Universität Augsburg

Fakultät für Angewandte Informatik

Masterarbeit

im Studiengang Informatik

zur Erlangung des akademischen Grades
Master of Science

Thema:	<Thema der Arbeit>
Autor:	Gerald Siegert MatNr. 1450117
Version vom:	21. Dezember 2017
1. Betreuerin:	Prof. Dr. X
2. Betreuer:	Prof. Dr. Y

Eidesstattliche Erklärung

Ich versichere, die von mir vorgelegte Arbeit selbstständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

Ort, Datum: Unterschrift:

Zusammenfassung

Abstract

Inhaltsverzeichnis

Abbildungsverzeichnis

Listingverzeichnis

Kapitel 1

Einleitung

1.1 Testen von Software und Systemen

Softwaretests sind in der heutigen Zeit eine wichtige Grundlage im Bereich der Qualitätssicherung bei Softwareprojekten. Abhängig von Komplexität und Sicherheitsanforderungen werden meist zwischen 30 und 60 Prozent der Kosten einer Software für die Qualitätssicherung und somit das Testen der Software verwendet (6). Ohne Softwaretests hätte heutige Software zahlreiche Fehler. Um die Wichtigkeit von Sofwaretests zu unterstreichen sieht z. B. das V-Modell (vgl. Abbildung 1.1) für jeden Entwicklungsschritt am Ende eine entsprechende Testphase vor, woher auch der Name V-Modell stammt.

Nun ist es natürlich sehr aufwändig und daher mit einem vertretbaren Aufwand kaum machbar, jeden Test manuell durchzuführen. Daher wird versucht, möglichst viel zu automatisieren. Vor allem bei Unit-Tests ist dies mithilfe des *XUnit*-Framework sehr einfach möglich. Dabei werden zunächst einzelne Testfälle erstellt und können im Anschluss aufgrund der aktuellen Codebasis jederzeit ausgeführt werden. Automatisierte Tests können auch dazu genutzt werden, um einen einzelnen Test mit verschiedenen Eingaben durchzuführen. Dadurch können verschiedene Eingabeklassen (wie negative oder positive Ganzzahlen) mit sehr geringem Aufwand in einem Test genutzt werden und somit verschiedene Testfälle direkt ausgeführt werden. Durch diese Testautomatisierung können somit zahlreiche Kosten eingespart werden.

Weitere Testframeworks gibt es auch zum Testen von sicherheitskritischen Systemen, welche nicht immer reine Software sein müssen. Dies ist einer der wichtigsten Anwendungsfälle für sogenannte *Model Checker* (MC). *Model Checking* (MC) wird dazu verwendet, um ein Modell auf seine Spezifikation zu testen. Dazu werden abhängig von den aktivierten Komponentenfehlern verschiedene Zustände des Modells vollautomatisch ausgeführt und anhand der dadurch im System auftretenden Fehler entschieden, ob die Spezifikation erfüllt wird (3, 4).

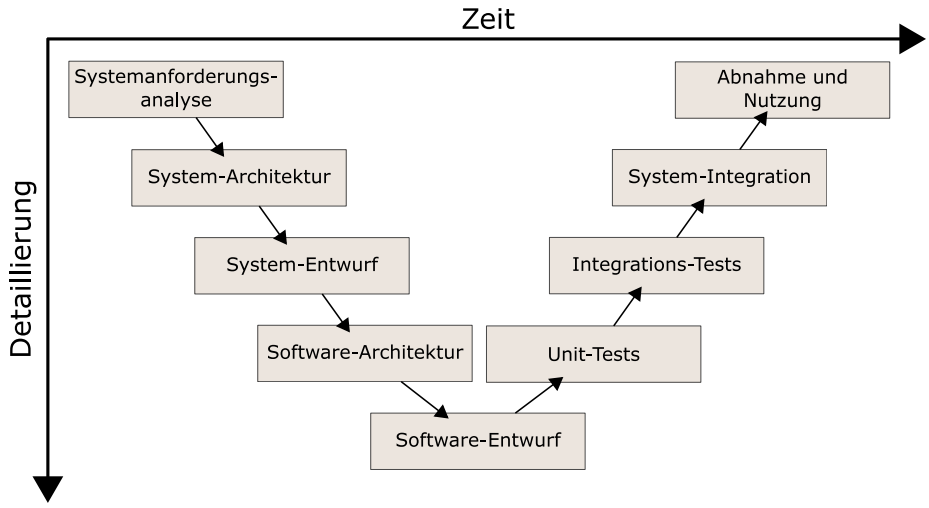

Abbildung 1.1: Aufbau des V-Modells (7)

