Dr. Stephan Grünfelder
A-3643 Maria Laach
stephan.gruenfelder@gmx.at
methver.webs.com

Dipl-Ing. Christoph Luckeneder
TU Wien
Institut für Computertechnik
Christoph.luckeneder@tuwien.ac.at

Testreihung mit allen Tricks

Reihung von automatischen Tests für komplexe Embedded-Systeme

Von Stephan Grünfelder und Christoph Luckeneder

Viele Firmen haben über mehrere Jahre automatisierte (Firmware-)Systemtests akkumuliert. Die Tests laufen oft auf proprietären, teuren Anlagen. Auch automatische Testdurchführung benötigt Zeit, mitunter viel Zeit. Wenn der Zeibedarf für einen vollständigen Durchlauf hoch wird, dann wird man versuchen die Tests so zu reihen, dass die Tests mit der höchsten Fehlerfindungswahrscheinlichkeit zuerst drankommen. Das ist zwar naheliegend, aber oft gar nicht so einfach. Das übergeordnete Ziel ist, die Zeit vom Start der Testsuite bis zum ersten Bugreport zu minimieren. Dieser Artikel beschreibt, wie man hier vorgehen kann. Es wird ein einfacher Lösungsansatz vorgestellt, der selbst für verteilte Echtzeit-Systeme anwendbar ist.

Ein Steuergeräte-Hersteller aus Bayern kämpft mit einer Herausforderung, die den Zweck der Technik, über die dieser Artikels berichtet, hervorragend illustriert: Die Testausführungszeit der mehreren Tausend Systemtests des Herstellers ist bereits länger als eine Woche. Und das, obwohl die Firma bereits mehrere Testsysteme parallel im Einsatz hat. Das Unternehmen hat aber den Anspruch von Continuous Testing – vom Testteam will man innerhalb von weniger als 24 Stunden eine qualifizierte Rückmeldung zu jeder Code-Änderung erhalten. Eine weitere Vervielfachung der bestehenden Testsysteme scheidet aus Kostengründen aus. Der Firma ist es vor etwas mehr als drei Jahren mit Hilfe von Data Mining und Künstlicher Intelligenz in Zusammenarbeit mit der TU München gelungen, einen Test-Scheduler zu schreiben, der mit hoher Wahscheinlichkeit die Tests zur Ausführung zuerst auswählt, die tatsächlich einen Fehler finden könnten. Somit gelang es tatsächlich in weniger als 24 Stunden *mit hoher Wahrscheinlichkeit* nach einer Code-Änderung alle neuen Fehler zu finden ohne weitere Testsysteme anzuschaffen. Das Unternehmen hat im Laufe der Zeit dann die KI-Methoden durch die sogenannte *Test-Impact-Analyse* ersetzt, die in diesem Artikel noch näher vorgestellt wird. Diese Analyse erfordert es, beim momentanen Stand der Technik, den gesamten Quellcode zu instrumentieren und die Testabdeckung von Systemtests zu erfassen. Das kann sehr aufwändig, oder, bei verteilten Systemen oder Systemen, die mir FPGAs realisiert sind, de facto unmöglich werden. Die folgenden Zeilen präsentieren einen einfacheren alternativen Lösungsansatz, der ohne Künstliche Intelligenz und ohne Instrumentierung mit Hilfe von Statistiken arbeitet und sich der (genaueren) Test-Impact-Analyse annähert.

Rahmenbedingungen des Test-Harness

Der alternative Lösungsansatz wurde in der Firma Riedel Communications am Standort Wien entwickelt. Wie auch der das Testteam des bayrischen Steuergeräteherstellers kämpft man hier oft mit subtilen Bugs in Messgeräten und Funktionsgeneratoren. Daher hat das Team zusätzlich zu den Testresultaten PASSED und FAILED auch das Resultat ERROR eingeführt. Dieses dritte Urteil bedeutet, dass ein Messgerät nicht antwortet oder unplausible Werte liefert und daher kein zuverlässiges Testurteil abgegeben werden kann. Ein Beispiel für so einen Error wäre, wenn das über Ethernet ferngesteuerte Oszilloskop nicht antwortet oder statt eines

Messwertes *NaN* (not a number) liefert. Es kann sein, dass das System Under Test dennoch fehlerfrei ist, es kann aber genauso gut das Gegenteil der Fall sein.

