Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit Nr. 111

Generierung und Optimierung von Testzeitplänen im Rahmen des SOA Change Managements

David Krauss

Studiengang:	Softwaretechnik
Prüfer:	Prof. Dr.-Ing. habil. Bernhard Mitschang
Betreuer:	Dipl.-Inf. Jan Königsberger
Beginn am:	23. Mai 2016
Beendet am:	22. November 2016
CR-Nummer:	D.2.5, H.4.1, H.5.3

Kurzfassung

Tester einer dienstorientieren Architektur stehen, anders als beim traditionellen Software-Testing, enormen Herausforderungen gegenüber. Heterogene Systemlandschaften, über Unternehmensgrenzen hinweg verteilte Akteure und die dynamische Natur einer solchen Architektur erfordern neue Ansätze beim Testing. Die Fehler-Ursachen-Analyse wird bei zusammengesetzten Services zum großen Problem, da bei der Integration von vielen Komponenten unklar ist, wo die Ursache eines Fehlers zu suchen ist. Das entwickelte Konzept nutzt im Rahmen der SOA Governance Abhängigkeiten aus, um einen geordneten Testzeitplan zu generieren. Seine Ausführung stellt sicher, dass während einer Testperiode nur eine ungetestete Service-Version beteiligt ist, was die möglichen Fehlerursachen stark einschränkt. Darüber hinaus wird die Nebenläufigkeit bei der Testausführung gefördert, indem die zuständigen Tester parallel an unterschiedlichen Testperioden arbeiten. Ein Prototyp des Konzepts wird als Teil eines SOA Governance Repositories implementiert. Er implementiert die Verwaltung von Releases, die Durchführung der Testzeitplan-Generierung und eine Testzeitplan-Visualisierung. Der Generierungsprozess selbst nutzt eine topologische Sortierung des umgekehrten Abhängigkeitsgraphen, um die Testperioden zu erstellen. Vier implementierte Optimierungen können den generierten Testzeitplan gezielt verbessern, um beispielsweise eine kurze Gesamt-Testdauer zu erzielen. Unter Verwendung der SOA-Daten eines großen Automobilherstellers wird die Implementierung unter realitätsnahen Bedingungen ausgeführt. Dabei zeigt die Fallstudie, dass das automatisierte Verfahren performant arbeitet und einen praxistauglichen Testzeitplan generiert.

Inhaltsverzeichnis

1 Einleitung **9**
- 1.1 Prolog . . . 9
- 1.2 Aufgabenstellung . . . 9
- 1.3 Gliederung . . . 10

2 Grundlagen **11**
- 2.1 Grundlagen der Service Oriented Architecture (SOA) . . . 11
- 2.2 Das Zeitplan-Problem . . . 11
- 2.3 Testing im Kontext der SOA . . . 12
  - 2.3.1 Besondere Herausforderungen . . . 12
  - 2.3.2 Herangehensweisen . . . 16
  - 2.3.3 Regressionstests . . . 16
  - 2.3.4 Automatisierte Generierung von Testfällen . . . 17
  - 2.3.5 Ermittlung von Abhängigkeiten . . . 17
- 2.4 SOA Governance . . . 17
- 2.5 SOA Governance Repository . . . 18
  - 2.5.1 SOA Governance Meta Model (SOA-GovMM) . . . 18
  - 2.5.2 Funktionen . . . 20
  - 2.5.3 Prototyp . . . 21

3 Konzept der Testzeitplan-Generierung **23**
- 3.1 Anforderungen . . . 23
  - 3.1.1 Verbundreleases . . . 23
  - 3.1.2 Testaufwand . . . 24
  - 3.1.3 Abhängigkeiten . . . 24
- 3.2 Generierung von Testzeitplänen . . . 25
  - 3.2.1 Erstellung des Abhängigkeitsgraphen . . . 26
  - 3.2.2 Topologische Sortierung . . . 29
  - 3.2.3 Generierung von Testperioden . . . 31
- 3.3 Darstellung eines Testzeitplans . . . 32
  - 3.3.1 Grafische Darstellung eines Testzeitplans . . . 32
  - 3.3.2 Vergleich mehrerer Testzeitpläne . . . 33
  - 3.3.3 Vergleich der maßgeblichen Kenngrößen . . . 34

4 Konzept der Optimierungen **37**
4.1 Testperioden expandieren . . . 37
4.2 Früher Teststart . . . 38
4.3 Spätes Testende . . . 39
4.4 Teilen des kritischen Pfads . . . 39
4.4.1 Kriterium zur Anwendung . . . 40
4.4.2 Schnittpunkt ermitteln . . . 41
4.4.3 Schnittpunkt testen . . . 41
4.4.4 Terminierung der Optimierung . . . 42

