetzte (UML-) Modelle. Im Diskurs der Modernisierung von Altanwendungen bedarf es häufig der Gewinnung von Informationen aus unzureichend gepflegten Dokumentationen, Review-Dokumenten, aber auch Quellcodes. Aufgrund der fragmentierten und umfänglichen Datenmengen innerhalb eines Softwareentwicklungsprojekts (u.a. Anforderungen, Schnittstellenspezifikationen, Modelle, Quellcode, Testdaten und Testfälle, Feedbacks, …) stellt sich die Frage, inwieweit Methoden der künstlichen Intelligenz eine Analyse zur Gewinnung von „Einsichten" sinnfällig unterstützen können. Im Folgenden seien Beispiele für mögliche Forschungsfragen im Diskurs der Modernisierung von Anwendungen genannt:

- In welcher Weise präsentieren sich Artefakte des Software Reengineerings, die als potentielle Quelldaten für KI-Ansätze genutzt werden können?
- Welche Zielstellungen des Software Reengineerings lassen sich durch den Einsatz von ML-Algorithmen erreichen?
- Wie können ML-Algorithmen auf die verfolgten Ziele trainiert werden und welche Daten können dafür verwendet werden?

Im vorliegenden Beitrag soll der Frage nachgegangen werden, wie über mehrere Versionen erstellte Spezifikationen im Diskurs des Anforderungsmanagements KI-basiert analysiert werden können. Mit Hilfe eines Vergleichs der gewonnenen Erkenntnisse mit dem tatsächlich realisierten Interaktionsverhalten lassen sich so Ansätze für eine Modernisierung aus Sicht der Kunden bzw. Nutzer identifizieren. Potenzielle Modernisierungen könnten sich z.B. die Vereinfachung von Interaktionszenarien, die Beseitigung von Redundanzen oder ggf. auch semantisch fehlerhaft implementierte Fachfunktionen beziehen.

3. Existierende Arbeiten

Eine erste Analyse existierender Arbeiten zum Einsatz künstlicher Intelligenz im Software Engineering brachte die folgenden Schwerpunkte:

- (Edjlali 2014) beschäftigt sich mit Big-Data-Einsatzszenarien beim Software Engineering. Diese beziehen sich auf Daten aus sozialen Netzwerken (z.B. Stack Overflow), erzeugte Dokumente und ausgeführte Programme.
- Beim Einsatz intelligenter Entwicklungstools, unterscheidet (Satish 2019) drei Komplexitätsstufen: 1. Automation manueller Aufgaben, 2. Regelbasierte Lösungen und 3. Selbstlernende Systeme.
- Die Identifikation benötigter Testszenarien auf der Grundlage einer KI-basierten Quellcodeanalyse wird von (Endres and Hopstock 2019) untersucht.
- (Zhao and Zhao 2019) beschäftigen sich mit der Vorhersage der Anforderungsentwicklung bei der Produktentwicklung. Analysiert werden Review-Dokumente mit Hilfe einer Kombination von neuronalen Netzen (supervised deep learning) und hierarchischen Themenmodellen (unsupervised learning).

4. KI im Anforderungsmanagement

Sehr allgemein können die folgenden Beispiele für dokumentenzentrierte (in Hinsicht auf die korrespondierenden Quelldaten) KI-Szenarien im Anforderungsmanagement identifiziert werden:

- Prognosen zum Interaktionsverhalten der Softwarenutzer (Mustererkennung) zwecks Erkennung von Optimierungspotentialen auf der Grundlage selbstlernender neuronaler Netzwerke (Yang et al. 2020)
- Prognosen zur Entwicklung von nutzerspezifischen Anforderungen auf der Grundlage über einen längeren Zeitraum erstellter Dokumente, aber auch allgemeiner Meinungsäußerungen/Bewertungen in z.B. Anwenderforen (Park and Kim 2020)
- Klassifikation existierender Anforderungen mit Hilfe von Sentimentanalysen (Natural Language Processing) zur Bewertung bzw. Gewichtung von geforderten Fachfunktionen (Dias Canedo and Cordeiro Mendes 2020)

5. Prototypische Sentimentanalysen

Von den formulierten Beispielen für KI-Szenarien im Anforderungsmanagement werden in dieser Untersuchung die Anwendungsmöglichkeiten von Sentimentanalysen adressiert. Sentimentanalysen dienen der Identifizierung und Klassifizierung von Emotionen und Meinungen in einem gegebenen Text. Es verwendet Textanalysetechniken, um die allgemeine Stimmung des Textes zu bestimmen.

Erste Versuche lexikonbasierter Techniken wurden mithilfe der Software SentiStrength (Islam and Zibran 2018) durchgeführt und brachten wenig aussagekräftige Ergebnisse hervor. Es wurde festgestellt, dass verfügbare Sentimentanalyse-Tools eine sehr geringe Spezialisierung auf Software-Anforderungen besitzen und daher nicht zielführend sind.

Die Konzeption eines auf SE-Anforderungen spezialisierten neuronalen Netzes lässt aussagekräftigere Ergebnisse erwarten. Für einen ersten Prototypen wurde das bestehendes KI-Framework *PyTorch* importiert, modifiziert, trainiert, getestet und evaluiert (Ranaweera 2022). Der Konfigurationsaufbau des Prototyps ist in Abbildung 1 dargestellt.

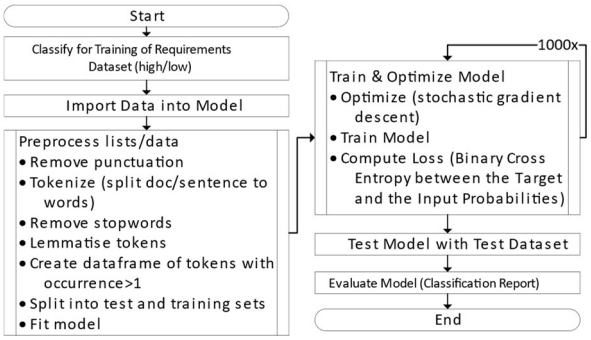


Abbildung 1 Konfigurationsaufbau des Prototyps

Der Prototyp hat mit Hilfe eines modifizierten Trainingsdatensatzes (Souvik 2019) erste Outputs eines NLP-Ansatzes produziert. Dabei wurde unter Verwendung von 48 Trainings- und 12 Testanforderungen eine Vorhersage-Genauigkeit von etwa 65 % erreicht. Mit Erhöhung der Qualität und Quantität des Trainingsdatensatzes wird in Zukunft eine deutliche Verbesserung der Vorhersagegenauigkeit erwartet. Die genutzten Trainingsdaten sind eine Sammlung von Software-Engineering-Anforderungen ergänzt durch die jeweilige Aufwandsschätzung [h für high/l für low]. Der Aufwand wurde manuell nach eigener Einschätzung zugeordnet. Die Aufwand-Zuordnung ersetzt die verbreitete Sentiment-Klassifizierung von positiv und negativ.

6. Reflektion und Ausblick

Die Auswertung der bisher erlangten Ergebnisse durch Prototypen lässt darauf schließen, dass eine Bewertung von Anforderungen bzw. geforderten Fachunktionen mithilfe von ML-Algorithmen durchaus realistisch ist, sofern geeignete Daten für den Trainingsprozess vorliegen. Eine vom Prototyp generierte Verlustkurve, welche in Abbildung 2 zu sehen ist, zeigt, dass die Qualität der Vorhersagen pro Epoche stetig verbessert wird. Der Trainingsprozess ist dementsprechend geeignet.

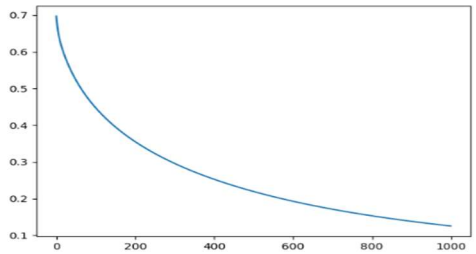


Abbildung 2 Verlustfunktion des Prototyps

Eine geplante Optimierung der Testdatenqualität besteht darin, die Trainingsdaten weiter zu unterteilen, etwa mit einem Aufwand-Score statt einer Einteilung in hoch und niedrig oder einer Aufteilung nach Anforderungsbereich. Darüber hinaus kann der bisher genutzte fiktive Testdatensatz durch aktuelle oder historische Echtdaten ersetzt werden. Im weiteren Verlauf der Forschungsarbeit wird der bestehende Prototyp durch die Bereitstellung der trainierten Modelle, im Sinne der Operationalisierung des Ansatzes, erweitert.

Quellenverzeichnis

Dias Canedo, Edna/Cordeiro Mendes, Bruno (2020). Software Requirements Classification Using Machine Learning Algorithms. Entropy (Basel, Switzerland) 22 (9). https://doi.org/10.3390/e22091057.

Edjlali, R. (2014). Shrinking Big Data. Gartner Symposium/ITxpo,

Endres, T./Hopstock, S. (2019). Machine Learning on Source Code. Available online at www.informatik-aktuell.de/betrieb/kuenstliche-intelligenz/machine-learning-on-source-code.html (accessed 4/19/2022).