Das Testteam in Wien speichert zusätzlich zur Sicherung der Testlogs auf einem File-Server das Resultat jeder Testdurchführung, das Datum, die Version der Firmware-Under-Test und die Testlaufzeit in einer SQL-Datenbank. Das ist eine ganz wichte Voraussetzung dafür, dass die Testreihung auf Basis von statistischen Auswertungen erfolgen kann.

Bevor ein Stapel von Systemtests gestartet wird, überprüft die Testumgebung mit Hilfe von Funktionsgeneratoren, Messgeräten und (De)multiplexern, ob alle Kabel korrekt angesteckt sind und ob die Messgeräte erwartungsgemäß reagieren. Das ist eine wichtige Voraussetzung dafür, dass man keine falschen Testurteile erhält, weil sich ein Kollege das Oszilloskop ausgeborgt hatte und nach dem Zurückbringen falsch angesteckt hat.

Rahmenbedingungen beim Bug-Lifecycle

Für jeden Bug, den das Testteam findet, wird ein eigenes Ticket in einem Bug-Tracker angelegt (in konkreten Fall ist das *Jira*). Der Bug-Tracker ist mit der Code-Versionsverwaltung (im konkreten Fall *Bitbucket*) integriert. Das Testteam notiert in jedem Bugreport jene Tests, die wegen des Bugs fehlschlagen. Ein Beispiel ist in Abbildung 1 zu sehen. Ursprünglich wurde das gemacht, um die erste Überprüfung des Bugfixes zu erleichtern: die im Ticket notierten Tests (meist ist es aber nur einer) werden ausgeführt um den Bugfix zu verifizieren.

Wenn das Entwicklungs-Team einen Bugfix ins Code-Repository einpflegt, dann wird die Ticket-Nummer des Bugs beim Commit in die Code-Versionsverwaltung angegeben. Gleichsam die „Begründung“ für eine Code-Änderung ohne neuen Entwicklungsauftrag. Durch die Integration der Versionsverwaltung mit dem Bugtracker ist die für den Bug-Fix nötige Code-Änderung auf diese Weise im Bugticket ersichtlich.

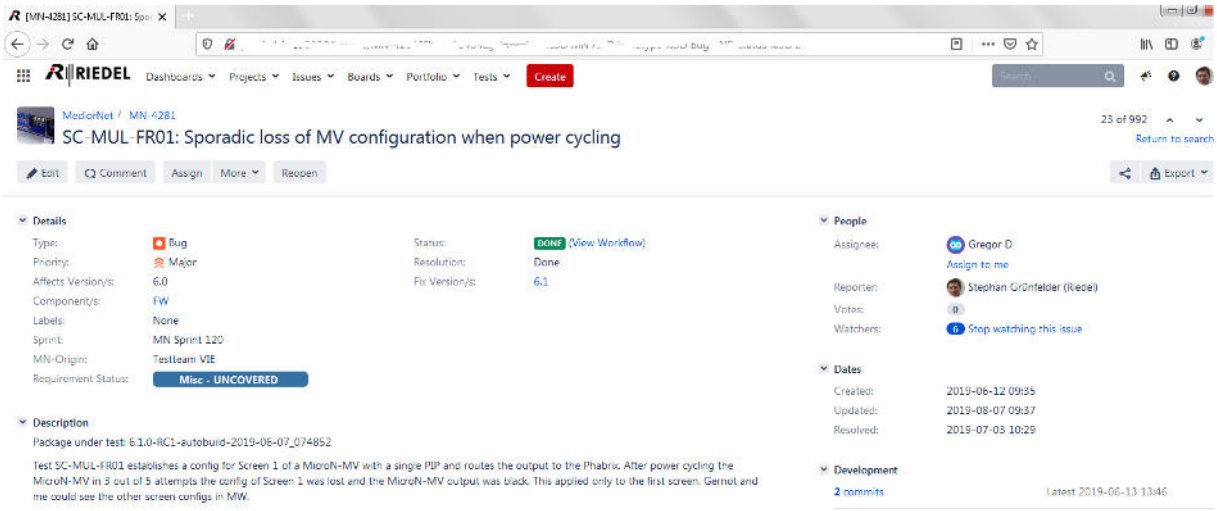

Bild 1: Ein typischer Bugreport bei Riedel Communications enthält im Titel die ID des Tests, der den Bug aufgedeckt hat. Zudem findet man unter dem Link „Commits“ die Code-Änderungen, die den Bug beheben.