5 Implementierung der vorgestellten Konzepte **43**
5.1 Architektur des SGR . . . 43
5.1.1 Erweiterungen . . . 44
5.1.2 Datenmodell . . . 45
5.2 Optimierungsgerüst . . . 46
5.3 Oberfläche des Prototypen . . . 46
5.3.1 Verbundreleases . . . 47
5.3.2 Testzeitpläne . . . 49
5.4 Optimierungszeitpunkt . . . 51
5.5 Implementierung der Optimierungen . . . 52
5.5.1 Testperioden expandieren . . . 53
5.5.2 Früher Teststart . . . 54
5.5.3 Spätes Testende . . . 55
5.5.4 Teilen des kritischen Pfads . . . 55
5.6 Komplexität der Optimierungen . . . 59

6 Fallstudie **61**
6.1 Verwendung eines realen Datensatzes . . . 61
6.1.1 Gewähltes Verbundrelease . . . 62
6.2 Generierung des Testzeitplans . . . 63
6.2.1 Angewandte Optimierungen . . . 63
6.2.2 Performance-Messungen . . . 64
6.3 Auswertung und Bewertung des generierten Testzeitplans . . . 65

7 Zusammenfassung und Ausblick **67**
7.1 Zusammenfassung . . . 67
7.2 Erweiterungsmöglichkeiten . . . 68
7.2.1 Generische Testzeitplan-Generierung als eigenständiger Service . . . 68
7.2.2 Standardwert für die Testleistung eliminieren . . . 68
7.2.3 Interaktive Darstellung des Abhängigkeitsgraphen . . . 68
7.2.4 Weitere Optimierungsmöglichkeiten . . . 69
7.3 Ausblick . . . 69
7.3.1 Testzeitplan-Generierung in anderen Domänen . . . 69

7.3.2 Reaktiver dynamischer Zeitplan 70
7.4 Fazit 70

Literaturverzeichnis **73**

Abbildungsverzeichnis

2.1 SOA Governance Meta Model [KSM14] . . . 19
2.2 Dem SGR zu Grunde liegendes Datenmodell . . . 20
2.3 Dashboard des SGR . . . 21

3.1 Systeme nutzen Service-Versionen über Verträge . . . 27
3.2 Testreihenfolge von Systemen und Service-Versionen . . . 28
3.3 Voranschreitender Algorithmus im Beispielgraphen . . . 30
3.4 Prototypische Darstellung eines Testzeitplans . . . 33
3.5 Prototypische Darstellung des visuellen Testzeitplan-Vergleichs mit zwei Kandidaten . . . 34
3.6 Beispielhafte Gewichtung von Anforderungen und deren Verknüpfung mit messbaren Kenngrößen . . . 35

4.1 Expandieren der Testperioden . . . 38
4.2 Teilen des kritischen Pfads . . . 40

5.1 Komponenten der SGR-Architektur . . . 43
5.2 Model-Klassen der Implementierung . . . 46
5.3 Klassendiagramm der Optimierungen . . . 47
5.4 Dialog zum Hinzufügen eines Verbundreleases . . . 48
5.5 Auflistung von Service-Versionen, die Teil eines Verbundreleases sein sollen . 49
5.6 Dialog zum Generieren von Testzeitplänen . . . 49
5.7 Fehlermeldung beim Verpassen des Releasezeitpunktes . . . 50
5.8 Darstellung eines generierten Testzeitplans . . . 51
5.9 Vergleich zweier Testzeitpläne ohne und mit Optimierungsausführung . . . 59

6.1 Vereinfachte und anonymisierte Darstellung der Abhängigkeiten zwischen den beteiligten Systemen und Service-Versionen . . . 62
6.2 Generierter Testzeitplan unter Verwendung randomisierter Testdauern . . . 63

1 Einleitung

Im Folgenden wird sowohl in die Struktur des Dokuments als auch in das Thema, dessen Entstehung und seine Relevanz eingeleitet.

1.1 Prolog

Unternehmen, die flexible, gut strukturierte und transparente Geschäftsprozesse implementieren wollen, führen neuerdings immer häufiger eine dienstorientierte Architektur (englisch: Service Oriented Architecture, SOA) ein [Erl05]. Diese soll unter anderem zu mittel- und langfristigen Kosteneinsparungen führen und sowohl die Integration als auch das Outsourcing von Teilprozessen vereinfachen. Auch vor der Einführung einer SOA mussten Unternehmen eine Vielzahl von Softwaresystemen betreiben, die durch die Einführung einer SOA durch eine mindestens gleich große Anzahl an Services realisiert werden. Gleichzeitig existieren Services in unterschiedlichen Service-Versionen. Um diese umfangreiche IT-Struktur zielgerichtet einsetzen zu können, kommt das SOA-Governance-Konzept als Teilgebiet der IT Governance zum Einsatz. Dieses Konzept stellt beispielsweise Standards, Prozesse und Strukturen für die erfolgreiche Umsetzung von Software-Projekten in einem sehr großen Unternehmensumfeld bereit. Der zu Grunde liegende Anwendungsfall sieht mehrmals jährlich eine gesammelte Ausbringung neuer Service-Versionen vor. Schnell wird klar, dass dieser Releasezeitpunkt einen enormen Planungsaufwand unter Berücksichtigung unzähliger Stakeholder, individueller Projektzeitpläne und Abhängigkeiten der Services untereinander erfordert. Die ausführliche Planung der Tests hilft dabei, das Release gut vorzubereiten, um die Risiken so klein wie möglich zu halten.