Gotwalt, Susan (1987). KI-Methoden für das Software-Engineering. Computerwoche. Available online at www.computerwoche.de/a/amp/ki-methoden-fuer-das-software-engineering,1161793 (accessed 1/4/2023).

Henkes, Heiko (2019). KI mischt die Karten auch im Software-Engineering neu. Das E3-Magazin. Available online at https://e-3.de/ki-mischt-die-karten-auch-im-software-engineering-neu/ (accessed 1/4/2023).

Islam, Md Rakibul/Zibran, Minhaz F. (2018). SentiStrength-SE: Exploiting domain specificity for improved sentiment analysis in software engineering text. Journal of Systems and Software 145, 125–146. https://doi.org/10.1016/j.jss.2018.08.030.

Park, Bo Kyung/Kim, R. Young Chul (2020). Effort Estimation Approach through Extracting Use Cases via Informal Requirement Specifications. Applied Sciences 10 (9), 3044. https://doi.org/10.3390/app10093044.

Satish, Sharath (2019). Künstliche Intelligenz in der Softwareentwicklung. Available online at www.thoughtworks.com/de-de/insights/blog/augmenting-software-development-artificial-intelligence (accessed 1/4/2023).

Souvik (2019). Software Requirements Dataset. Available online at www.kaggle.com/datasets/iamsouvik/software-requirements-dataset (accessed 1/12/2023).

Yang, Bin/Wei, Long/Pu, Zihan (2020). Measuring and Improving User Experience Through Artificial Intelligence-Aided Design. Frontiers in psychology 11, 595374. https://doi.org/10.3389/fpsyg.2020.595374.

Zhao, Lingling/Zhao, Anping (2019). Sentiment Analysis Based Requirement Evolution Prediction. Future Internet 11 (2), 52. https://doi.org/10.3390/fi11020052.

Modellierung und Simulation von dynamischen container-basierten Software-Architekturen in Palladio

Nathan Hagel
Karlsruher Institut für Technologie
Karlsruhe, Deutschland
nathan@hagel.dev

Zusammenfassung

Moderne, verteilte Software-Systeme werden heutzutage nicht mehr nur statisch auf Maschinen deployed. Stattdessen werden die gewünschten Komponenten oder Container und deren Skalierungen deklarativ definiert. Eine Kontrollschleife versucht dann, den vorgegebenen Zustand des Systems dynamisch durch Starten und Stoppen von Containern und Pods zu erreichen. Auch Skalierungen von Diensten und Rollouts von neuen Produktversionen lassen sich auf diese Art realisieren.
Die Auswirkungen auf die Performance und Skalierbarkeit der Anwendung beim Einsatz dieser Techniken sind bisher nur schwer vorhersagbar.
Die in dieser Arbeit entwickelte Erweiterung für Palladio-Modelle verbessert die Modellierung und Simulation von dynamischen, container-basierten Software-Architekturen. Sie ermöglicht, container-basierte Anwendungen, sowie das Deployen mithilfe von Containerorchestrierungswerkzeugen wie Kubernetes im Palladio-Component-Model (PCM) abzubilden und zu analysieren. Für die Abbildung wurden die gängigsten Konzepte am Beispiel von Kubernetes ausgewählt und Anforderungen an dessen Abbildung definiert. Anschließend erfolgte eine Erweiterung des PCM durch diese Konzepte. Um die dynamische Allokation von Pods in Kubernetes abzubilden, wurde ein Pod-Allokations-Scheduler für PCM-Modelle entwickelt und prototypisch implementiert. Abschließend wurde ein dynamisches Simulationskonzept für Palladio erarbeitet, wobei Kontrollschleifen abgebildet wurden, die den Zustand des Clusters überwachen. Zur Evaluation wurde ein Referenz-Cluster definiert, welches mithilfe der PCM-Erweiterung und eines im Rahmen dieser Arbeit definierten Workflows abgebildet wurde. Zusammenfassend wurde festgestellt, dass sich die gängigsten Containerorchestrierungskonzepte mithilfe von Palladio abbilden und simulieren lassen. Dieses Paper basiert auf der Bachelorarbeit des Autors [1].

Motivation In den letzten Jahren wurden Containertechnologien für das Deployen von Software immer relevanter [5], [2]. Hierbei spielt neben der reinen Verwendung von Containern der Einsatz von Containerorchestrierungswerkzeugen wie Kubernetes eine große Rolle. Sie ermöglichen es, einen gewünschten Zustand des Systems deklarativ zu beschreiben. Dieser Zustand wird dann vom Containerorchestrierungswerkzeug automatisiert hergestellt und überwacht. Auch Skalierungen von Diensten und Rollouts neuer Versionen lassen sich damit vergleichsweise einfach umsetzen. Dabei stellt sich die Frage, wie sich die Verwendung dieser Technologien auf die dynamische Performance der Software auswirkt und welche Konfigurationen der Skalierungs- oder Allokationsstrategien sich für spezifische Anwendungsfälle am besten eignen. Auch für bestehende, traditionell deployte Systeme spielen diese Auswirkungen eine wichtige Rolle. Durch sich ändernde Anforderungen im Verlauf der Benutzung einer Anwendung kann es sinnvoll sein, die Art des Deployments hin zu einer container-basierten Anwendung zu evolvieren. Dabei kann ein Re-Engineering in eine container-basierte Anwendung, beispielsweise die Skalierungsmöglichkeiten erheblich verbessern.
Frühzeitige Informationen über die Auswirkungen einer Containerisierung einer bestehenden Software zu erhalten, kann bei der Entscheidung, ob ein Re-Engineering zu einer container-basierten Anwendung durchgeführt werden soll, hilfreich sein. Diese Informationen über reale Experimente, beispielsweise im Produktionssystem zu erlangen, kann schnell sehr kostspielig werden oder die Nutzer in der Verwendung der Anwendung einschränken. Mithilfe des Software-Architektursimulators Palladio [3] können bereits komponentenbasierte Anwendungen modelliert und analysiert werden. Eine Erweiterung um Container und Konzepte der Containerorchestrierung kann dabei die Entwicklung sowie den Entscheidungsprozess deutlich vereinfachen, insbesondere wenn für eine Anwendung bereits ein PCM-Modell existiert.

Zielsetzung Um dynamische, container-basierte Softwarearchitekturen im PCM modellieren und simulieren zu können, müssen neben Containern als Kapselung von Softwareartefakten auch die grundlegenden Konzepte aus der Conatinerorchestrierung abbildbar sein. Für die Auswahl der Konzepte wurde sich an Kubernetes als de-facto Standard in diesem Bereich orientiert. Zu diesen Konzepten gehören neben Containern: das Cluster, Nodes und Pods, Ressourcen Requests und Limits, Services und Ingress, Deployments und Replica Sets als deklarative Spezifikation des gewünschten Zustands, ein Scheduler für Pods, sowie eine Möglichkeit zur Skalierung des Systems. Neben der Abbildung der reinen Konzepte sollte eine Lösung gefunden werden, um nicht-statische Deployments mit Palladio zu analysieren. Hierbei spie-

len insbesondere die Pod-Allokation und die Skalierung eine wichtige Rolle. Durch eine Abbildung der genannten Konzepte in Kombination mit der Definition eines Workflows zur Analyse von dynamischen, container-basierten Anwendungen können Entwickler und DevOps-Spezialisten frühzeitig Informationen über die Auswirkung der Verwendung von Containerorchestration erhalten.

Durchführung Zuerst wurde analysiert, welche Konzepte aus der Containerorchestrierung für eine realistische Abbildung ins PCM notwendig sind. Anschließend erfolgte eine Anforderungsanalyse, um festzustellen, welche relevanten Eigenschaften der gewählten Konzepte abgebildet werden mussten. In einem ersten Schritt wurde sich auf die statischen Elemente wie Pods, Nodes und das Cluster konzentriert. Ausgehend davon wurden die bestehenden PCM-Elemente analysiert, um herauszufinden, wo diese wiederverwendet werden konnten, bzw. welche Elemente eine Erweiterung benötigten, um die gewählten Konzepte abzubilden. Nachdem Abbildungen für die grundlegenden Konstrukte gefunden wurden, wurden Analysen angestellt, um Abbildungen für Skalierung und Pod-Allokation zu definieren. Dabei wurden Möglichkeiten zur Modellrekonfiguration betrachtet und Konzepte erarbeitet, um jeweils die Pod-Allokation als auch die Autoskalierung, bspw. in Form eines Horizontal-Pod-Autoscaler (HPA) umzusetzen. Prototypisch wurde ein Pod-Allokations-Scheduler für Palladio-Modelle entwickelt und implementiert. Zur Evaluation wurde der Anteil der abgebildeten, für im Rahmen dieser Arbeit als relevant erachteten Konzepte bestimmt. Zusätzlich wurde ein Referenzcluster auf Basis einer bestehenden Anwendung definiert. Dieses wurde mithilfe des definierten Workflows in das PCM abgebildet und dabei Grenzen der Erweiterung untersucht. Abschließend wurden für ein Beispielszenario die Entscheidungen des Pod-Allokationsschedulers für Palladio-Modelle mit denen des Standard-Kubernetes-Schedulers verglichen.