Strategien zur Testpriorisierung

Aktuelle akademische Publikationen zum Thema Testreihung (engl. *Test Case Priorization, TCP*) nennen verschiedene Strategien zur Reihung von Tests. Unter anderem das schnelle Erreichen einer hohen Coverage von Anforderungen oder das schnelle Erreichen einer hoher Coverage von Code [1]. Beide Strategien mögen für eine allererste Testdurchführung gut anwendbar und nützlich sein, haben aber gravierende Nachteile im viel häufigeren Anwendungsfall: beim Regressionstest nach einer Code-Änderung oder -Erweiterung. Eine schnell erreichte hohe Abdeckung von Anforderungen oder Code muss nicht heißen, dass schnell (neue) Fehler entdeckt werden, denn die durch diese Strategie zuerst abgedeckten Code-Teile könnten völlig entkoppelt von den letzten Code-Änderungen sein. Für den Regressionstest ist eine Testreihung gefragt, die jene Tests zuerst aufruft, die die höchste Wahrscheinlichkeit haben *neu* eingebrachte Fehler zu finden. Die meisten wissenschaftlichen Publikationen zum Thema Test-Effizienz drehen sich daher um die Reihung der Tests im Regressionsfall.

Um Reihungen von Tests zu vergleichen, hat sich in akademischen Publikationen das Maß *Average Percentage of Faults Detected* etabliert

$$APFD = 1 - \frac{TF1 + TF2 + \cdots + TFm}{nm} + \frac{1}{2n}$$

In dieser Formel ist m die Anzahl der auffindbaren Fehler im System und n die Anzahl der Tests. TF_i ist die Position des Tests der Fehler i aufdeckt [2]. Position in der Testreihung. Diese Maßzahl erreicht dann 100%, wenn zuerst alle Tests laufen, die Fehler finden, und erst danch die „Blindgänger-Tests“. APFD berücksichtigt aber keine Testkosten, also die Testlaufzeit. Diese kann beträchtlich zwischen den Tests variieren. Um auch die Laufzeiten zu berücksichtigen, wurde das Maß APFDc, Average Percentage of Faults Detected per Cost, ins Leben gerufen. Eine riesige, komplizierte Formel. Aber auch diese Formel kann zum Beispiel die Schwere eines Fehlers nicht berücksichtigen. Es ist daher für den industriellen Anwender schwer zu sagen, ob eine Testreihung, die einer anderen um drei Prozentpunkte überlegen ist, tatsächlich die nützlichere Reihung ist.

Test-Impact-Analyse

Ohne Zweifel haben in iterativen Projekten jene Systemtests das größte Potenzial Fehler aufzudecken, die neuen bzw. geänderten Code duchlaufen. Um einen Zusammenhang der (Black Box) Systemtests mit dem Code Unter Test herzustellen, wird typischerweise bei Werkzeugen zur Test-Impact-Analyse der Code zunächst instrumentiert. Das heißt es werden dem Code Under Test automatisch spezielle Erweiterungen hinzugefügt, die gestatten die durchlaufenen Quellcode-Zeilen zu rekonstruieren. Die Code-Coverage der System-Tests wird auf diese Weise bei der Testdurchführung „vermessen“. Wenn sich der zu testende Code nun ändert, dann können die *impacted tests* mit Hilfe dieser Messungen ermittelt werden. Das sind jene Tests, die von der Änderung betroffene Code-Teile durchlaufen. Genauer gesagt durchliefen – denn die Vermessung ist ja durch die letzte Änderung nun nicht mehr aktuell.

