Optimaliseren van Continue Integratie door middel van Test Prioritering

Pieter De Clercq

Supervisor(s): Prof. dr. B. Volckaert, Prof. dr. ir. F. De Turck, J. Vaneessen, D. Kerkhove

Abstract— **Sinds de introductie van traditionele softwareontwikkelingsmodellen in de vorige eeuw is de omvang en de complexiteit van software almaar toegenomen. Deze ontplooiingen hebben softwareontwikkelaars ertoe aangezet om over te schakelen op zogenaamde Agile ontwikkelingsmethoden, waarin frequent samenvoegen van code en automatisch testen centraal staan. Op lange termijn zal de toenemende omvang van software echter een bijkomend negatief effect hebben op de grootte van het testpakket[1]. Om dit probleem van schaalbaarheid op te lossen, stelt deze masterproef een framework en een nieuw prioriteringsalgoritme voor. Hierbij worden tests gerangschikt volgens kans op falen. Het framework is geëvalueerd op twee bestaande applicaties en de resultaten zijn veelbelovend.**

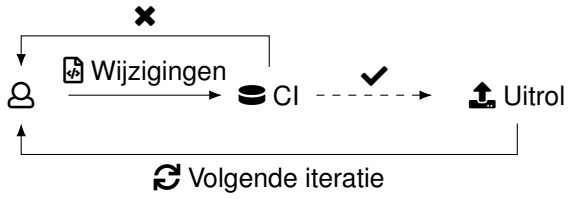
Trefwoorden— **Continue Integratie, test suite, performantie, optimalisatie, prioritering**

I. Introductie

Kenmerkend aan de hedendaagse wereld is de verbazingwekkende snelheid waarmee alles in deze wereld verandert. Dit geldt in het bijzonder voor de informaticasector, waarin elke dag nieuwe ontwikkelingen plaatsvinden. Hoewel deze ontwikkelingen vooral hardware gerelateerd ogen, denk maar aan smartwatches, zelfrijdende auto's en biologische technologie, kunnen ze niet functioneren zonder een nog meer geavanceerde softwarecomponent. Bijgevolg is zowel de omvang als de complexiteit van software de laatste decennia exponentieel toegenomen.

Softwareontwikkelaars hebben ondervonden dat de traditionele ontwikkelingsmethoden deze evolutie niet kunnen bijbenen en hebben hun focus verlegd naar andere strategieën. In plaats van de volledige applicatie in één keer te implementeren, verkiezen ontwikkelaars vandaag de dag de Agile methoden [1]. Bij deze methoden is het hoofddoel om zo snel mogelijk een minimale versie van de applicatie op de markt te brengen, om de financiële risico's te verkleinen. Achteraf kan extra functionaliteit stapsgewijs worden toegevoegd. Een rapport van The Standish Group bevestigt dat de slaagkans aanzienlijk groter is bij het hanteren van een Agile ontwikkelingsmethode [2].

Deze evolutie draagt echter ook negatieve gevolgen met zich mee. Het is één zaak om een project succesvol af te leveren, maar daarmee is de betrouwbaarheid nog niet gegarandeerd. Complexere software verhoogt onvermijdelijk de vatbaarheid voor fouten. De Agile aanpak tracht dit probleem op te lossen door middel van Continue Integratie (CI) [3], [4]. Dit idee vereist dat het volledige testpakket (succesvol) wordt uitgevoerd na elke aanpassing aan de code (Figuur 1). Vandaag de dag bestaan verschillende CI-services die dit process versoepelen door middel van automatisatie. Dit automatisatieproces kan optioneel worden aangevuld met het automatisch uitrollen van nieuwe versies naar de eindgebruikers (Continuous Deployment [5]).



Figuur 1: Continue Integratie.

Desalniettemin zal er zich op lange termijn nog een ander probleem voordoen. Aangezien de omvang van software exponentieel toeneemt en elke aanpassing in de code minstens één nieuwe test vereist, zal het aantal tests nog sneller stijgen. Dit leidt tot een schaalbaarheidsprobleem. Deze masterproef tracht dit probleem op te lossen door het testpakket te optimaliseren en de schaalbaarheid te verhogen.