Ergebnisse Auf Basis der definierten Anforderungen für die Kubernetes-Konzepte konnten für fast alle Elemente Abbildungen in das PCM gefunden oder definiert werden (z.B. Pods, Nodes, Deployments). Die einzige Ausnahme bildete das Replica Set, welches sich jedoch mit dem Deployment sehr stark überschneidet und deshalb nicht extra abgebildet wurde. Für den implementierten Pod-Allocator für Palladio-Modelle konnte anhand von Tests festgestellt werden, dass sich dieser grundsätzlich äquivalent zum Standard-Kubernetes-Scheduler verhält. Das Referenz-Cluster konnte im Rahmen der Evaluation mithilfe des definierten Workflows zur Verwendung der PCM-Erweiterung vollständig in Palladio abgebildet werden. Für die Abbildung der Kontrollschleifen wurde mit Modellrekonfigurationen basierend auf Query Views Transformation operational (QVTo) und dem Actions-Modell von Stier [4], welches zusätzlich die Möglichkeit bietet, transiente Effekte, bspw. den Overhead beim Starten eines neuen Containers zur Simulationszeit abzubilden, eine Möglichkeit gefunden, um auch dynamische Szenarien abzudecken.

Fazit & Ausblick In dieser Arbeit wurde eine Abbildung für container-basierte Anwendungen auf Basis von Containerorchestrierungswerkzeugen, wie Kubernetes entwickelt. Zusätzlich wurden Vorbereitungen, bspw. mit der Implementierung eines Pod-Allokations-Schedulers für PCM-Modelle getroffen, um dynamische Simulationen dieser Modelle durchzuführen. Darauf basierend wurde ein dynamisches Simulationskonzept entwickelt. Es wurde ein Workflow zur Verwendung der in dieser Arbeit entwickelten PCM-Erweiterung definiert. Die Abbildungen wurden anhand einer Referenzanwendung und eines Referenz-Cluster-Zustands evaluiert, sowie die Scheduling-Entscheidungen des implementierten Pod-Allokations-Schedulers mit denen des originalen Kubernetes-Schedulers verglichen. Als Fortsetzung dieses Projekts bietet sich die Umsetzung des dynamischen Simulationskonzepts an. Dies würde eine ausführlichere Evaluation der entwickelten Konzepte im Vergleich mit realen Systemen erleichtern. Damit können verschiedene Skalierungs- und Allokationsstrategien implementiert und evaluiert werden. Die genannten Schritte könnten die Performancevorhersage für Containeranwendungen erheblich erleichtern. Im Rahmen eines Folgeprojektes wird bereits an der Umsetzung dieser Überlegungen gearbeitet.

Literatur

[1] Nathan Hagel. "Modellierung und Simulation von dynamischen Container-basierten Software-Architekturen in Palladio". Bachelor's Thesis. Karlsruher Institut für Technologie, 2022.

[2] Grand view research. "Application Container Market Analysis Report By Deployment, By Platform (Kubernetes, Docker), By Organization Size (SMEs, Large Enterprise), By Service, By Application, By Region, And Segment Forecasts, 2019 - 2025". In: (1.01.2023). URL: https://www.grandviewresearch.com/industry-analysis/application-container-market#.

[3] Ralf Reussner u. a. *Modeling and Simulating Software Architectures - The Palladio Approach*. Okt. 2016, S. 1–349. ISBN: 9780262034760.

[4] Christian Stier. "Adaptation-aware architecture modeling and analysis of energy efficiency for software systems". Dissertation. 2019. URL: https://doi.org/10.5445/KSP/1000086089.

[5] Qi Zhang u. a. "A Comparative Study of Containers and Virtual Machines in Big Data Environment". In: *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. 2018, S. 178–185. DOI: 10.1109/CLOUD.2018.00030.

Code Smell Detection using Features from Version History

Ulrike Engeln
Institute for Software Systems, Hamburg University of Technology

1 Introduction

Code smells are indicators for bad quality of source code. In 1999, Fowler and Beck introduce the concept of smells to ease identification of refactoring opportunities in software. They define about 20 structures in code that commonly require re-engineering.

Manual identification of smells is usually time-consuming and costly since deep understanding of the whole software project is necessary to, for example, identify dependencies in the code. Therefore, there exist several attempts of automated code smell detection in literature. Most of them are based upon static properties of the software, e.g., code metrics. However, not all design flaws are of structural nature. There exist three different kinds of code smells. They target coding style, responsibilities, and interdependencies of classes or methods. While smells that target coding style are of structural nature only, smells that describe interdependencies, e.g., Feature Envy, cannot be detected by structural properties. For identification of such smells, Palomba et al. propose using the version history of the code as source of information. Smells targeting responsibilities affect both, code metrics and version history. For example, God Classes usually have many lines of code and appear frequently in the version history.

A well suited approach for the development of a smell detector are machine learning techniques that learn based on features, i.e., measurable properties of the software under investigation, e.g., code metrics. If we apply machine learning techniques to automate smell detection, then we expect classifiers trained on code metrics and information from version history to produce orthogonal results since they focus on different aspects of smells. To improve performance in code smell detection and enable a classifier to detect all three kinds of smells, Barbez et al. introduce a hybrid, ensemble learning-based smell detection technique, which combines classifiers using code metrics and information from version history. We propose a different approach of combining the two sources of information, which, rather than combining existing detectors, directly learns identification of smells from the two sources of information.

Since in their work Palomba et al. do not draw features from version history but apply heuristic rules, one major issue of our machine learning approach is to decide how to express information from the version history by features.

2 Features from Version History

The introduced method of feature extraction from version history builds the core of our work. The general idea is to measure how often files or methods change simultaneously.

We introduce three design parameters, which determine which parts of version history are considered as simultaneous change.

- **Direction** specifies whether prior or future changes are evaluated.
- **Window size** specifies the number of history entries that are considered.
- **Weighting** specifies how changes are weighted as function of their distance.

For the weighting, we define a constant, a linear, and an exponential strategy as follows:

$$w_{\text{const}} = \begin{cases} 1, \text{ if } distance \leq window\_size. \\ 0, \text{ otherwise.} \end{cases}$$

$$w_{\text{lin}} = \begin{cases} 1 - \frac{distance}{window\_size}, \text{ if } distance \leq window\_size. \\ 0, \text{ otherwise.} \end{cases}$$

$$w_{\text{exp}} = \begin{cases} 2^{-distance}, \text{ if } distance \leq window\_size. \\ 0, \text{ otherwise.} \end{cases}$$

Figure 1 illustrates the general idea of measuring simultaneous changes. For each file, we create a vector containing entries for all available files. We identify all commits containing the file and distribute points for all files that appear within the defined window (here: forward oriented of size 3 starting from commit 2) according to the chosen weighting strategy.

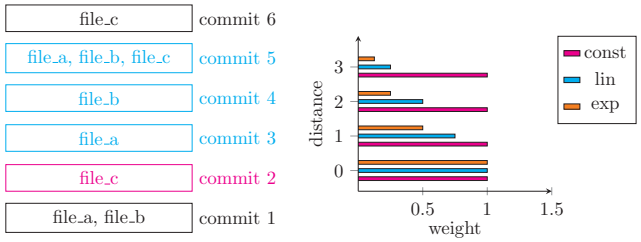


Figure 1: Concept for distributing weights from git history for *file_c*. Relevant commits are colored. Right: different weighting concepts.

We use statistical description of the normalized weight vectors, among others maximum, median and mean, as historical features.

3 Evaluation of Historical Features

For evaluation of the introduced historical features we investigate the following research question.

> **RQ1**: Do the historical features introduced in this work lead to better code smell detection compared to using only code metrics?

We answer the formulated question through three hypotheses, which we test on the code smells God Class and Feature Envy. We select those two smells since they target responsibility and interdependencies, respectively. They belong to the two kinds of smells that we expect to show in version history. Further, for the two smells, labeled data is available in existing work.

With the first hypothesis,

> **RQ1.H1**: There exists information about smells in version history.

we seek to verify whether the designed historical features can capture information about smells from version history at all.

The second hypothesis investigates the impact of the design parameters:

> **RQ1.H2**: For Feature Envy, most information is gained from history if metrics focus on the close past.

We assume that, especially for smells that target interdependencies, the choice of design parameters (see Sec. 2) matters.

The last hypothesis addresses the core of our work. We explore whether historical metrics complement code metrics assuming that:

> **RQ1.H3**: In particular for detection of smells that target interdependencies, information from version history adds value compared to using only code metrics.

For verification of the hypotheses, we perform experiments following the workflow from Figure 2. As ground truth, we apply the available data from Madeyski et al. and Palomba et al. [2, 3]. We have 111 instances of God Class and 125 instances of Feature Envy available. For each smell, we add good instances such that we obtain a data set of $\frac{1}{11}$ smelly instances and $\frac{10}{11}$ good instances. We extract code metrics and historical features of all instances from the data sets of the two smells. Next, we split the available data into test and training part where we ensure that data from a project either appears in the test or the training data to avoid correlation in the data sets. The training data is used to train a random forest of 100 decision trees, which we evaluate by different performance measures. For more reliable evaluation results, we repeat the training process applying 10-fold cross-validation.