Die Tests gemäß einer Test-Impact-Analyse zu reihen ist ein bestechendes Konzept. Es gibt aber Arten von Software-Änderungen, bei der dieses Vorgehen zahnlos ist:

- Daten/Konfigurations-Änderungen
- Änderungen des Software-Builds

Zudem gibt es Systeme, für die das Verfahren schlecht oder garnicht anwendbar ist:

- Wenn das System-Under-Test aus vielen Executables besteht (also z.B. in Form von MicroServices implementiert ist oder ein verteiltes System ist): Sollte man alle Coverages erfassen?
- Für Eingebettete (Echtzeit-)Systeme, bei denen Code-Instrumentierung unmöglich ist, lässt sich diese Technik nur mit erheblichen Schwierigkeiten einsetzen.
- Das Konzept ist nicht einfach bei Verwendung von Hardware-Beschreibungssprachen anwendbar. Wer FPGAs verwendet, muss sich also nach einer alternativen Lösung umsehen.

Das SUT des wiener Testteams ist ein verteiltes Echtzeitsystem mit FPGAs. An eine out-of-the-box-Verwendung eines Werkzeugs zur Test-Impact-Analyse ist nicht zu denken.

Näherungsverfahren

Die Stärke der Test-Impact-Analyse ist, die impacted Tests durch die Vermessung der Tests und der Analyse der Code-Änderungen zu ermitteln und eine „optimale" Reihung vorzuschlagen. Das ist zum Beispiel eine Reihung gemäß Abeckung von neuem Code dividiert durch Testlaufzeit. Man kann sich der optimalen Testreihung aber auch mit Hilfe von Statistiken nähern und folgenden Tests höhere Priorität als anderen geben:

- Tests, die erst seit kurzem existieren. Sie testen vermutlich neue Features und damit neuen Code.
- Tests, die in der jüngsten Vergangenheit fehlschlugen. Die dabei aufgedeckten Bugs könnten nun ausgebessert sein. Damit sind diese Tests für den Regressiontest heiße Kandidaten, denn es wird ggf. veränderter Code durchlaufen.
- Tests, die immer schon erfolgreicher als andere beim Aufdecken von Bugs waren [3].

Mit diesen einfachen Mitteln ist die Reihung gewiss effektiver als eine reine Zufallsreihung. Doch die zuvor erwähnten Rahmenbedingungen erlaubten noch weitere Ideen zu implementieren. Idee Nummer eins ist die Zuordnung von Quellcode des SUT zu Systemtests auf Basis von mit dem Bugtracker gesammelter Daten.

Wie erwähnt, und in Abbildung 1 beispielhaft zu sehen, lässt sich eine moderne Quellcode-Verwaltung mit dem Bugtracker integrieren. Es besteht also eine Verbindung vom Bugticket, das den Fix getriggert hat, zum Quellcode, wo der Fix implementiert wurde. Im Bugticket steht aber auch die ID des Tests drin, der den Bug aufgedeckt hat. Das wiener Testteam hat damit aus dem Bugtracker unzählige Verknüpfungen von Tests zum Code gesammelt, ganz ohne Instrumentierung, siehe Abbildung 2. Die große Datenmenge (es waren leider viele Bugs) versetzt das Team nun in eine ähnlich günstige Lage, wie sie die Testvermessung der Test-Impact-Analyse erlaubt. Das Testsystem kann (automatisiert über eine API) von der Versionsverwaltung die letzten Code-Änderungen abfragen und danach aus der Liste der Verknüpfungen jene Tests heraussuchen, die in den geänderten Code-Blöcken schon irgendwann einmal einen Bug aufgedeckt haben. Das sind natürlich Tests, die unbedingt bevorzugt laufen müssen.