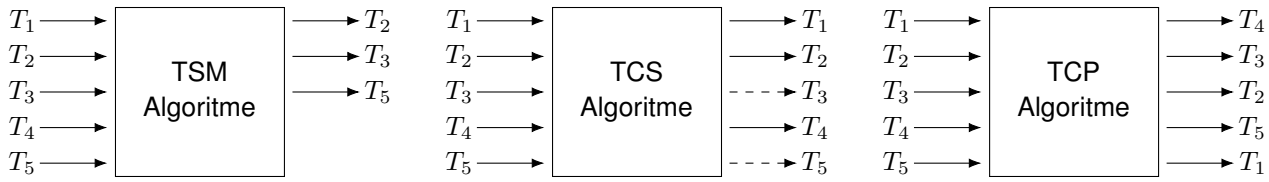
II. Technieken

Deze masterproef presenteert drie technieken om dit schaalbaarheidsprobleem op te lossen.

A. Testpakket Minimalisering (TSM)

De eerste techniek is Testpakket Minimalisering (TSM) [6]. TSM probeert de grootte van het testpakket te verkleinen door redundante tests permanent te verwijderen, volgens volgende definitie:

[1] De verzameling van alle tests in de applicatie.



(a) Testpakket Minimalisering (b) Test Selectie (c) Test Prioritering

Figuur 2: Overzicht van de technieken.

Gegeven:
- $T = \{t_1, \ldots, t_n\}$ een testpakket bestaande uit tests t_j.
- $R = \{r_1, \ldots, r_m\}$ een verzameling vereisten die voldaan moeten zijn om te stellen dat een applicatie grondig getest is.
- $\{T_1, \ldots, T_m\}$ deelverzamelingen van tests uit T. Voor $i \in [1..m]$ wordt elke deelverzameling T_i wordt geassocieerd met een vereiste r_i, zodanig dat eender welke test $t_j \in T_i$ kan worden uitgevoerd om te voldoen aan vereiste r_i.

Het probleem van TSM is vervolgens gedefinieerd als het vinden van een minimale deelverzameling T' van tests $t_j \in T$, zodanig dat aan elke vereiste $r_i \in R$ voldaan is.

B. Test Selectie (TCS)

In plaats van tests permanent te verwijderen, is het ook mogelijk om de veranderingen aan de code te analyseren om zo te bepalen welke tests zeker uitgevoerd moeten worden. Analoog kunnen andere tests mogelijk worden uitgesloten, omdat ze (waarschijnlijk) niet zullen falen [6].

Gegeven:
- T het testpakket.
- P de vorige versie van de code.
- P' the huidige (aangepaste) versie van de code.

TCS vindt een deelverzameling $T' \subseteq T$ die gebruikt kan worden om P' adequaat te testen.

C. Test Prioritering (TCP)

TSM en TCS voeren zo weinig mogelijk tests uit om de omvang van het testpakket te verkleinen. Soms kan het echter gewenst zijn om toch elke test uit te voeren, bijvoorbeeld bij kritische software voor medische doeleinden. In dit geval kan het testpakket nog steeds geoptimaliseerd worden, door de uitvoeringsvolgorde aan te passen. Test Prioritering (TCP) [6] rangschikt de tests zodanig dat een vooropgesteld doel zo snel mogelijk bereikt wordt. In deze masterproef zal het doel steeds zijn om zo snel mogelijk een falende test te detecteren.

Gegeven:
- T het testpakket.
- PT de verzameling van alle permutaties van T.
- $f : PT \mapsto \mathbb{R}$ een functie die gebruikt wordt om permutaties met elkaar te vergelijken.

TCP bepaalt de optimale permutatie $T' \in PT$ zodanig dat $\forall T'' \in PT : f(T') \geq f(T'') \Rightarrow (T'' \neq T')$.

III. Algoritmen

Deze masterproef focust op de TCP techniek, aangezien deze techniek elke test uitvoert en daardoor geen risico loopt om falende tests over te slaan. Om de optimale uitvoeringsvolgorde te bepalen worden drie algoritmen voorgesteld.