1.2 Aufgabenstellung

Im Rahmen dieser Arbeit sollen die Abhängigkeiten zwischen Services untersucht werden. Dabei soll die Frage geklärt werden, welche Abhängigkeiten in einem großen SOA-Systemverbund existieren können und mit Hilfe welcher Informationen diese abgebildet werden können. Auf Basis der daraus entstehenden Erkenntnisse soll die automatisierte Generierung unter Berücksichtigung aller notwendigen Rahmenbedingungen konzipiert und anschließend im existierenden Prototypen eines SOA Governance Repositories umgesetzt werden. Mit Hilfe einer

geeigneten Visualisierung sollen Testzeitpläne in diesem Prototypen angezeigt werden können. In einem weiteren Schritt sollen Konzepte zur Optimierung dieser Zeitpläne untersucht und implementiert werden.

1.3 Gliederung

Die Arbeit ist wie folgt gegliedert:

Kapitel 2 – Grundlagen beschreibt die Grundlagen und Besonderheiten beim Testing einer SOA. Außerdem wird SOA Governance und die prototypische Implementierung des Konzepts im SOA Governance Repository behandelt.

Kapitel 3 – Konzept der Testzeitplan-Generierung stellt die automatisierte Testzeitplan-Generierung, -Visualisierung und -Optimierung auf der konzeptionellen Ebene vor.

Kapitel 4 – Konzept der Optimierungen dokumentiert die Funktionsweise und Einsatzmöglichkeiten der Optimierungen, die während der Testzeitplan-Generierung ausgeführt werden können.

Kapitel 5 – Implementierung der vorgestellten Konzepte behandelt die prototypische Implementierung im Rahmen des SOA Governance Repositories.

Kapitel 6 – Fallstudie zeigt die Anwendung des Algorithmus auf einen realen Datensatz auf. Dabei werden unterschiedliche Aspekte wie die Performance des Algorithmus und die Qualität seiner Ergebnisse untersucht.

Kapitel 7 – Zusammenfassung und Ausblick gibt einen reflektierenden Überblick über die Herausforderungen und Ergebnisse der Arbeit sowie einen Ausblick auf mögliche weiterführende Arbeitsgebiete.

2 Grundlagen

Dieses Kapitel ist eine Einführung in die Grundlagen der weiteren Abhandlung. Es werden Fragestellungen und bestehende Arbeiten zum Thema SOA Governance und Testing im Kontext einer SOA behandelt.

2.1 Grundlagen der Service Oriented Architecture (SOA)

Die dienstorientierte Architektur ist ein Architekturmuster, das mit Hilfe von verteilten IT-Systemen die Durchführung von Geschäftsprozessen anstrebt [Ros08]. Folglich sollen die beteiligten Komponenten gemeinsam einem übergeordneten Zweck dienen. Dabei wird ihre Struktur und Ausführungslogik von einer klar definierten und gut dokumentierten Schnittstelle verborgen. Durch die langfristige Anwendung des Architekturmusters soll die Wiederverwendbarkeit der Komponenten gesteigert werden [Erl05]. Kernkomponenten einer SOA sind Dienste (Services), die als eigenständige und abgeschlossene Komponenten Funktionalität bereitstellen. Services können in vielen verschiedenen Sprachen und auf unterschiedlichen Plattformen implementiert und bereitgestellt werden. Die nachrichtenbasierte Kommunikation mittels XML-Nachrichten führt zu einer losen Kopplung der Komponenten [Erl04]. Diese geringen Abhängigkeiten untereinander ermöglichen die Orchestrierung von Services zu zusammengesetzten Services. Dabei werden bestehende Implementierungen wiederverwendet, um einen neuen Geschäftsprozess zu realisieren [Cum09]. Auf diese Weise können Geschäftsprozesse über Unternehmensgrenzen hinweg integriert werden. Trotz der starken Fokussierung auf Services, dürfen alle anderen Aspekte einer SOA nicht in den Hintergrund rücken. Insbesondere die Einführung einer dienstorientierten Architektur birgt für Unternehmen zunächst enorme Risiken. Erst durch die langfristige Anwendung rechtfertigen sich die verhältnismäßig hohen Kosten zur Einführung des Architekturmusters.