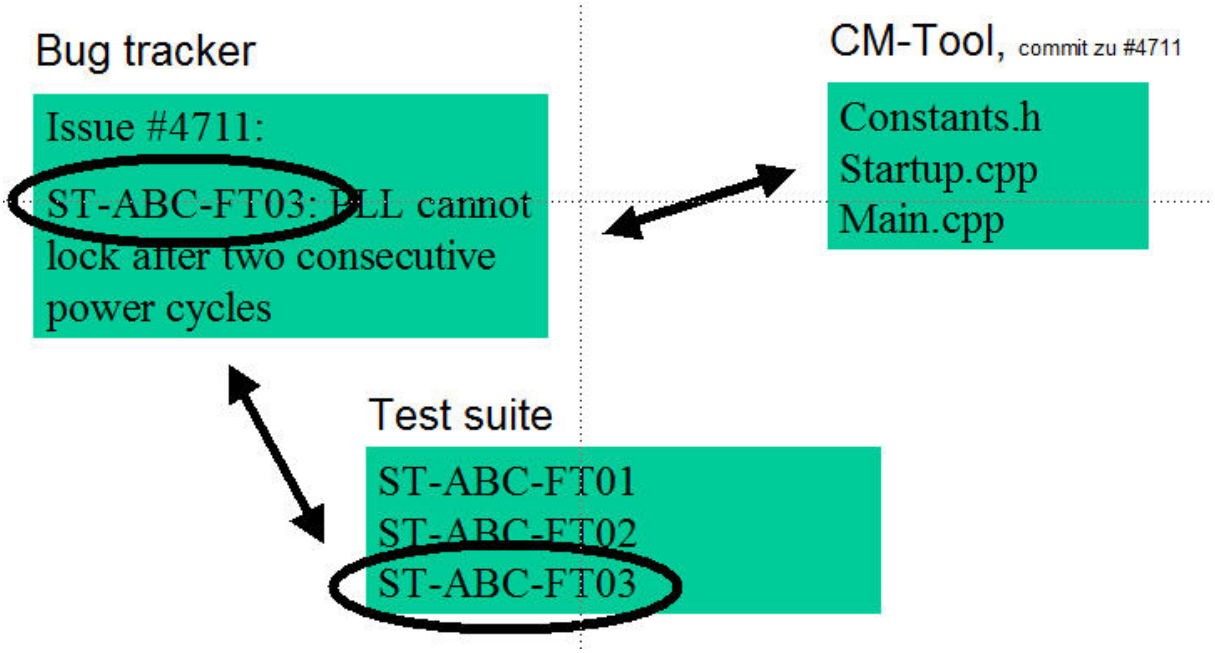

Bild 2: Die Integration des Bugtrackers mit der Quellcode-Verwaltung (CM-Tool) erlaubt eine kleine aber sinnvolle Auswahl von Impacted Tests.

Natürlich gibt es durch diese Art der Ermittlung von Impacted Tests nicht annähernd eine Garantie einer Vollständigkeit. Das schöne an diesem Verfahren ist aber, dass es auch für FPGA-Code sinnvoll anwendbar ist. Insgesamt kommen also nun schon 4 Kriterien parallel zum Einsatz, die beim Start der Testumgebung *statisch* Testprioritäten vergeben: (1) das Alter des Tests, (2) Fehlschläge in jüngester Vergangenheit, (3) historisch erfolgreiche Tests und (4) Tests, die im gerade geänderten Code schon einmal einen Fehler fanden.

Auf Basis dieser Kriterien erhöht die Testumgebung beim Start eines nächtlichen Testlaufs automatisch ggf. die Prioritätszahl eines Tests. Tests mit gleich hoher Priorität reiht die Testumgebung zufällig. Doch ganz ausgereizt hat man das Potenzial der gesammelten Daten noch nicht. Der Testreihung kann man noch ein wenig „Intelligenz“ verleihen, indem man eine „dynamische“ Komponente hinzufügt.

Repriorisierung zur Effizienzsteigerung

Bei Riedel Communications werden mehrere Varianten einer Produktfamilie getestet. Wenn das Testteam nach einem nächtlichen Testdurchlauf einen gefundenen Fehler mit den Entwicklern durchspricht, ist oft eine der ersten Fragen der Entwickler, welche anderen Produktvarianten vom gleichen Problem betroffen sind. Vor der Umsetzung der nun vorgestellten Idee hatte das Testteam nicht immer eine Antwort parat, denn die Tests waren zufällig gereiht und es konnte durchaus sein, dass ein fehlgeschlagener Test nur für eine einzige Produktvariante über Nacht lief. Um hier besser aufgestellt zu sein, hat das Testteam nach Impulsen in wissenschaftlicher Literatur gesucht.