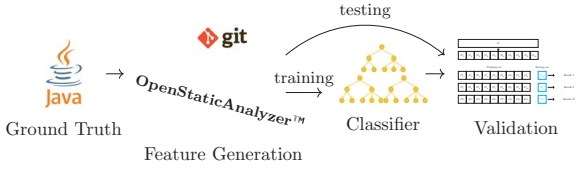


Figure 2: General workflow of the experiments.

4 Results

Evaluation of RQ1.H1 shows that historical features contain information about code smells. For RQ1.H2, surprisingly, we are not able to identify specific design parameters that show best performance. The results of RQ1.H3, finally, are given in Table 1. For God Class, we observe improvement of 0.3% to 1.8% in all performance measures but precision, which decreases by about 0.4%. For Feature Envy, we observe a significant improvement, e.g., F1-score increases from 43.03% to 54.36%.

Table 1: Performances with and without historical features.

	Feature Envy		God Class	
	code metrics	with history	code metrics	with history
Accuracy	90.79%	92.18%	95.95%	96.22%
F1-Score	43.03%	54.36%	74.87%	75.72%
Precision	57.20%	70.20%	77.22%	76.83%
Recall	42.37%	49.94%	73.96%	75.74%
AUC	0.6885	0.7323	0.8651	0.8737

5 Conclusion and Future Work

In our work, we introduce a method to draw historical features that improve smell detection. Results from Section 4 show that historical features allow for better detection of smells targeting interdependencies, e.g., Feature Envy. For God Class, the observed differences are too small to assume an impact.

Future work should focus on four points:

- impact of the design parameters of the features,
- value of historical features for smells targeting responsibilities,
- extension to other smells,
- extension to other programming languages.

A major challenge of any empirical evaluation, and thus for all open points, is the availability of ground truth since there barely exist labeled data of code smells. In our work, we observe that results highly depend on the selected training and test set. Thus, a wider data set could lead to more insights for the first two points. Extension to other smells requires further data sets. For the fourth point, an alternative to training on new data is the application of transfer learning, which we investigate in the master thesis that belongs to this paper [1].

References

[1] Ulrike Engeln. *Transfer learning code smells using version history.* Master's thesis, Hamburg University of Technology, Forthcoming 2023.

[2] L. Madeyski and T. Lewowski. Mlcq: Industry-relevant code smell data set. EASE '20, page 342–347. ACM, 2020.

[3] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, D. Poshyvanyk, and A. de Lucia. Mining version histories for detecting code smells. *IEEE Trans. Softw. Eng*, 41(5):462–489, 2015.

Monatliche Kurzberichte zur Softwarequalität, deren derzeitige Wahrnehmung im Entwicklungsprozess und Verbesserungspotenzial*

Dominik Klein♠, Jakob Rott♣

♠dominik.klein@tum.de – TU München, School of Computation, Information and Technology, Garching b. München
♣rott@cqse.eu – CQSE GmbH, München

Zusammenfassung

Die Wahrnehmung von Berichten über die interne Softwarequalität, die im Rahmen der Qualitätssicherung von Systemen teils kostspielig erstellt werden, ist in der Praxis nicht hinreichend untersucht. Während sich die Beachtung durch Manager beobachten lässt, ist gerade die von Dev-Teams empfundene Nützlichkeit und, ob Handlungen abgeleitet werden, unbekannt. Werden festgestellte Qualitätsdefizite nicht behoben, blieben z. B. Wartungsprobleme im System bestehen und eingesetzte Aufwände für die Berichterstellung wären in diesem Aspekt folgenlos. In diesem Artikel[ℵ] wird vorgestellt, wie viele von 1 800, in Berichten genannten, Findings in industriellen Systemen später behoben wurden. Außerdem wird eine softwareassisstenten-gestützte Methode vorgestellt, die es erlaubt, Qualitätsberichte zu verteilen, und von den Lesern Feedback einzuholen. Die prototypische Implementierung wurde in einem Konzern mit gewachsenem Applikationsportfolio evaluiert.

Es zeigte sich, dass etwa ein Viertel der berichteten Findings später behoben wurden. Die Möglichkeiten einer assisstentengestützten Führung durch Qualitätsberichte wurde von den Studienteilnehmern begrüßt.

1 Einleitung

Der bloße Einsatz von Analysewerkzeugen im Qualitätssicherungsprozess bewirkt nicht automatisch, dass Findings (mgl. Qualitätsdefizite) behoben werden [1]. Es bedarf z. B. der entsprechenden Awareness und Zeit, um die Qualität zu erhalten, bzw. eine Verbesserung zu erzielen. Berichte können als Sicherheitsnetz dienen und auf übrige Auffälligkeiten hinweisen. Im Artikel wird zunächst das Berichtsformat sog. *Monthly Assessments* (MAs) eingeführt, anschließend anhand von etwa 2 600 derselben ihr Aufbau analysiert und untersucht, wie viele der, in den Berichten genannten, Findings später behoben wurden. Im zweiten Teil wird eine neue softwareassisstenten-gestützte Methode vorgestellt, um MAs mit besserem Nutzererlebnis zu verteilen und einen geeigneten Rückkanal für Feedback an die MA-Autoren zu schaffen.

*Das dem Artikel zugrundeliegende Vorhaben wurde mit Mitteln des Bundesministeriums für Bildung und Forschung unter dem Förderkennzeichen **Q-Soft, 01IS22001A** gefördert. Die Verantwortung für den Inhalt der Veröffentlichung liegt bei den Autoren.
ℵAlle vorgestellten Ergebnisse und Techniken wurden im Rahmen der Bachelorarbeit von Dominik Klein an der Technischen Universität München erhoben und entwickelt.

2 Das Berichtsformat »Monthly Assessment«

Monthly Assessments (MAs) sind ein Format für regelmäßige Entwicklungsberichte zur Softwarequalität. Nach mehrjähriger Anwendung wurden sie von Jürgens und Proft im Jahr 2020 auf der OOP-Konferenz vorgestellt [2]. Mit ihnen soll auf Abweichungen vom Qualitätsziel† aufmerksam gemacht werden. MAs betrachten jeweils die Softwareentwicklung aus dem Vormonat und verarbeiten dabei Ergebnisse von toolgestützten Qualitätsanalysen. Verfasst werden diese Kurzberichte meist von entwicklungsteamexternen Personen, den sog. *Quality Engineers* (QEng.). In MAs werden gerade solche Findings genannt, die bisher vom Dev-Team weder behoben noch als toleriert gekennzeichnet wurden und vom QEng. als behebens- oder prüfenswert eingestuft werden. MAs sind im Wesentlichen dreiteilig und sprechen durch Inhalt auf unterschiedlichen Abstraktionsebenen unterschiedliche Zielgruppen (Managementstufen und Rollen im Dev-Team) an.

- *Assessmentfarbe:* Der Qualitätstrend, vornehmlich anhand des vergangenen Monats bemessen, wird durch eine Ampelfarbe dargestellt. Eine *grüne* Bewertung bedeutet die Einhaltung des Qualitätsziels, eine *gelbe* bzw. *rote*, dass kleinere bzw. größere Abweichungen vom Qualitätsziel beobachtet wurden.
- *Kopfzeile:* Fasst das Assessment zusammen, weist ggf. auf Abweichungen vom Qualitätsziel hin.
- *Details:* Einzelne Findings oder häufige Findingstypen werden hervorgehoben und ggf. durch einen Vorschlag zur Behebung des Findings ergänzt.

MAs werden an Projektstakeholder per E-Mail verschickt und im Intranet des jeweiligen Auftraggebers als Webseite veröffentlicht, die eine Übersicht (inkl. MAs) über das gesamte Applikationsportfolio bietet.

	Compare	Delta	Finding	Metrics	Task
Kopfzeile	-	-	2	2	-
Details	376	666	2115	225	2

Tabelle 1: Genutzte Verweise innerhalb der untersuchten MAs. Meist wurde aus dem *Details*-Bereich verlinkt. *Compare* zeigt eine Vergleichsansicht von Quelltextduplikaten (Klonen); *Delta* bietet einen (meist Findings-)Vergleich zw. zwei Codeständen; *Finding* zeigt ein einzelnes Finding, sowie Metadaten und eine Erklärung; *Metrics* zeigt aggregierte Qualitätsmetriken auf Basis der Datei- und Ordnerstruktur; *Task* verweist auf eine, im Tool festgehaltene, Aufgabe zur Behebung von Findings.

†Ein Qualitätsziel legt z. B. den Fokus auf die Behebung von lediglich neu hinzugekommenen Findings bzw. auf die proaktive Behebung bestehender Findings (vgl. auch [2], S. 12).

Finding behoben innerhalb 45 Tage	182 (10,1%)
F. behoben später als 45 Tage danach	234 (13%)
Beständiges Finding	957 (53,2%)
Info technisch nicht nachvollziehbar	427 (23,7%)

Tabelle 2: Werden Findings, die in Berichten von Quality Engineers explizit referenziert werden, behoben? Hier gezeigt ist die rohe Behebungsverteilung über die letzten Jahre. Hinweis zur Interpretation: Nicht-Behebungen können begründet sein und wurden möglicherweise mit den QEngs. diskutiert.