1.2 Problemstellung

Am Softwaretchnik-Lehrstuhl der Universität Augsburg wurde in den letzten Jahren das Framework S# (sprich *Safety Sharp*) entwickelt. Dieses Framework kann dazu genutzt werden, selbstorganisierte und sicherheitskritische Systeme zu analysieren und ihre Schwächen zu ermitteln (4, 5). Dazu wurde auch bereits die ZNN.com-Fallstudie, welche Shang-Wen Cheng in seiner Dissertation beschrieben hat (1), als vereinfachtes Modell mit den wesentlichen Bestandteilen implementiert. Nun soll nicht mehr nur dieses Modell getestet werden, sondern ein reales System mit einem ähnlichen Aufbau. Dafür wurde Apache Hadoop[1] ausgewählt, welches in der Industrie im Bereich Datenverarbeitung und dem Stichwort *Big Data* eingesetzt wird. Mit Hadoop ist es möglich, ein Servercluster zu erstellen, auf dem anschließend dafür entwickelte Anwendungen ausgeführt werden. Es soll daher nun getestet und analysiert werden, wie sich Hadoop unter verschiedenen Lastprofilen verhält und dabei bestimmte Fehler auftreten, wie z. B. wenn die Verbindung zu einem der Server verloren geht oder ein Server komplett ausfällt.

Bei der ZNN.com-Fallstudie als reines Modell gab es bereits eine ähnliche Aufgabenstellung, welche im Positionspapier (2) genauer erläutert wurde. Das Hauptziel dieses Projektes ist daher nun, anstatt eines reinen Modells ein reales System zu testen. Dafür wird ein modellbasierter Ansatz als Testkonzept genutzt und Hadoop zunächst als Modell nachgebildet. Dieses Modell wird dann dazu genutzt, um ein reales Hadoop-Cluster entsprechend zu steuern und zu testen, wie sich das reale System unter bestimmten Bedingungen verhält und dabei zu ermitteln, wann es nicht mehr funktionsfähig ist.

[1] `https://hadoop.apache.org/`

1.2.1 Aufbau des Modells

Zunächst muss natürlich erst einmal der Versuchsaufbau selbst in S# modelliert werden. Ein Modell beinhaltet in S# zunächst einmal die Komponenten des Systems und deren Zusammenhänge, also wie die Komponenten miteinander agieren. Wichtig sind in einem S# -Modell aber auch mögliche Komponentenfehler, welche bekannt sind und jederzeit auftreten können. Komponentenfehler werden bereits in der Designphase eines Modells eingearbeitet und können bei der späteren Ausführung flexibel aktiviert und deaktiviert werden, um die Probleme des zu testenden Systems zu ermitteln (vgl. **??**).

Um nun Hadoop in S# zu modellieren wird zunächst ein Konzept erstellt, in dem ausgearbeitet wird, welche Komponenten und Komponentenfehler relevant sind. Anschließend müssen deren Zusammenhänge und wesentlichen Eigenschaften ausgearbeitet werden und in das Konzept übernommen werden. Sobald das Konzept fertig ausgearbeitet ist, kann das Modell in S# implementiert werden.

1.2.2 Mapping zum realen System

Nachdem die Basis des Modells steht, kann die Funktionalität entwickelt werden. Dazu werden in S# nur Basisfunktionen eingebaut, um mit dem realen System kommunizieren zu können. Dies geschieht mit einer Art Treiber, welcher mithilfe von mehreren SSH-Verbindungen mit dem realen Hadoop-Cluster kommuniziert und so das Mapping zwischen Modell und realem System übernimmt. Jede der SSH-Verbindungen ist dabei nur für einen Einsatzzweck gedacht, sodass es Verbindungen für u. A. folgende Einsatzzwecke gibt:

- Starten von Benchmark-Anwendungen
- Monitoring des realen Cluster
- Injektion von Komponentenfehler

Der Vorteil von mehreren Verbindungen liegt darin, dass jede Verbindung unabhängig ist und nicht auf die Antwort des zuvor gestarteten Programms warten muss. So ist es möglich, mithilfe mehrere Verbindungen mehrere Anwendungen parallel zu starten und jede Rückgabe auszuwerten und währenddessen verschiedene Komponentenfehler zu aktivieren.