De algoritmen kunnen gebruik maken van drie gegevensreeksen:

- **Getroffen tests:** Door gebruik te maken van eerdere codebedekkingsresultaten[2] en de lijst met aanpassingen aan de code kan het framework bepalen welke tests mogelijks getroffen zijn door deze aanpassingen. Bijgevolg kan aan deze tests een hogere prioriteit toegewezen worden.
- **Historische uitvoeringsdata:** Vervolgens kan een algoritme gebruikmaken van historische uitvoeringsdata. Stel dat een test tijdens de vorige uitvoering gefaald is, dan bestaat de kans dat deze nu opnieuw zal falen. Bijgevolg moet deze test een hoge prioriteit krijgen om te controleren of het onderliggend probleem inmiddels is opgelost.
- **Uitvoeringstijden:** Ten slotte kan de gemiddelde uitvoeringstijd van een test gebruikt worden. Wanneer twee tests evenveel kans maken om te falen, moet de test met de laagste uitvoeringstijd de voorkeur genieten om sneller een potentieel falende test te detecteren.

A. Gretig algoritme

Het eerste algoritme is een gretige heuristiek die oorspronkelijk ontworpen is om het *set-covering probleem* op te lossen [7]. De heuristiek start met een de lege

[2]de *code coverage* geeft aan welke instructies in de code worden uitgevoerd bij het uitvoeren van een test

verzameling tests en de verzameling C van alle codelijnen. Vervolgens selecteert het algoritme iteratief de test die het meeste codelijnen in C bedekt. Na elke geselecteerde test worden de bedekte lijnen verwijderd uit C. Het algoritme herhaalt deze stappen totdat elke test geselecteerd is, of tot dat C leeg is. Door de selectievolgorde bij te houden en die te gebruiken als uitvoeringsvolgorde, kan dit algoritme worden gebruikt voor TCP.

B. HGS-algoritme

Het tweede algoritme is bedacht door Harrold, Gupta en Soffa [8]. In tegenstelling tot het gretige algoritme, gebruikt dit algoritme een andere invalshoek. Het algoritme begint door de lijst met codelijnen C te sorteren volgens stijgend aantal tests dat een codelijn bedekt. De reden voor deze sortering is dat sommige tests hoe dan ook moeten uitgevoerd worden, aangezien ze mogelijks de enige tests zijn die een bepaalde codelijn bedekken. Deze tests kunnen echter ook codelijnen bedekken die door andere tests worden bedekt, waardoor deze overbodig worden. Het algoritme overloopt alle codelijnen in deze volgorde en selecteert steeds een van de corresponderende tests. Na elke geselecteerde test wordt elke lijn die bedekt is door de geselecteerde test verwijderd uit C. Dit process herhaalt zich tot C leeg is en bijgevolg elke codelijn bedekt is.

C. ROCKET-algoritme

Ten slotte beschouwt deze masterproef het ROCKET-algoritme [9]. Dit algoritme rangschikt tests door aan elke test een score toe te kennen, op basis van historische uitvoeringsgegevens. Daarna wordt de cumulatieve score CS_t voor elke test t berekent en wordt de objectieffunctie g gedefinieerd. In deze functie stelt E_t de uitvoeringstijd van de test voor:

$$g = (maximaliseer(CS_t), minimaliseer(E_t))$$

Vervolgens optimaliseert het algoritme deze functie om de ideale uitvoeringsvolgorde S te bepalen, als volgt:

$$(\forall i \in 1 \ldots n)(g(S_i) \geq g(S_{i+1})$$

IV. Framework

Deze masterproef stelt het VeloCIty framework voor. Dit programmertaal-onafhankelijk framework laat toe om Test Prioritering te gebruiken op bestaande softwareprojecten. De architectuur bestaat uit drie hoofdcomponenten, een meta predictor en Alpha als een nieuw prioriteringsalgoritme.

A. Agent

De eerste component is de agent. Deze component wordt geïntegreerd in het testframework van de applicatie en is verantwoordelijk voor het uitvoeren van de tests in de optimale volgorde. Deze volgorde wordt opgevraagd aan de volgende component, de controller.