3 Ist-Aufnahme: Wahrnehmung von MAs

In einer vorbereitenden Analyse von 2 604 MAs (exkl. MAs über entwicklungsfreie Zeit; inkl. »grüne« MAs meist ohne verlinkte Findings) zeigte sich, dass häufig auf Findings verwiesen wird, die ein Qualitätsziel verletzten (Tab. 1). Dies geschieht, um den Bericht zu kräftigen und Verbesserungsvorschläge anzubringen. Diese Vorstudie diente als Grundlage, um Verbesserungspotenzial im Prozess um die MAs aufzudecken.
So gibt es nach dem Versand und der Veröffentlichung der MAs derzeit keinen geregelten Rückkanal, um festzustellen, ob und von wie vielen sie gelesen werden und wie hilfreich die Vorschläge für die Empfänger sind.
Um die Wirkung annährungsweise festzustellen, wurde untersucht, ob die Findings, auf die in den MAs verwiesen wurde, im Anschluss (45 Tage nach Berichtverteilung) von den Dev-Teams behoben wurden. Dies spräche dafür, dass der Bericht sowohl gelesen wurde, als auch hinsichtlich seiner Relevanz Zustimmung gefunden hat. Zudem wurden die Empfänger der MAs gebeten, die folgenden Fragen zu beantworten.

a) Lesen Sie üblicherweise die MA-E-Mails?
b) Wussten Sie, dass MAs von Quality Engineers verfasst und nicht automatisiert erstellt werden?
c) Haben Sie bewusst Findings aus MAs behoben?

Findingsbehebungen Die etwa 2 600 MAs enthielten 1 800 Verweise auf konkrete Findings. Tabelle 2 zeigt, inwieweit diese nach der Verteilung der MAs behoben wurden. Etwa ein Viertel der Findings wurde behoben: 10% innerhalb und weitere 13% später als 45 Tage nach der Berichtverteilung. Grob die Hälfte der Findings verblieb in den Systemen. Wiederum gut ein Viertel der untersuchten Findings konnte aus technischen Gründen nicht mehr nachvollzogen werden.

Stakeholderbefragung Die Rücklaufquote des verbreiteten Fragebogens war gering. Von den 257 kontaktierten Projektbeteiligten antworteten nur 19. Die Rückmeldungen sind in Abbildung 1 dargestellt.

Schlussfolgerung Der Anteil der durch die Dev-Teams behobenen Findings spricht dafür, dass in den MAs relevante Findings genannt werden. Die Analyse zeigt Tendenzen, lässt allerdings keine klaren Ableitungen zu. Die Befragung deutet auf ein grundlegendes Interesse an den MAs hin, so werden entsprechende E-Mails gelesen und Findings bewusst behoben.

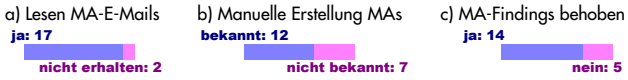


Abbildung 1: Ergebnisse der Stakeholderbefragung. Werden Verbreitungsmails gelesen? – Ist klar, dass MAs nicht automatisiert erstellt werden? – Wurden vom Befragten schon Findings aus MAs behoben?

Gleichzeitig wird Bedarf deutlich, besser aufzuklären, dass MAs von Experten erstellt werden, um so deren Wertigkeit klarer zu machen. Ein entwicklungsbegleitendes Tooling, das, sowohl fachliches als auch prozedurales, Feedback zu MAs zulässt, böte Gelegenheit der Frage nach der Wirksamkeit von MAs gezielter zu begegnen und Optimierungspotential zu finden.

4 Softwareassistent⇒Wirksamkeitserhöhung?

Auf Basis von *Teamscale*, einem Analysetool für Softwarequalität, wurde ein prototypischer Softwareassistent entwickelt, der es Nutzern ermöglicht, MAs interaktiv durchzuarbeiten. Absatzweise wird den Lesern ein MA präsentiert und jeder Teil kann separat bewertet werden: Einerseits durch einen Smiley (☺, 😐 und ☹), andererseits durch Freitexteingabe.

Verprobung Der MA-Assistent wurde für zweieinhalb Monate zur Evaluation beim o. g. Unternehmen eingesetzt. Stakeholder, insb. Entwickler, wurden aufgefordert, den Prototypen zu nutzen und ihre Erfahrung in einer Onlinebefragung rückzumelden. In einer fünfstufigen Likert-Skala (5: *stimme voll zu*, 1: *stimme überhaupt nicht zu*) wurde abgefragt, ob MAs aufmerksam durchgearbeitet würden – mit oder ohne Assistentenführung. Die Antworten finden sich in Abb. 2. Bei einem Viertel der Teilnehmern trüge der Assistent dazu bei, MAs mehr zu beachten. Obgleich keineswegs eine Allgemeingültigkeit abgeleitet werden kann, ist das Ergebnis vielversprechend und lädt ein, die Verteilung von MAs zu modernisieren. So könnten kleinteilige Rückmeldungen aus Dev-Teams stetig dazu beitragen, dass Folgeassessments an Relevanz gewinnen.

5 Diskussion und Ausblick

Diese Arbeit stellte heraus, dass einige der Findings, die in MAs von QEngs. genannt wurden, später aus der Codebasis entfernt werden. Zudem zeigt sich, dass die Auswertungsmöglichkeiten im bisherigen Prozess um die MAs beschränkt sind und diese durch einen Softwareassistenten deutlich erweitert werden könnten. Dieser würde zudem einen Feedbackkanal von Dev-Teams zu QEngs. aufbauen.
Eine zukünftige Arbeit kann der längere Einsatz des Prototypen sein, um die Nutzbarkeit weiter zu evaluieren und einen dauerhaften Einsatz anzustreben.

Literatur

[1] Elmar Jürgens. Vom Wiegen allein wird die Sau nicht fett-Erfahrungen aus einem Jahrzehnt Qualitätsanalyse in Forschung und Praxis. *Software Engineering*, 2020.
[2] Elmar Jürgens und Uwe Proft. Kosten-Nutzen-Berechnung von Qualitätsanalysen – Erfahrungen bei der Munich Re. `cqse.eu/sl/nutzenberechnung-qa-munichre`, 2020. OOP Konferenz.

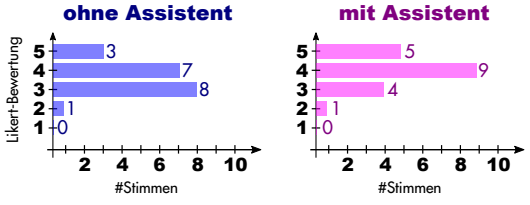


Abbildung 2: In der Umfrage, ob Beteiligte ein MA mit oder ohne Assisstentenunterstützung aufmerksam durcharbeiten würden, zeigen sich unter Nutzung des Assisstenten höhere Werte.

Using a Hardware Simulation for Automatic Software Performance Model Parameterization

Sebastian Weber
sebastian.weber@fzi.de
FZI Forschungszentrum Informatik

Abstract

While the fulfilment of functional requirements during software re-engineering or maintenance can likely be monitored with existing test cases, checking whether quality requirements (e.g. performance) are still satisfied requires additional effort. A developer would have to either measure the target system or analyse it based on models. The precise parameterization of these models is usually based on measurements which require an executable software and the target hardware. This paper proposes an approach to use hardware simulations for the automatic parameterization of performance models to remove the need for the target hardware. The results show that the accuracy of the hardware simulation for the chosen hardware requires improvements, if these results are intended to replace measurements. Nevertheless they show the applicability of the approach.

1 Introduction

Modeling and analysis of a system composed of software and hardware can be achieved with different approaches. The approach used to implement the hardware-simulation-based parameterization is the Palladio Approach [3] which aims at component-based software, because it offers comprehensive and extendable modeling and analysis tooling. The Palladio Component Model (PCM) consists of five different model types.

The repository model contains software components that can require and provide interfaces. If a component provides an interface it has to provide implementations for the methods of this interface. These implementations are called Service Effect Specification (SEFF) and describe which resources (e.g. HDD or CPU) are required for the execution of this implementation. The amount of work required on a resource is described on a high level of abstraction as work units which can for example be mapped to number of cycles for a CPU.

In the assembly model the components can be combined to a system according to the interfaces they require and provide. Also the interfaces provided by the system are specified. The allocation model describes how the elements of the assembly model are deployed on hardware which is modelled in the resource environment model. The last model is the usage model which defines how the system is used. The simulation of these models can be used to analyse different quality properties of the system. In this paper only the performance of the system is considered. The simulator used is SimuLizar [2].

As hardware simulation gem5 [1] was chosen, because it supports a multitude of different hardware targets and simulation granularities. Besides its high configurability it is actively developed and has a permissive license. A hardware specification and an executable are required to run a simulation. The hardware specification primarily consists of specifications of the CPU, cache, RAM and memory. Depending on the chosen granularity, different parameters can be adjusted.

The contribution of this paper is the combination of the Palladio approach with the hardware simulation gem5, which was done during a master's thesis [4], to automatically parameterize Palladio models based on given hardware simulation inputs. This allows developers to quickly and easily adapt the model of a system to changed hardware or software by simply adapting the required hardware simulation inputs in the models. Deployment and measurement of the system are not required as long as the chosen hardware simulation is accurate enough. Section 2 describes how the integration was implemented, Section 3 presents evaluation results and Section 4 concludes this paper with a summary and outlook.