In [3] wird beschrieben, dass ein Testsystem auf Basis eines Testurteils die verbleibenden, noch nicht ausgeführten Tests, neu reiht. In [3] werden Korrelationen der Testurteile dafür herangezogen um beim Testurteil FAILED mit hoher Wahrscheinlichkeit einen neuen Test zu wählen, der ebenfalls fehlschlägt, siehe Abbildung 3. Die Motivation der entsprechenden akademischen Publikationen ist aber nicht automatisiert bessere Daten für eine sinnvolle

Bewertung eines Bugs zu liefern. Die Autoren wollen besseren Werte für APDF erreichen (und waren mit ihrer Strategie erfolgreich). Der Weg für beide Ziele ist aber der gleiche: wenn ein Test fehlschlägt, dann sollte man anderen Tests (dynamisch) eine höhere Priorität geben, die *wegen* dieses Fehlschlags voraussichtlich auch eher fehlschlagen.

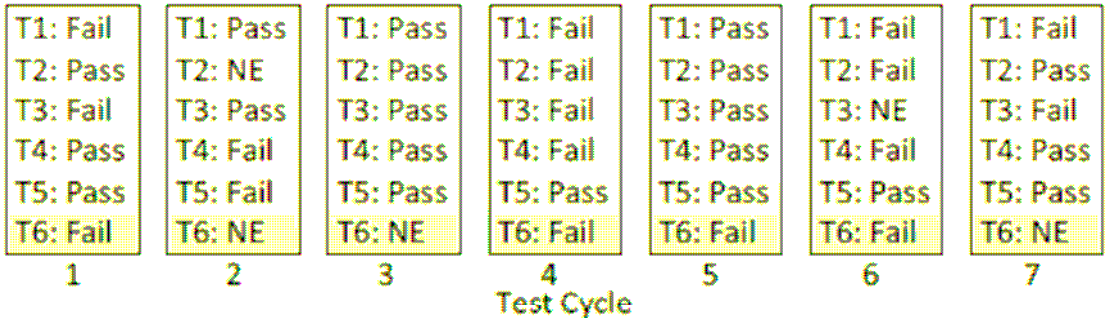

Bild 3: Nur dann wenn es viele Testzyklen gibt und die Anzahl der nicht durchgeführten (NE, not executed) Tests klein ist, kann man mit Hilfe von Korrelationsanalysen oder mit KI-Methoden die Testreihung dynamisch sinnvoll adaptieren.

Um dem eigenen Testsystem nun die nötige „Intelligenz“ zu geben nach einem Testfehlschlag für Produktvariante A einen möglichst ähnlichen Test (für eine andere Produktvariante oder auch für A selbst) auszuwählen, muss das Testsystem die Ähnlichkeit von Tests feststellen. Man bedient sich also *nicht* der Korrelation der historischen Testergebnisse von Tests, wie in Abbildung 3 zu sehen, sondern nimmt „handfestere“ Daten. Zur Feststellung der Ähnlichkeit von Tests wird in der Literatur vorgeschlagen mit Similarity-Metriken die textuelle Beschreibung der Tests, den Quellcode von Tests oder den von den Tests durchlaufenen Code zu vergleichen. Die so zu vergleichenden Strings sind sehr lang und die erwähnten Metriken, die diese Strings vergleichen, sind beliebig komplex. Die auf diese Weise zu machenden Berechnungen sind manchmal so zeitaufwändig, dass sie für den industriellen Einsatz nicht praktikabel sind [4]. Doch die Implementierung eines schnellen Vergleichs von Tests für eingebettete Systeme ist oft einfacher als gedacht.

Unterschriftenvergleich

Tests für eingebette Systeme folgen sehr oft dem gleichen Schema: Setze das System in einen definierten Initialzustand, setze Parameter im SUT, stelle einen Funktionsgenertor und/oder ein Messgerät ein, Messung, setze weitere Parameter im SUT, stelle einen Funktionsgenertor und/oder ein Messgerät ein, Messung u.s.w.