B. Controller

De controller is een server die twee taken uitvoert. De eerste taak is het afhandelen van verzoeken door de agent en deze door te sturen naar de *predictor*. Daarnaast ontvangt de controller ook feedback van de agent na elk uitgevoerd testpakket. Deze informatie wordt gebruikt om de meta predictor bij te werken, hierover later meer.

C. Predictor

Het laatste onderdeel van de architectuur is de predictor. Deze component inspecteert de aanpassingen aan de code om zo de optimale uitvoeringsvolgorde te bepalen. Deze volgorde wordt bepaald aan de hand van tien ingebouwde voorspellingsalgoritmen. Deze algoritmen zijn varianten van de drie eerder besproken algoritmen, aangevuld met het Alpha-algoritme (zie verder).

D. Meta predictor

Aangezien de predictor tien algoritmen bevat die elk een andere uitvoeringsvolgorde zullen voorspellen, is er nood aan een extra onderdeel dat de uiteindelijke volgorde bepaalt. De meta predictor is een tabel die een score toekent aan elk voorspellingsalgoritme. De volgorde van het algoritme met de hoogste score wordt gekozen als finale uitvoeringsvolgorde. Nadat de tests zijn uitgevoerd evalueert de controller de volgorde van de andere algoritmen en worden de scores bijgewerkt.

E. Alpha algoritme

Naast de Gretig, HGS- en ROCKET-algoritmen bevat dit framework ook een eigen algoritme. Dit algoritme start met het analyseren van de tests om de verzameling ATS van getroffen tests te bepalen. Binnen ATS, beschouwt het algoritme elke test die ten minste één keer gefaald is in de laatste drie uitvoeringen. Deze tests worden geselecteerd in stijgende uitvoeringstijd. Daarna worden de overblijvende tests in ATS geselecteerd, eveneens volgens stijgende uitvoeringstijd. Na deze twee stappen worden de resterende tests geselecteerd volgens het eerder besproken gretig algoritme.

V. Evaluatie

Het framework is geïnstalleerd in twee bestaande applicaties. Het eerste project is het Dodona project [3] van de Universiteit Gent. Dit project is gebruikt om de performantie van de voorspellingsalgoritmen te evalueren. Aangezien de agent in deze masterproef enkel compatibel is met Java applicaties en Dodona gebouwd is in Ruby-on-Rails, is een tweede testapplicatie gebruikt om de installatie te verifiëren. Hiervoor werd het Stratego[4] project gebruikt. Aansluitend beantwoordt deze masterproef drie onderzoeksvragen, met als doel waardevolle inzichten te vergaren over testpakketten.

A. Performantie

Tabel 1 vergelijkt de vier besproken algoritmen op twee vlakken. Deze vlakken zijn respectievelijk het aantal uitgevoerde tests tot de eerste gefaalde test en de bijbehorende uitvoeringstijd van het partiële testpakket. Deze resultaten tonen aan dat het Alpha-algoritme bijna 30 keer minder tests uitvoert en dat de eerste falende test bijna onmiddellijk gedetecteerd wordt. Het Gretig en HGS-algoritme halveren het aantal uitgevoerde tests en zorgen tegelijk voor een sterke reductie van de uitvoeringstijd. De performantie van het ROCKET-algoritme is opmerkelijk. Dit algoritme voert veel meer tests uit dan de mediaan van de originele volgorde, maar detecteert een falende test vier keer sneller.

Algoritme	Mediaan (tests)	Mediaan (tijd)
Origineel	78	123 s
Alpha	3	1 s
Greedy	33	12 s
HGS	10	6 s
ROCKET	170	32

Tabel 1: Performantie op het Dodona project.

B. Onderzoeksvragen

Deze masterproef beantwoordt drie onderzoeksvragen door gebruik te maken van gegevens van Travis CI. Deze gegevens zijn verzameld en gepubliceerd door het TravisTorrent project [10] (3 702 595 uitvoeringen) en door Durieux et al. [11] (35 793 144 uitvoeringen).

OV1: Kans op falen. De eerste onderzoeksvraag beschouwt de kans dat de uitvoering van een testpakket zal falen. Volgens beide bronnen is deze kans 15 %.