2 Implementation

To be able to utilize gem5 for the parameterization of a PCM, the required input data for gem5 has to be stored in related models. The executables are stored in the SEFFs of the repository model instead of a specification of a demand in work units. The hardware specifications are attached to the corresponding resource containers. Possible parameters of the executables are stored in the usage model.

Palladio simulates the execution of the system under the given usage model. Whenever a method of the system is called, it forwards this call with possible parameters to the corresponding component. If

this component needs resources according to the specification of the SEFF, this resource demand in work units will be scheduled on the corresponding resource indicated by the allocation of the component. If the resource demand is given as an executable it will be processed by a SimuLizar extension first.

The extension resolves the hardware specification based on the resource the demand should be scheduled on. The executable is already given in the SEFF and possible parameters are retrievable from the call. Subsequently the executable, the parameters and the hardware specification are forwarded to gem5.

During the execution of the hardware simulation, the simulation of Palladio is stopped. When the hardware simulation is finished, the results of it are converted to a number of work units which can be returned to Palladio. At this point the simulation of Palladio continues with the computed number of work units. By caching the results the number of time-consuming hardware simulation executions is reduced to one per hardware simulation input. This is viable due to gem5 being deterministic, so it always yields the same result.

3 Evaluation

The evaluation of the approach focuses on the achieved accuracy of gem5 and on the increased simulation time of Palladio when using the hardware-simulation-based parameterization. To evaluate the accuracy, the execution of three example applications is simulated with different parameters. These applications were the calculation of a MD5 hash from a file (MD5), the conversion of an audio file from "Waveform Audio File Format (wav)" to "Apple Lossless Audio Codec (alac)" (Alacconvert) and the computation of Fibonacci numbers (Fibonacci).

Table 1 shows the number of measured and simulated cycles together with their ratio with the given parameter or parameter properties in the first column. The ratio of the measured and simulated number of cycles converges to approximately 0.75 with increasing input size. This indicates that the hardware specification either was not parameterized precisely enough or does not contain the necessary parameters, e.g. the latency of cpu instructions. Due to the simulation times being between 10000 to 30000 times higher than the execution time of the software on the target hardware, no larger parameters were considered.

The simulation time of Palladio was about 150 times higher when evaluating resource demands with the hardware simulation for MD5 and Alacconvert, but these results are not transferable to larger applications or inputs. Nevertheless they show the high simulation times of hardware simulations on low levels of abstraction and their significant effect on software simulations on high levels of abstraction. Despite the induced overhead this approach should be cheaper and faster than building and measuring a test system for the model parameterization. If a test system is available it could be used in the parameterization process instead of a hardware simulation to reduce the simulation time. Using the hardware-simulation-based parameterization only for parts of the model and a manual parameterization for the rest can also reduce the induced overhead.

Parameter	Measurement	Simulation	Ratio
MD5			
1MB	8.569.324	8.383.694	1,02
2MB	15.291.831	16.558.028	0,92
3MB	21.704.249	24.730.282	0,88
Alacconvert			
1 sec	22.145.658	27.741.100	0,8
2 sec	38.884.680	53.229.985	0,73
Fibonacci			
10	952.276	184.843	5,15
20	1.253.866	562.234	2,23
30	35.216.364	46.083.982	0,76
35	380.132.623	509.121.790	0,75

Table 1: Measured and simulated number of cycles for program execution

4 Conclusion

This paper discussed the usage of a hardware simulation to automatically parameterize a software performance model. The evaluation shows the applicability of the approach despite of the improvable accuracy. Future work is to test more precise parameterizations of the hardware specification to improve the accuracy. Furthermore other hardware simulation on the same or higher levels of abstractions should be used. This might reduce the simulation time without degrading the accuracy too much.

References

[1] N. Binkert et al. "The gem5 simulator". In: *ACM SIGARCH computer architecture news* 39.2 (2011), pp. 1–7.

[2] M. Becker, S. Becker, and J. Meyer. "SimuLizar: Design-Time Modeling and Performance Analysis of Self-Adaptive Systems". In: *Software Engineering 2013*. Ed. by S. Kowalewski and B. Rumpe. Bonn: Gesellschaft für Informatik e.V., 2013, pp. 71–84.

[3] R. H. Reussner et al. *Modeling and simulating software architectures: The Palladio approach*. MIT Press, 2016.

[4] S. Weber. "Co-Simulation of Hardware and Software in the Palladio Component Model". MA thesis. Karlsruhe Institute of Technology (KIT), 2022.

Commit-Based Continuous Integration of Performance Models

Martin Armbruster (martin.armbruster@kit.edu)
KASTEL - Institute of Information Security and Dependability, Karlsruhe Institute of Technology

1 Introduction

Architecture-level performance models (aPM) such as the *Palladio Component Model* (PCM) [5] can be used for, e.g., performance predictions to explore design alternatives and combines the aspects of architecture and performance models. An up-to-date architecture model can support the software maintenance by reducing the architectural degradation or guide the software evolution. At the same time, performance models allow the investigation of the software performance without the need to implement or change the system. However, keeping them up-to-date requires manual effort which hinders their adoption. Especially in the agile software development which is characterized by incremental and iterative development cycles, no or short design phases prevent manual modeling activities.

2 Foundations

PCM The PCM is a metamodel and framework for describing and analyzing component-based software architectures [5]. In *Repository* models, components and their interfaces with offered and required services are defined. Furthermore, a component includes *Service Effect Specifications* (SEFF) which abstractly model the behavior of services in terms of actions including, e.g., calls to other services. Performance model parameters (PMP) such as a resource demand can be attached to actions which are then employed during PCM simulations to predict the performance.

There are approaches such as the *Reconstructive Integration Strategy* [3] which extract a PCM from source code. However, they do not support incremental updates and would extract a complete PCM for every source code change. In addition, the *Coevolution approach* [3] is able to incrementally update a PCM based on code changes. Nevertheless, it assumes the availability of editors that record the changes during the development.

Continuous Integration of Performance Models (CIPM) Addressing the aforementioned issues of keeping aPMs up-to-date with automatized activities, the CIPM approach proposes a Continuous Integration (CI) pipeline [4]. As first step in the pipeline, a commit-based integration strategy extracts changes from a commit and incrementally updates an aPM. At the same time, relations between code elements and their corresponding elements in the aPM are established. To estimate the PMPs, the source code is adaptively instrumented, i.e., only the parts of the source code which have changed are instrumented, and monitored. The taken measurements are used to calibrate the aPM. As a consequence of the adaptive instrumentation and monitoring, the monitoring and calibration overhead can be reduced. Moreover, unaffected PMPs are not estimated again because they have not change.

The CIPM approach's realization supports Java as source code and targets the PCM as aPM. Parts of the pipeline were prototypically implemented and evaluated in previous work. In particular, a delta-based extraction of changes from a commit was implemented and evaluated with an artificial Git history. In a second and third work, the adaptive instrumentation was implemented twice which inserts the text-based instrumentation statements into a copy of the source code. Both implementations derive the insertion locations from an instrumentation model which stores the SEFF actions to instrument as instrumentation points. Moreover, a calibration pipeline was implemented in the third work and evaluated with three case studies.

3 Approach

This master thesis [1] presents an approach building upon the previous work with these two main goals: (1) closing the gaps by completing the pipeline for the aPM extraction and instrumentation, and (2) evaluating the pipeline with a real Git history. As a result, in the approach, the Java source code in the state of a new commit is parsed into a code model. By a state-based comparison with the code model of the previous commit, a delta-based change sequence is obtained which describes how the code model of the previous commit can be transformed into a code model conforming to the state of the new commit.

Afterwards, the changes are utilized to incrementally update the PCM, i.e., only the PCM elements affected by the source code changes are updated while the other PCM elements remain unchanged. The PCM update is defined by technology- and project-specific rules which specify how a change in the Java code model is reflected as changes in the PCM. Thus, the rules allow to obtain an architecture model with respect to the applied technologies and to project-

specific conventions in the code.

In this thesis, the implemented rules are targeted at extracting the architecture of Microservice-based applications where a Microservice is modeled as a component. Thus, in a first step, Microservices are found by the organization of the source code in modules of the build system which are recognized by build files. In addition, the rules cover *Jakarta Servlets* and *Jakarta RESTful Web Services* (JAX-RS) to identify the Microservices' REST APIs in a second step which are represented in PCM interfaces. As an example, if a module of the build system is removed, the corresponding component in the PCM is also deleted. On the other hand, if a class with a corresponding PCM interface is renamed, the PCM interface is renamed, too.

In case of changes in a method with a corresponding SEFF, the SEFF is updated by adding or removing actions and revising the relations to the statements in the code. Those changes in the actions are reflected one-to-one in the instrumentation model by adding or removing instrumentation points.

As last step in the thesis' approach, the source code is adaptively instrumented. During the instrumentation, a copy of the code model is extended with the instrumentation statements whose insertion locations are based on the instrumentation model and the relations between the SEFF actions and the corresponding source code statements. The instrumented code model is printed as source code which can be compiled and executed to take the required measurements for the calibration.