2.2 Das Zeitplan-Problem

In Schulen und Universitäten müssen regelmäßig umfangreiche Zeitpläne zur Planung von Veranstaltungen erstellt werden. Unter anderem müssen dabei die Schulstunden, Kurse und Vorlesungen den vorhandenen Räumlichkeiten zugewiesen werden. Gleichzeitig muss das

Lehrpersonal eingeteilt werden, ohne dass dabei Überschneidungen für das Personal auftreten. Natürlich sollen auch Schüler und Studenten die für sie wichtigen Veranstaltungen besuchen können, ohne dass dabei Überschneidungen auftreten. Für diese komplexe Aufgabe der Zeitplan-Erstellung gibt es bereits mehrere Lösungsansätze. Die Problemstellung, einen Zeitplan zu finden, der eine Menge von Räumlichkeiten zu vorgegebenen Zeitintervallen zuweist, ohne dass ein Raum während eines Zeitintervalls mehrfach genutzt wird, ist NP-hart (nicht in Polynomialzeit lösbar). Dies wird in [BGW02] durch die Reduktion des Graph-Färbbarkeitsproblems auf das Zeitplan-Problem gezeigt. Ein Lösungsansatz besteht in der Ausführung eines genetischen Algorithmus. Dabei wird eine anfängliche Population von Zeitplan-Instanzen (mit Überschneidungen) so lange durch Mutationsoperatoren abgeändert, bis keine signifikanten Verbesserungen erzielt werden. Immer wieder unterscheiden sich die auftretenden Problemstellungen in diesem Bereich durch Kleinigkeiten, wie lokale Einschränkungen oder rechtliche Aspekte. Eine möglichst allgemeine Formulierung des Problems versuchen Gröbner et al. [GW02]. Im vorgestellten Modell werden mit Hilfe unterschiedlicher Sichten (zum Beispiel: Sicht der Lehrers, Sicht der Klasse oder Sicht der Raumbelegung) Ressourcen den Ereignissen zugewiesen.

Für die Testzeitplan-Generierung können bestehende Lösungen des Zeitplan-Problems adaptiert werden, um die überschneidungsfreie Zuweisung von Ressourcen zu ermöglichen.

2.3 Testing im Kontext der SOA

Wie bei anderen Architekturen spielt das Testing im Rahmen einer SOA eine wichtige Rolle. Tests stellen sicher, dass eine SOA die an sie gestellten funktionalen und nicht-funktionalen Anforderungen erfüllt. Allerdings müssen traditionelle Teststrategien für den Einsatz in einer SOA neu ausgelegt werden [KG12]. Dabei gibt es mehrere Besonderheiten (Abschnitt 2.3.1) und Einschränkungen beim Einsatz von Unit-, Integrations-, System- und Regressionstests. Service-Implementierungen können für gewöhnlich zuerst mit herkömmlichen Unit-Tests getestet werden. Integrations- und Systemtests sind bei Komponenten einer SOA besonders wichtig, stellen aber auf Grund der in Abschnitt 2.3.1 genannten Besonderheiten meist eine große Herausforderung dar. Das Konzept und der Prototyp aus Kapitel 3 und 5 helfen dabei, die Interaktionen zwischen Services zu testen. Verschiedene Möglichkeiten, Regressionstests auszuführen, werden in Abschnitt 2.3.3 vorgestellt.

2.3.1 Besondere Herausforderungen

Eine SOA besteht typischerweise aus einer Vielzahl von Komponenten, die zusammen eine Geschäftslogik implementieren. Alle Komponenten können dabei über unzählige Organisationen verteilt sein, sowohl was die Entwicklung, die Zuständigkeit und die Bereitstellung betrifft. Gleichzeitig erfordern diese Tatsachen eine umfassende Systemkenntnis, um Tests ausführen

zu können. Ein Testingenieur muss Services möglicherweise im Black-, Gray- und White-Box-Verfahren testen, weil eine Vielzahl heterogener Systeme an der Implementierung beteiligt sind [Wot+13]. Neben diesen technischen Herausforderungen darf die korrekte Umsetzung der Geschäftslogik nicht aus dem Blick verloren gehen.

Evolution und Dynamik

Eine der großen Herausforderungen beim Testen einer SOA ist ihre Flexibilität [PGFT11]. Services erlauben über ein Verzeichnis die Bindung mit unterschiedlichen Implementierungen zur Laufzeit. Der Test aller möglicher Bindungen gilt dabei meist als zu kostspielig und ineffizient [CDP06]. Darüber hinaus bedeutet die Beteiligung vieler Komponenten, dass regelmäßig neue Versionen einzelner Komponenten entstehen. Tests dieser neuen Versionen sollten möglichst alle Systemschichten beinhalten. Im Rahmen des später vorgestellten Governance-Konzepts (Abschnitt 2.5), wird die unkontrollierte Erzeugung von Abhängigkeiten zu Laufzeit vermieden. Im Unternehmensumfeld werden stattdessen, bei der Zusammensetzung einer SOA, mit Hilfe fester Verträge ganz bewusst Abhängigkeiten erstellt. Dazu wurden von [Zho+07a] und [Zho+07b] weitere Konzepte vorgestellt. Außerdem zeigt [HL05] eine Teststrategie, die auf diesen Verträgen basiert.