OV2: Gemiddelde uitvoeringstijd. Na het inspecteren van de TravisTorrent gegevens werd duidelijk dat de tijdsinformatie onvolledig en inaccuraat was, waardoor deze gegevens niet konden gebruikt worden om een betrouwbaar antwoord op deze vraag te bieden. De andere bron geeft aan dat Travis CI hoofdzakelijk gebruikt wordt voor kleinere projecten, met een gemiddelde uitvoeringstijd van 385 s per testpakket en een maximum van 26 h.

OV3: Opeenvolgend falen. De laatste onderzoeksvraag betreft de kans dat de tests van een project meer dan twee keer na elkaar falen. Hiervoor kan enkel de TravisTorrent bron worden gebruikt aangezien deze bron een koppeling bevat tussen alle uitvoeringen van hetzelfde project. Volgens deze bron is de kans op opeenvolgend falen 51.76 %.

VI. Conclusie en aanvullend werk

In deze masterproef is aangetoond dat het testpakket van softwareprojecten geoptimaliseerd kan worden, in het bijzonder met behulp van Test Prioritering. Het voorgestelde framework is succesvol geïmplementeerd in twee projecten en de resultaten zijn veelbelovend. Er zijn echter nog een aantal limitaties, alsook ruimte voor verbetering.

Agent. Voor de implementatie van de agent in deze masterproef werd gekozen voor Java, meer specifiek het JUnit testframework. Dit heeft als nadeel dat tests niet parallel kunnen worden uitgevoerd. Om dit mogelijk te maken is er, naast de technische moeilijkheden, nood aan een coördinatiemechanisme om tests over meerdere processorthreads te plannen.

Predictor. Momenteel werken alle voorspellingsalgoritmes los van elkaar. Mogelijks kan er verborgen potentieel schuilen in het combineren van verschillende algoritmen, bijvoorbeeld door gewichten toe te kennen aan elk algoritme en op basis daarvan de voorspellingen samen te voegen.

Meta predictor. De eenvoudige meta predictor die in deze masterproef wordt gebruikt kan aangepast worden naar bijvoorbeeld een saturerende teller, of naar een Machine Learning algoritme. Analoog kan Machine Learning eventueel worden gebruikt als voorspellingsalgoritme.

[3] https://dodona.ugent.be/

[4] Java Spring applicatie ontwikkeld voor het vak Software Engineering 2.

Referenties

[1] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas, "Manifesto for agile software development," 2001.

[2] Standish Group et al., "Chaos report 2015," *The Standish Group International*, 2015.

[3] John Ferguson Smart, *Jenkins: The Definitive Guide*, O'Reilly, Beijing, 2011.

[4] Glenford J. Myers, Corey Sandler, and Tom Badgett, *The Art of Software Testing*, Wiley Publishing, 3rd edition, 2011.

[5] Mohammed Arefeen and Michael Schiller, "Continuous integration using gitlab," *Undergraduate Research in Natural and Clinical Science and Technology (URNCST) Journal*, vol. 3, pp. 1–6, 09 2019.

[6] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," *Softw. Test. Verif. Reliab.*, vol. 22, no. 2, pp. 67–120, Mar. 2012.

[7] Raphael Noemmer and Roman Haas, *An Evaluation of Test Suite Minimization Techniques*, pp. 51–66, 12 2019.

[8] M. Jean Harrold, Rajiv Gupta, and Mary Lou Soffa, "A methodology for controlling the size of a test suite," *ACM Trans. Softw. Eng. Methodol.*, vol. 2, no. 3, pp. 270–285, July 1993.

[9] D. Marijan, A. Gotlieb, and S. Sen, "Test case prioritization for continuous regression testing: An industrial case study," in *2013 IEEE International Conference on Software Maintenance*, 2013, pp. 540–543.

[10] Moritz Beller, Georgios Gousios, and Andy Zaidman, "Travistorrent: Synthesizing travis ci and github for full-stack research on continuous integration," in *Proceedings of the 14th working conference on mining software repositories*, 2017.

[11] Thomas Durieux, Rui Abreu, Martin Monperrus, Tegawendé F. Bissyandé, and Luís Cruz, "An analysis of 35+ million jobs of travis ci," 2019.