1.2.3 Erstellung der Lastprofile

Sobald das Grundmodell steht, können die Testfälle selbst entwickelt werden. Als Testfälle dienen dazu unterschiedliche Lastprofile, um verschiedene Auslastungsgrade und Nutzungsszenarien zu simulieren. Dazu sollen die Lastprofile verschiedene Benchmarks beinhalten, deren einzelne Anwendungen kombiniert oder alleine auf dem realen System ausgeführt werden:

- Hadoop Mapreduce Examples
- Intel HiBench
- SWIM (Statistical Workload Injector for Mapreduce)

Eine Besonderheit bildet der SWIM-Benchmark, welcher sehr Ressourcenintensiv ist und daher auf einem *Single Node Cluster* sehr Zeitintensiv sein kann. Der Intel HiBench basiert auf einzelnen Bestandteilen der Mapreduce Examples. Die Examples wiederum sind nicht mehr als zahlreiche voneinander unabhängige Beispielanwendungen für Hadoop, wodurch die Möglichkeit besteht, sehr einfach unterschiedliche Profile zu simulieren, je nachdem welche der Einzelanwendungen alleine oder parallel gestartet werden. Daher muss zunächst auch geprüft werden, welcher Benchmark welche Möglichkeiten bietet, um die benötigten Testfälle bzw. Lastprofile zu erstellen und so den dynamischen Teil des zu testenden Modells zu erstellen.

1.2.4 Erstellen und Ausführen der Tests

Sobald Modell und Testfälle stehen, kann mit der Erstellung der Tests fortgefahren werden. Die Tests müssen nun so erstellt werden, dass sie sich einerseits auf veränderte Bedingungen des realen Clusters anpassen, aber auch automatisiert die einzelnen Anwendungen der Lastprofile aktivieren und ausführen. Dies schließt auch unterschiedliche Profile für die Aktivierung der Komponentenfehler ein. Zum einen kann nur eine Simulation ohne Fehler gestartet werden, zum anderen aber auch unterschiedliche Komponentenfehler aktiviert werden. Der *Model Checker* von S# besitzt dazu auch Möglichkeiten, um Komponentenfehler auch kombiniert auszuführen. Dazu werden basierend auf zuvor definierte *Constraints* Komponentenfehler aktiviert, um so typische Probleme des realen Systems zu simulieren. Basierend darauf wird dann ermittelt, welche Fehler nun im realen System auftauchen.

1.2.5 Evaluierung der Ergebnisse

Je nachdem welche Constraints bei der Ausführung genutzt werden, sind nun unterschiedliche Fehler und Daten im realen System ermittelt worden, welche zum Abschluss evaluiert werden müssen. Einige Erwartungen sind da natürlich bereits im Voraus klar: Sollte es zu einem Netzwerk- oder Serverausfall eines Hadoop-Nodes kommen, muss das System dies selbstständig erkennen und die Anwendung an einen anderen Node abgeben. Dabei sollte das System auch erkennen, welche anderen Nodes bereits beschäftigt sind und entsprechend auf dem von Hadoop genutzten *Load Balancer* einen Node auswählen. Neben einer Fehleranalyse können aber auch die Laufzeiten unter bestimmten Bedingungen analysiert werden.

Literaturverzeichnis

[1] Cheng, S.-W. : *Rainbow: Cost-effective Software Architecture-based Self-adaptation*. Pittsburgh, PA, USA, Carnegie Mellon University, Diss., 2008. – AAI3305807

[2] Eberhardinger, B. ; Habermaier, A. ; Reif, W. : Toward Adaptive, Self-Aware Test Automation. In: *2017 IEEE/ACM 12th International Workshop on Automation of Software Testing (AST)*, 2017, S. 34–37

[3] Grumberg, O. ; Clarke, E. ; Peled, D. : Model checking. (1999)

[4] Habermaier, A. ; Eberhardinger, B. ; Seebach, H. ; Leupolz, J. ; Reif, W. : Runtime Model-Based Safety Analysis of Self-Organizing Systems with S#. In: *2015 IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops*, 2015, S. 128–133

[5] Habermaier, A. ; Leupolz, J. ; Reif, W. : Unified Simulation, Visualization, and Formal Analysis of Safety-Critical Systems with S#. In: Beek, M. H. (Hrsg.) ; Gnesi, S. (Hrsg.) ; Knapp, A. (Hrsg.): *Critical Systems: Formal Methods and Automated Verification: Joint 21st International Workshop on Formal Methods for Industrial Critical Systems and 16th International Workshop on Automated Verification of Critical Systems, FMICS-AVoCS 2016, Pisa, Italy, September 26-28, 2016, Proceedings*. Springer International Publishing. – ISBN 978–3–319–45943–1, 150–167

[6] Polo, M. ; Reales, P. ; Piattini, M. ; Ebert, C. : Test Automation. In: *IEEE Software* 30 (2013), Jan, Nr. 1, S. 84–89. `http://dx.doi.org/10.1109/MS.2013.15`. – DOI 10.1109/MS.2013.15. – ISSN 0740–7459

[7] Pätzold, M. ; Seyfert, S. : *V-Modell*. `https://commons.wikimedia.org/wiki/File:V-Modell.svg`. Version: Januar 2010. – Lizenz: CC-BY-SA 3.0