Unausgereifte Teststrategie

Es existieren keine allgemein anerkannten Richtlinien oder etablierte Testing-Tools zur Anwendung bei einer SOA [KG12]. Stattdessen wenden Tester häufig die ihnen bekannten Methoden an. Außerdem fehlt es meist an einer neuen übergeordneten Strategie, die beispielsweise den erhöhten Interaktionsaufwand zwischen den beteiligten Komponenten berücksichtigt. Ein Service kann zwar durch Unit-Tests gut und erfolgreich getestet werden, trotzdem können noch zahlreiche Fehler bei der Interaktion mit anderen Komponenten auftreten. Dann gestaltet sich die Fehler-Ursachen-Analyse als besonders schwierig. Die Fehlerursache wird durch die Weiterreichung von Teilergebnissen über mehrere Services zunehmend verschleiert. Die Aufgabe der Ursachen-Analyse umfasst dann beispielsweise die Verfolgung fehlerbehafteter XML-Nachrichten zwischen den betroffenen Komponenten.

Abhängigkeit vom Service Provider

Oft ist an der Umsetzung einer SOA mindestens eine Drittanbieter-Komponente beteiligt. Die Kontrolle über den Quellcode liegt dabei in der Hand des Drittanbieters. Zum Test steht dann möglicherweise nur die Service-Schnittstelle zur Verfügung. Werden bei einem Black-Box-Test eines solchen Services Fehler gefunden, kann die Behebung durch den beteiligten Service-Entwickler eine langwierige Angelegenheit werden. Der Ansatz in [Bar+09] zeigt wie Black-Box-Tests transparenter gemacht werden können, um aussagekräftiges Feedback von der

Black-Box-Komponente zu erhalten. Bei der Auswahl eines Service Providers soll deswegen der Fokus auch auf eine vertrauensvolle und zuverlässige Zusammenarbeit gelegt werden. Im schlimmsten Fall steht zum Testzeitpunkt einer SOA noch keine ausführbare Variante eines Services zur Verfügung. Dann müssen Abhängigkeiten durch Stumpf-Implementierungen ersetzt werden. Dem gilt es natürlich durch einen präzisen Testplan vorzubeugen, der eine optimale Testreihenfolge beinhaltet und darüber hinaus individuelle Projekt-Zeitpläne berücksichtigt.

Testbedingungen

Selbstverständlich müssen die Tests von SOA-Komponenten unter möglichst echten Bedingungen ausgeführt werden. Das heißt, die Testumgebung soll der späteren Einsatzumgebung sehr ähnlich sein, damit die Testergebnisse auf die Einsatzumgebung übertragen werden können. Zu testen sind funktionale und nicht-funktionale Aspekte sowohl von Services und Geschäftsprozessen [CDP06], als auch von der beteiligten Infrastruktur. Die spätere Einsatzumgebung kann aus vielen heterogenen Hard- und Softwaresystemen bestehen, was eine große Schwierigkeit für das Simulieren der Testbedingungen darstellt. Darüber hinaus soll jede Testausführung möglichst günstig sein und keine Auswirkungen auf die reale Welt haben.

Vielzahl an Stakeholdern

Stakeholder sind Einzelpersonen oder Organisationen, die entweder aktiv an einem Softwareprojekt beteiligt sind, oder bestimmte Interessen bezüglich des Projekts vertreten [Had12]. Bei der Fehlersuche und -behebung ist immer wieder die Zusammenarbeit mehrerer Stakeholder gefragt. Deshalb sollten vor der Ausführung eines Testplans stets die Zuständigkeiten aller beteiligten Stakeholder geklärt werden. Tabelle 2.1 fasst alle Stakeholder zusammen und beschreibt kurz ihre Beziehung zum Testing.