4 Evaluation

The evaluation of the presented approach aims at finding out how accurate the generated models are, how accurate the code is instrumented, and how large the reduction of the monitoring overhead is compared to a full instrumentation of the source code. Therefore, the evaluation is executed with the *TeaStore*[1] which is a web-based store for tea and related products [2]. Its commits between version 1.1 and 1.3.1 are divided into four intervals and propagated within the intervals to simulate the development. All intervals span 292 commits of which 58 commits include changes in 144 Java files with overall 10030 added and 8270 removed lines.

To assess the accuracy of the generated models, the updated Java code model is compared to a newly parsed model corresponding to the specific commit. The automatically updated PCM is compared to a manually updated one. In order to obtain a manual PCM, the changes of a commit are manually analyzed and applied on the previous PCM according to the rules defined in the approach.

[1] `https://github.com/DescartesResearch/TeaStore`

For the instrumentation, several indicators are considered. They include a counting of the statements before and after the instrumentation, a compilation of the instrumented code, and a partly manual inspection of the instrumentation statements. At last, the reduced monitoring overhead is calculated as the ratio of the actual instrumented instrumentation points to all potential instrumentation points.

The evaluation results indicate that the generated models are accurately updated and the source code is accurately instrumented. The reduction of the monitoring overhead ranges between 60.5% and 97.6%. As a consequence, the thesis' approach leads to an improved usability of the CIPM approach and a reduced effort through automatization.

References

[1] Martin Armbruster. "Commit-Based Continuous Integration of Performance Models". Master Thesis. Karlsruher Institut für Technologie, Sept. 14, 2021. DOI: 10.5445/IR/1000154588.

[2] Jóakim von Kistowski et al. "TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research". In: *Proceedings of the 26th IEEE International Symposium on the Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*. MASCOTS '18. Milwaukee, WI, USA, Sept. 2018.

[3] Michael Langhammer. "Automated Coevolution of Source Code and Software Architecture Models". PhD thesis. Karlsruhe, Germany: Karlsruhe Institute of Technology (KIT), 2017. 259 pp. DOI: 10.5445/IR/1000069366. URL: http://nbn-resolving.org/urn:nbn:de:swb:90-693666.

[4] Manar Mazkatli et al. "Incremental Calibration of Architectural Performance Models with Parametric Dependencies". In: *IEEE International Conference on Software Architecture (ICSA 2020)*. 2020. DOI: 10.1109/ICSA47634.2020.00011.

[5] Ralf H. Reussner et al., eds. *Modeling and Simulating Software Architectures – The Palladio Approach*. MIT Press, 2016. 408 pp. ISBN: 978-0-262-03476-0.

Collaborative software visualization with SEE

William Behnke
Dept. of Mathematics and Computer Science
University of Bremen
wbehnke@uni-bremen.de

Hannes Lennart Kuß
Dept. of Mathematics and Computer Science
University of Bremen
hkuss@uni-bremen.de

***Abstract*—SEE is a software engineering tool for visualizing software metrics based on the code-cities metaphor. It assists distributed teams in analyzing software collaboratively by offering multi-user functionality (including a voice chat), that allows team members to communicate naturally while investigating software. The tool utilizes graphs supplied in the Graph eXchange Language (GXL) format to represent software data, and allows users—among other things—to compare the current architecture with the original plan and to track changes of a software over time. One of our long-term goals is to enhance communication and collaboration among team members, to bridge spatial gaps, and to facilitate the understanding of software in (spatially separated) teams.**

I. Introduction

On a typical day, software developers spend around 82 minutes in meetings [1]. One of our goals in SEE is to reduce that time to the needed minimum. In software development, it is relevant to visualize different aspects of code in order to get a better understanding [2]. According to Cherubini et. al.: "developers produced visualizations: to understand, to design and to communicate." [2] However, the visualization of code is not easy and developers often miss this feature [2]. SEE (for Software Engineering Experience) is a tool developed by our research group to visualize software data in 3D using the code-city metaphor. [3].

Our tool focuses on co-operative understanding, enabling multiple people to understand and to communicate about software together. Our tool builds up on established 3D visualization platforms, specifically utilizing the Unity Engine. We empower developers to incorporate interactive capabilities, enabling multiple individuals to partake in virtual meetings, and facilitate information exchange within the context of software visualization.

SEE provides developers with enhanced abilities to comprehend the quality of their software, get a comprehensive overview, navigate complex architectural structures, and monitor the runtime behavior of their software. One of our goals is to support software development teams through the provision of an intuitive and informative environment. The development of SEE is being carried out at the University of Bremen in collaboration with Axivion [1], a company specialized in static code analysis and software architecture verification.

[1]https://www.axivion.com/

The focus of this paper is to introduce our project and, in particular, to emphasize the significance of incorporating emotional states within our software. Emotional state means the mood of the participating individual by analyzing their expressions and gestures.

In addition, we aim to present the future plans for our tool and provide further insight into the topic for other developers.

II. SEE Capabilities

The SEE platform is equipped with multiplayer functionality, which allows multiple users—represented as humanoid avatars—to concurrently interact with both the viewed software and each other. This includes actions such as moving the avatar or altering the location or size of components drawn within the *code city*, as well as the use of a voice chat to facilitate communication between users. This collaborative environment enables users to work together and discuss the software system being analyzed.

SEE represents a software system's architecture hierarchically (persisted as GXL graph), with each node in the graph representing a module of source code. Binary relations among components, such as function calls, are depicted as hierarchically bundled edges.

The incorporation of software metrics into the graph is achieved through the use of additional visual attributes of the shapes—such as depth, height, width or color—and additional decorations, e.g., antennas above blocks. The layout of the *Code city* is customizable and can be automated by different types of hierarchical graph layouts (e.g., treemaps, EvoStreets, rectangular and circular packing). An example of a *Code city* can be seen in Figure 1. The SEE platform enables—among others—the visualization of the current state of a software architecture through the representation of components as nodes and dependencies as edges connecting them. This representation can be compared to the original architecture plan, allowing users to identify discrepancies between the intended specification and the implemented architecture. This comparison facilitates a clear understanding of deviations from the original design and allows for the identification of potential issues or areas for improvement.

In addition to this, SEE offers an "evolution view" feature which allows users to view the historical evolution of a project, providing a comprehensive overview of the changes that occurred over time and identifying the

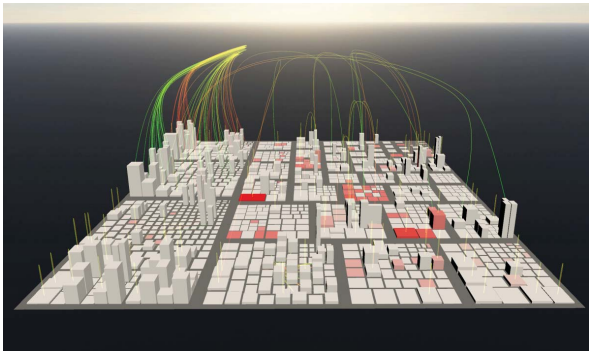


Fig. 1. Screenshot of an example *Code City*.

components that have undergone the most significant alterations. SEE also includes a built-in code viewer with the capability to synchronize the current line of code being viewed among all participants in a virtual meeting, enabling all attendees to easily reference the code being discussed.

III. Enhancing Nonverbal Communication

The SEE platform originated as a student project, initially utilizing the Unreal Engine as a base. After further development, we transitioned to the Unity Engine, as it offers greater versatility in terms of deployment across various platforms, and the ability to utilize a wider range of tools and community-driven products.

One of the most promising focus of development for the SEE platform is the improvement of non-verbal communication through capturing and displaying user emotions and gestures. To achieve this, we are experimenting with the use of the HTC Facial Tracker [2] and OpenCV [3] to monitor and mirror facial states. The incorporation of facial tracking technology into code review processes has several potential benefits. One of the key advantages is the ability to analyze nonverbal cues such as expressions and gestures, which can provide additional information about the emotional state of the individuals involved in the process.

In psychology it is well known, that nonverbal communication is as important as verbal communication. Hence, the incorporation of nonverbal communication in SEE can lead to more efficient and effective communication and collaboration among team members, as they are able to more easily interpret the reactions of their colleagues. Additionally, this may also aid in identifying areas of confusion or difficulty, allowing for more focused and productive discussions.

Furthermore, this feature can be especially beneficial for managers or supervisors who may not be profoundly involved in the technical aspects of the software development process. It allows them to more easily identify and address issues related to team dynamics and communication, improving the overall performance of the development team. By offering a clearer insight into the emotional state of the team, managers and supervisors can effectively address and resolve any conflicts or challenges that may arise during the development process, ultimately improving time management. In most project contexts, emotional states refer to the emotional well-being of software developers.

Overall, incorporating facial tracking technology into code review processes has potential to improve time management and increase the efficiency and effectiveness of software development teams. This can lead to more productive and successful software development projects, ultimately resulting in better software and more satisfied customers.

[2]see https://www.vive.com/de/accessory/facial-tracker/

[3]see https://opencv.org/opencv-face-recognition/

IV. Upcoming Features

In this paper, we discussed some key aspects for the continued development of SEE. Additionally, we also identified other areas of interest for further exploration, such as:

- *Improving edge visualization using different types of animations:* Improving the visualization of edges between different components of the software, by improving animations to more clearly represent relationships and dependencies.
- *Runtime configuration:* Providing more flexibility and control for users by allowing them to configure the visualization at runtime, tailoring it to their specific needs and preferences.
- *Live documentation:* Incorporating live documentation within SEE, to provide a more complete and integrated understanding of the software.