Das Setzen von Parametern erfolgt bei Riedel Communications typischerweise über Ethernet mit dem Protokoll Ember+ [5]. Jeder Parameter hat eine ID. Zum Beispiel „phaseOffset“. Diese ID kann aber in mehreren Geräten des verteilten Systems vorkommen. Um einen Parameter eines bestimmten Geräts anzusprechen, wird die Geräteadresse mit der Parameter-ID kombiniert. Die Lösung liegt fast auf der Hand: Die Ember+ Schnittstelle wurde so erweitert, dass jeder Test in einem separaten Logfile eine Signatur hinterlässt, die aus einer Kette der verwendeten Parameter-IDs samt gesetzter Parameterwerte besteht. Es entsteht ein String, der deutlich kürzer ist, als von den zuvor angedeuteten Verfahren, und gleichzeitig die Natur des Tests haarscharf charakterisiert. Auf Basis dieser Signaturen stellt das Testsystem die Ähnlichkeit von Tests fest.

Es gibt, wie gesagt, eine ganze Reihe von ausgeklügelten Verfahren, die die Ähnlichkeit von Strings bewerten. Sie werden in Suchmaschinen für das Erkennen von vermutlich gesuchten Begriffen bei Tippfehlern, für Data Mining, für Plagiatsprüfungen, u.v.m. verwendet. Für den Zweck des Vergleichs von Tests reicht eine ganz einfache Similarity-Metrik: die Jaccobian Similarity.

$$J(A,B) = |A \cap B| / |A \cup B|$$

Diese Maßzahl ist dann 100%, wenn beide Strings exakt das gleiche Vokabular verwenden. Ein String wird dabei als Menge (im Sinne der Mengenlehre) verstanden. Die Elemente dieser Menge sind in den Signatur-Logs die Tupel (Parameter-ID, Parameterwert). Das bedeutet im konkreten Fall: Wenn zwei Tests A und B die gleichen Parameter auf die gleichen Werte setzen (aber ggf. in anderen Geräten), dann ist J(A,B) = 100%.

Um nun dem Test-Scheduler die erwähnte „Intelligenz“ zu verleihen, wurde er so adaptiert, dass er dynamisch Prioritäten verändern kann. Nach jeder Testdurchführung bewertet das Testsystem die Prioritäten der noch nicht ausgeführten Tests neu:

- Wenn das Testurteil FAILED ist, dann wird der ähnlichste Test gesucht und dessen Priorität dramatisch erhöht.
- Wenn das Testurteil ERROR ist, dann werden die 10 ähnlichsten Tests gesucht und deren Priorität reduziert. Dies geschieht um keine Zeit durch Fehler der Testumgebung zu verlieren.
- Wenn das Testurteil PASSED ist, dann wird der ähnlichste Test gesucht. Wenn dessen Priorität nicht durch eine andere Bewertung hinaufgesetzt wurde, wird sie nun geringfügig herabgesetzt. Dies geschieht um Diversität im Test-Set zu erhöhen[1].

Damit die Bewertung der Ähnlichkeit auch gut bei kurzen Tests funktioniert, muss man „Gleichanteile“ von Tests aus den Signaturen entfernen. Bringen alle Tests zuerst alle Geräte in einen Initialzustand, bevor sie ihre eigenen Wege gehen, so ist diese Initialisierung der erwähnte Gleichanteil. Das Streichen dieser Initialisierung aus allen Signaturen erhöht die Aussagekraft von J(A,B) bei kurzen Tests deutlich.

Zusammenfassung

Es wurde ein Verfahren vorgestellt, dass sich mit Hilfe von einfach zu ermittelnden Daten einer Test Impact Analyse annähert und diese um eine dynamische Repriorisierung erweitert.

Jede ordentliche wissenschaftliche Publikation würde in der Zusammengfassung den Leser nun mit eindrucksvollen APFD-Vergleichen in einigen bunten Box-Charts beeindrucken. Das wird man in diesem Artikel nicht finden. Die hier vergestellte Testreihung erhebt keinen besonders wissenschaftlichen Anspruch. Im Gegenteil: Sie zeigt, dass man auch ohne künstliche Intelligenz, ohne multivariane Optimierung und ohne superkomplizierte Data-Mining-Algorithmen die Reihung von automatischen Tests deutlich verbessern kann.