Stakeholder	Beschreibung	Test-Verantwortlichkeit
Service Provider	Stellt die Implementierung von Services zur Nutzung bereit. Trifft Vereinbarungen mit Service- Konsumenten über ein SLA (Service Level Agreement).	Testet die bereitgestellten Services anhand eines Spezifikationsdokuments, um funktionalen und nicht-funktionalen Anforderungen zu entsprechen. Außerdem ist er verantwortlich für die Bereitstellung der Testumgebung.
Service-Konsument	Organisation oder Person, die einen einzelnen oder einen zusammengesetzten Service nutzt. Schließt SLA mit dem Service Provider ab.	Führt End-to-End-Tests durch, um die Funktionalität zu testen.
Service Integrator	Führt bestehende Service-Implementierungen zusammen, um einen zusammengesetzten Service oder eine Benutzeranwendung zu erstellen.	Testet die Erfüllung der geschäftsprozesslichen Anforderungen.
Service-Entwickler	Plant, implementiert und liefert Schnittstelle und Implementierung eines Services. Verwendet zur Implementierung eventuell existierende Services.	Führt White-Box-Tests seiner Implementierung durch. Die Testbedingungen entsprechen nicht den späteren Einsatzbedingungen. Nicht-funktionale Tests können im Kleinen nicht realistisch durchgeführt werden.
Infrastructure Provider	Stellt einen Enterprise Service Bus (ESB) zur Verbindung einzelner SOA-Komponenten bereit. Ist für die Auffindung und Bindung durch ein Verzeichnis zuständig.	Testet hauptsächlich nicht-funktionale Anforderungen des ESB.
Endanwender	Verwendet ein System, hat aber keine Kenntnisse die dahinter stehende SOA-Technologie.	Akzeptanztests für die Endbenutzer-Anwendung.

Tabelle 2.1: Rolle der Stakeholder beim Testing [KG12]

2.3.2 Herangehensweisen

Einige Herausforderungen aus Abschnitt 2.3.1 können bewältigt werden, indem neue Service-Versionen in einer SOA gesammelt ausgebracht werden. Diese sogenannten Verbundreleases stellen einen Zeitpunkt dar, zu dem alle neuen und getesteten Versionen gleichzeitig veröffentlicht werden. Dieses Vorgehen stellt sicher, dass keine unkontrollierten Komponenten-Updates ins System gelangen, die nicht im Verbund mit anderen neuen Komponenten getestet wurden. So kann die Flexibilität einer SOA in einen bezwingbaren Umfang gebracht werden. Zu einem solchen Verbundrelease sollte zusätzlich eine einheitliche, übergeordnete Teststrategie erarbeitet werden, die Tests auf allen Ebenen der SOA vorsieht und ausführt [CDP09][RMI07]. Auf erster Ebene müssen Unit-Tests durchgeführt werden, welche besonders die Funktionalitäten und Fehlerbehandlungen einer Komponente sicherstellen. Schließlich muss die Interaktion mit anderen Komponenten in Composition-Tests und End-to-End-Tests getestet werden. Dabei werden gezielt Geschäftsprozesse ausgeführt und auf die Erfüllung ihrer funktionalen und nicht-funktionalen Anforderungen getestet. [JPW13] sieht dafür ein erweitertes *Business Process Model* vor. Dieses Modell verwendet Vor- und Nachbedingungen, die zur Testausführung einer Komponente ausgewertet werden. In [Bor+10] wird ein Framework zum Testen von Service-Kompositionen vorgestellt. [Bar+11] zeigt darüber hinaus wie die Code-Abdeckung bei Service-Kompositionen zur Testanalyse beiträgt. Zuletzt stellen Regressionstests sicher, dass durch die Einführung neuer Versionen keine alten Funktionalitäten beeinträchtigt werden.

2.3.3 Regressionstests

Anders als Unit-, Intergrations- und Systemtests, werden Regressionstests nicht zum Entwicklungszeitpunkt erstellt und ausgeführt. Regressionstests sollten als Teil der Wartungsarbeiten beim Einführen von jeglichen Änderungen in einer SOA ausgeführt werden. Dabei soll sichergestellt werden, dass durch keine der Neuerungen die Qualität von bestehenden Softwareteilen beeinträchtigt wird. Im Kontext der SOA stellen einige Gesichtspunkte Hindernisse besonders für Intergrations- und Regressionstests dar. [Moh+12] nennt als solche

- den Mangel an Beobachtbarkeit und Kontrolle von Quellcode und Struktur der Services,
- das hohe Maß an Dynamik und Anpassungsfähigkeit einer SOA,
- die hohen Kosten zur wiederholten Ausführung während der Tests.

Regressionstests können mit verschiedenen Ansätzen zur Auswahl der Testfälle [BP12] [Mei+12] umgesetzt werden. Bei selektiven Regressionstests wird eine Teilmenge aller Testfälle für den Regressionstest ausgewählt. Der Testingenieur wählt dafür anhand eines Kontrollflussgraphen diejenigen Testfälle aus, die seiner Meinung nach durch die Überführung in eine neue Version beeinflusst werden. Eine weitere Möglichkeit stellt das Code-basierte Testen dar. Dabei werden veränderter Code und damit zusammenhängende Codeteile durch Regressionstests getestet. Risikobasiertes Testing konzentriert sich auf Softwareteile, die durch eingeführte

Änderungen am meisten von Auswirkungen betroffen sind. Eine gute Auswirkungsanalyse zu jeder Änderung deutet dabei auf die betroffenen Teile hin.