V. Conclusion

This paper presents SEE, a software visualization tool that facilitates software comprehension by means of Code Cities. We discussed the technology used in the development of SEE to demonstrate our plans for future implementations, and enhancing nonverbal communication by capturing and displaying user emotions and gestures. More information about SEE can be found on our website: https://see.uni-bremen.de/.

References

[1] A. N. Meyer, E. T. Barr, C. Bird, and T. Zimmermann, "Today was a good day: The daily life of software developers," *IEEE Transactions on Software Engineering*, vol. 47, no. 5, pp. 863–880, 2021.

[2] M. Cherubini, G. Venolia, R. DeLine, and A. J. Ko, "Let's go to the whiteboard: How and why software developers use drawings," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 557–566. [Online]. Available: https://doi.org/10.1145/1240624.1240714

[3] R. Wettel and M. Lanza, "Codecity: 3d visualization of large-scale software," in *Companion of the 30th International Conference on Software Engineering*, ser. ICSE Companion '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 921–922. [Online]. Available: https://doi.org/10.1145/1370175.1370188

Towards Statically Checking Adherence to API Protocols

Jochen Quante
Robert Bosch GmbH, Corporate Research
Renningen, Germany
jochen.quante@de.bosch.com

Sushmita Suresh Naragund
TU Kaiserslautern
Kaiserslautern, Germany

Abstract

API protocols specify sequence constraints on API calls. They are typically available in form of finite state machines. Traditionally, API protocols are checked during runtime only: With each API call, the state in the state machine is tracked. If this leads to an error state (or an unsupported operation in a given state), the protocol is violated. However, it would be much more desirable to check adherence to the protocol statically, i.e., prior to execution of the code. In this paper, we report on our endeavors and experiences on doing such checks statically.

1 API Protocols

A given API typically comes with some assumptions on how it will be used. For example, consider a file I/O API. A file has to be opened before it can be read, and no more data can be read from it once it has been closed. A typical API protocol for file operations thus may look like this:

```
(fopen (fread|fwrite|fseek)* fclose)*
```

This regular expression specifies all allowed sequences of operations on this API. We treat API functions as the basic symbols, so valid function call sequences are the words of that language. Like any regular expression, this can be transformed to a deterministic finite automaton (DFA).

Of course, not all API protocols are expressible in a regular language. For example, it is not possible to specify that in a stack component, `pop` may only be called as many times as `push` has been called before: That would require a context-free language. We still stick to regular expressions due to practicability reasons; it is also the standard in related work on the topic [3]. For the stack example, we can still come up with a useful API protocol: It can ensure that we only `pop` when we know that there is something on the stack.

```
(push | isEmpty | isEmpty pop | push pop )*
```

2 Checking Protocol Adherence

We now want to check whether a given application that uses the API adheres to the protocol, i.e., if it obeys its sequence restrictions. Let us assume that we have another automaton that represents all possible API call sequences, which has been extracted from the code. Then our task can be rephrased as comparing the languages represented by the two automata.

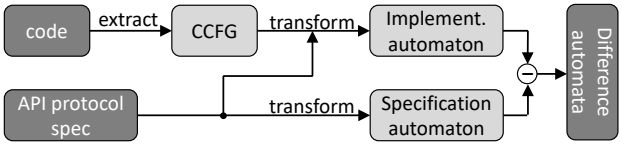


Figure 1: Approach

Let us denote the language that a DFA A accepts by $L(A)$. The difference of the languages of two DFAs A and B is computed as $L(A)-L(B) = L(A \cap \overline{B})$. The complement of a DFA is derived by making accepting states non-accepting and vice versa. The intersection of two automata is computed by product construction. With these standard ingredients, we can calculate the language differences between protocol automaton P and implementation automaton I. They can be interpreted in the following way:

$L(I) \subseteq L(P)$	The API is correctly used.
$L(I) \not\subseteq L(P)$	The code violates the protocol.
$L(P) \subseteq L(I)$	The code uses the entire protocol.
$L(P) \not\subseteq L(I)$	The code does not use everything that is allowed by the protocol.

Furthermore, the corresponding difference automata (when non-empty) can provide hints about what exactly the violation or unused feature is. This can help a developer in quickly finding and fixing the underlying problem.

3 Simple Implementation Automaton

For a first evaluation of this idea, we implemented a simple static implementation automaton extraction approach and the automaton transformations as described above. It is sketched in Fig. 1 and works like this:

First, the control flow graph for each function is taken on basic block level. Each basic block is represented as a sequence of two nodes (entry and exit), and these nodes are connected according to the control flow between blocks. For basic blocks that con-

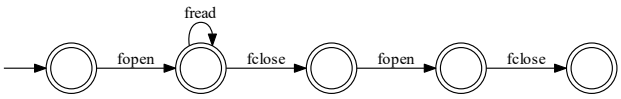


Figure 2: Initial extracted automaton.

tain calls, this sequence (of two nodes) is then transformed to a sequence of call and return edges. After this has been done for all functions, inlining of functions is performed: Each pair of call/return edges is supplemented by the graph of the respective function (if it exists, and if it is not a recursive call). Finally, unnecessary intermediate nodes and edges (subgraphs that contain no calls at all) are removed. The resulting graph is a combined call and control flow graph (CCFG) for a given function. Note that this graph may contain infeasible paths – it is an overapproximation of all possible paths through all function calls.

To convert this graph to an automaton, we create a non-deterministic automaton where each node is a state and each edge is a transition. Edges that correspond to a function call of the API under consideration become transitions labeled with the respective symbol, all other edges become ε edges. Only the exit node becomes an accepting state. Then, the automaton is converted to a deterministic one using subset construction. The result is an automaton that contains all API call sequences that can potentially occur in the code, along with infeasible ones.

Two problems occur for this very simple approach: Firstly, it does not distinguish between different instances of API usages, e.g., when different files are processed in parallel. Secondly, it abstracts away a lot of information so that the results are quite imprecise. However, the general transformation from CCFG to automaton can also be applied on the results of more advanced extraction algorithms. For example, object process graphs could be used instead [1] to increase precision.

4 Example

To evaluate this approach, we took the file API protocol from above and used a small compiler written in C as subject system. The compiler reads an input source file and ultimately writes an output binary file. We extracted the CCFG for the compiler code and translated it to an FSM. The result is shown in Fig. 2. Apparently, the sequence of operations can stop in any step. A quick check in the code reveals that this is due to error handling. So we add error handling to our protocol: After any operation, we may jump to an error final state. We choose `log_error` as the function that indicates this, so we enforce error logging in every error case. The resulting code automaton consists of 14 states, and the resulting difference automaton $I - P$ is partly shown in Figure 3. The difference

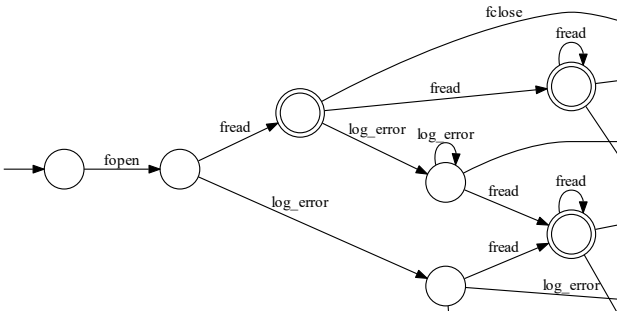


Figure 3: Difference automaton (implementation automaton minus protocol automaton) shows violations.

automaton is non-empty, which means the extended protocol is violated, and the language of the automaton consists of all violations. We can thus easily see from the Figure that, e.g., `fread` may be called after a `log_error`, and that the program may terminate after `fread` without logging. However, all that is subject to the limitation that these may be infeasible paths (in fact, most of them are). Furthermore, it turns out that `log_error` is too general to be used in the file I/O protocol – it is used in many other error cases as well.

5 Protocol Recovery

The technique introduced above can also be used to interactively reconstruct API protocols from code, in a way similar to the well-known reflexion analysis [2]: The user formulates an API protocol hypothesis, checks it against the code, gets feedback in the form of difference automata, adjusts the hypothesis, and iterates until the result is satisfactory.

6 Conclusion

We introduced a lightweight technique for checking API protocols against code. The technique can also be used to infer such protocols iteratively and thus support in program understanding. Despite the imprecise static extraction technique used, the results still give valuable insights into the analyzed program. We therefore conclude that the approach has the potential to be useful in practice.

References

[1] T. Eisenbarth, R. Koschke, and G. Vogel. Static object trace extraction for programs with pointers. *Journal of Systems and Software*, 77(3):263–284, 2005.

[2] G. C. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *Proc. of 3rd Symp. on Foundations of Software Engineering*, pages 18–28, 1995.

[3] M. Pradel, P. Bichsel, and T. R. Gross. A framework for the evaluation of specification miners based on finite state machines. In *Proc. of Int'l Conf. on Software Maintenance*, pages 1–10, 2010.