Doch Vorsicht: Kein automatisches Verfahren, auch keine Test Impact Analyse auf Basis von Code-Änderungen, kann Tests mit 100%iger Gewissheit ausschließen. Eine Regressiontestplanung für ein Produktrelease sollte daher immer „per Hand“ gemacht werden.

---

[1] In Abbildung 3 wird die Idee eine Korrelationsanalyse skizziert. Auch solche Analysen können zur dynamischen Repriorisierung verwendet werden und die Diversität zu erhöhen. Wenn zwei Tests etwa stark positiv korrelieren und der eine gerade mit dem Testurteil PASSED terminiert, dann würde man die Priorität des anderen Test verringern, also den anderen Test später reihen.

Ausblick

Das Verfahren hat das Team des Erstautors seit Anfang 2020 in Einsatz. Implementiert wurde es in C#. Der Quellcode kann unter gewissen Bedingungen von interessierten Personen eingesehen werden. Als nächste Schritte denken wir daran unserem Test-Scheduler noch weitere stategische Intelligenz mit Hilfe von Reinforcement Learning zu verleihen.

Danksagung

Der Erstautor bedankt sich bei Prof. Hermann Kaindl vom Institut für Computertechnik der TU Wien für seine Einladung ein wissenschaftliches Projekt zu begleiten. Die Entstehung des hier beschriebenen Verfahrens ist ein unerwareter Spin-Off dieser Zusammenarbeit.

Referenzen

[1] Zhe Yu, Fahmid Fahid, Tim Menzies, Gregg Rothermel, Kyle Patrick, and Snehit Cherian: “TERMINATOR: Better Automated UI Test Case Prioritization”. *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE ’19)*, August 26–30, 2019, Tallinn, Estonia. ACM, New York, NY, USA. https://doi.org/10.1145/3338906.3340448[2] Rothermel, G., Untch, R. H., Chu, C., & Harrold, M. J. (2001). Prioritizing test cases for regression testing. *IEEE Transactions on software engineering*, 27(10), 929-948.

[3] Dipesh Pradhan, Shuai Wang, Shaukat Ali, Tao Yue, Marius Liaaen: "REMAP: Using Rule Mining and Multi-Objective Search for Dynamic Test Case Prioritization". *2018 IEEE 11th International Conference on Software Testing, Verification and Validation*.

[4] Breno Miranda, Emilio Cruciani, Roberto Verdecchia, and Antonia Bertolino. 2018. FAST Approaches to Scalable Similarity-based Test Case Prioritization. *In Proceedings of ICSE ’18: 40th International Conference on Software Engineering*, Gothenburg, Sweden, May 27 - June 3, 2018 (ICSE ’18).

[5] https://github.com/Lawo/ember-plus/wiki

Stephan Grünfelder ist Autor des Buchs „Software-Test für Embedded Systems“ und Leiter des Testteams der Riedel Communications Austria GmbH. Vor dieser Tätigkeit, die ihm sehr viel Spaß

macht, hat er viele Jahre Test-Seminare für Industrieunternehmen abgehalten bzw. Unternehmen bei Testthemen beraten (siehe www.methver.webs.com). Seine Freizeit verbringt er am liebsten ganz ohne Computer mit seiner Familie auf einem kleinen Bauernhof im Waldviertel in Niederösterreich.

e-mail: **stephan.gruenfelder@riedel.net**

Christoph Luckeneder ist wissenschaftlicher Mitarbeiter am Institut für Computertechnik der TU Wien. Seine Interessen am Testen und Verifizieren von cyber-physikalischen Systemen wurde im Rahmen seines Masterstudiums geweckt. Seither befasst er sich in Forschungsprojekten mit der Verifikation von Automotive- und Robotik-Systemen. Den Fokus setzt er dabei auf die System-Modellierung zur anschließenden formalen Verifikation und effizienten Generierung von aussagekräftigen Testfällen

e-mail: **christoph.luckeneder@tuwien.at**