2.3.4 Automatisierte Generierung von Testfällen

Es gibt bereits einige Tools auf dem Markt, die effiziente Unit-Tests von Services ermöglichen. IBM bietet dafür den *Rational Tester for SOA Quality* [1]. Eine Alternative heißt *SoapUI* [2]. Diese Tools generieren aus Artefakten wie WSDL-Dateien (Web Service Description Language), die in vielen Projekten als Service-Schnittstellenbeschreibungen dienen, automatisiert Testfälle.

2.3.5 Ermittlung von Abhängigkeiten

Die Abhängigkeiten in einer SOA sind von großer Bedeutung für die Fehlersuche und Auswirkungsanalyse. Sind die Abhängigkeiten bekannt, so können Fehlerursachen oftmals leichter gefunden werden. Bei der Auswirkungsanalyse helfen Abhängigkeiten bei der Risikobewertung, zum Beispiel im Falle eines Ausfalls einer Komponente. Wie in Abschnitt 2.3.1 vorgestellt, können in einer agilen SOA noch zur Laufzeit Entscheidungen über die Bindung eines Services getroffen werden. Deswegen gestaltet es sich als besonders schwierig die dynamischen Abhängigkeiten eines Systems zu ermitteln. In [BCD07] wird ein Software-Tool vorgestellt, das die Ermittlung von dynamischen Abhängigkeiten einer SOA zur Laufzeit ermöglicht. Auch statische Abhängigkeiten offenbaren viele Details einer Architektur. In einer DSM (Dependency Structure Matrix) angeordnet, können *bad smells* eines Designs ausgemacht, und die konsequente Durchsetzung verschiedener Entwurfsmuster überprüft werden [San+05]. Auch bei Open-Source-Projekten muss häufig sichergestellt werden, dass die Ursprungs-Architektur im Laufe der Zeit beibehalten wird [Tra+00].

2.4 SOA Governance

Unternehmen, die eine SOA einführen, müssen stets zielgerichtet auf die Vorteile der SOA hinarbeiten. Dazu zählt nicht nur die Einführung technologischer Neuerungen, sondern auch deren Ausrichtung auf den gesamten Softwareprozess und die Organisation der Projekte [GG07]. Um die zielgerichtete Arbeit mit exakter Ausrichtung auf Geschäftsziele zu erreichen, hat sich das Konzept der SOA Governance immer weiter durchgesetzt [DLDB09]. SOA Governance ermöglicht die Verwaltung von technischen und organisatorischen Strukturen mit dem Ziel der effektiven Anwendung des SOA-Musters. In [KSM14] werden elf Anforderungen erläutert, ohne deren expliziter Erfüllung es häufig zu Problemen bei der Umsetzung einer SOA

[1] http://www-03.ibm.com/software/products/de/servicetest
[2] http://www.soapui.org

kommt. In [BP09] wird der Begriff *SOA Test Governance* als Teilmenge der SOA Governance definiert. Im Zuge dessen wird die Schaffung von Methodiken, Prozessen, Notationen und Tools empfohlen, welche vermehrt bei den Integrationstests der Services erforderlich werden.

2.5 SOA Governance Repository

Das SOA Governance Repository (SGR)[KM16] stellt einen Ansatz zum umfassenden Steuern einer SOA dar. In seiner bestehenden Form ermöglicht es beispielsweise die Überwachung und Kontrolle von Konsumenten, Stakeholdern und Services in unterschiedlichen Versionen.

2.5.1 SOA Governance Meta Model (SOA-GovMM)

Das in diesem Abschnitt vorgestellte Datenmodell entspricht einer prototypischen Variante im Sinne des in [KM16] vorgestellten Metamodells. Dieses besteht aus vier Teilen (siehe Abbildung 2.1). Komponente *A: Service Provider* erlaubt die Modellierung von Services und Service-Versionen, die mit Hilfe von Service-Artefakten unterschiedlicher Art dokumentiert werden. Eine Service-Version befindet sich zu jeder Zeit in einem Life-Cycle-Zustand. Außerdem können die Endpunkte und Ausführungsumgebungen einer Service-Version erfasst werden. Die registrierten Endpunkte stehen in direkter Beziehung mit einem Konsumenten in Komponente *B: Service Consumer*. Die bestehenden Beziehungen werden umfassend durch eine Vielzahl möglicher Vertragseigenschaften in einem Vertrag dokumentiert. Darüber hinaus bieten die Komponenten *C: Organizational Structure* und *D: Business Object* Modellierungsmöglichkeiten für die Unternehmensstruktur und Geschäftsmodelle. Im Teil *C* können Personen mit gewissen Fähigkeiten (zum Beispiel zertifizierter Tester[3]) einer Rolle (zum Beispiel System-Tester) in der SOA zugeteilt werden. Um die Unternehmensstruktur abbilden zu können, werden Personen einer Abteilung zugewiesen. Komponente *D* ermöglicht die Modellierung von Geschäftsmodellen, abgekoppelt von den Servicestrukturen in Komponente *A*. Diese unternehmensspezifischen Objekte können in mehreren Versionen existieren und als Service-Artefakte in einer Implementierung dienen.

[3] http://www.german-testing-board.info/

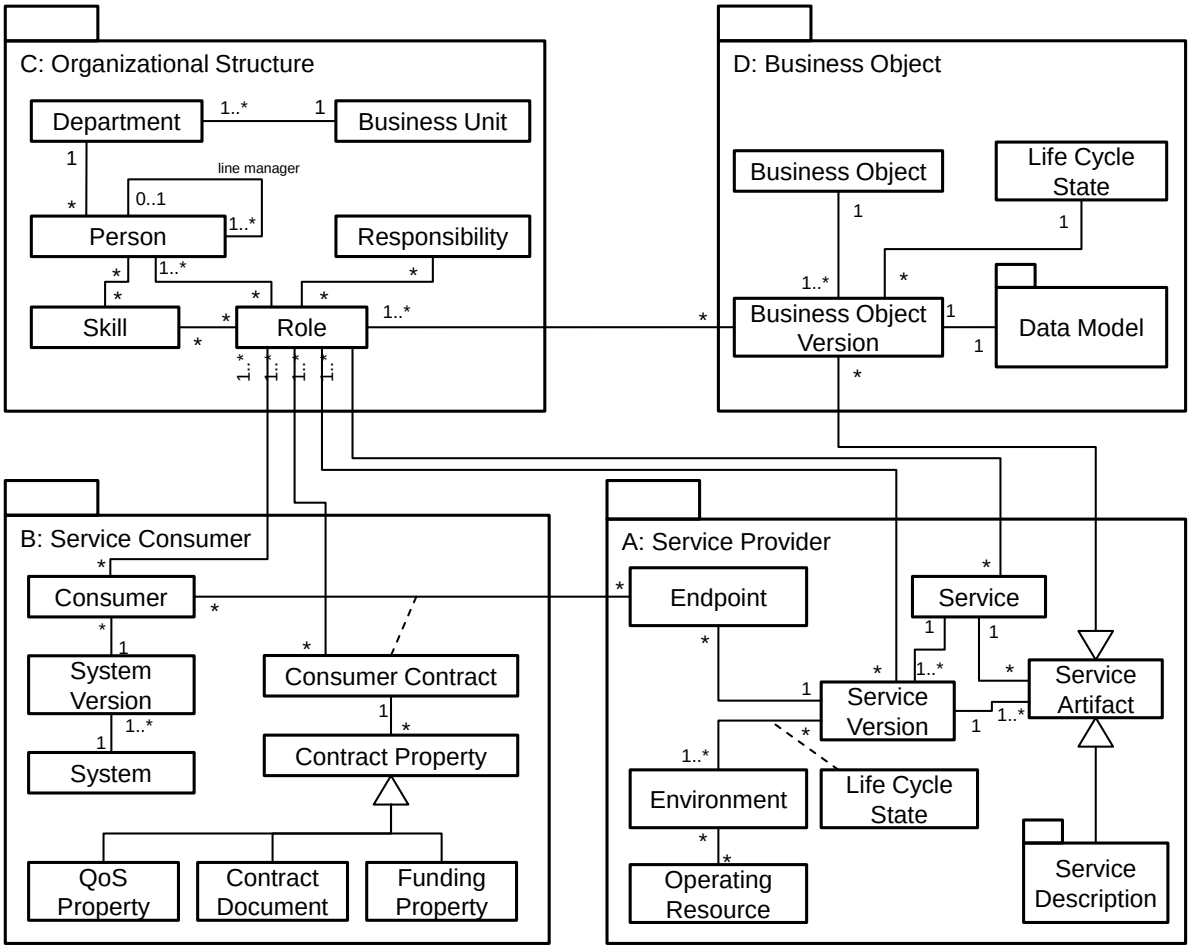


Abbildung 2.1: SOA Governance Meta Model [KSM14]

Eine vereinfachte Teilmenge des in Abbildung 2.1 dargestellten Metamodells bildet die Grundlage für das SGR. Die Klasse *Service* repräsentiert darin einen SOA Service, dessen tatsächliche Implementierungen durch eine oder mehrere Service-Versionen dargestellt werden (siehe Abb. 2.2). Für Konsumenten von Services besteht die Möglichkeit der Vertragsschließung mit einer bestimmten Service-Version. Verträge beinhalten alle relevanten Informationen wie Dokumente oder freie Schlüssel-Wert-Paare. Die Anpassung gegenüber des in [KM16] vorgestellten Metamodells besteht nun in der Assoziation von Services mit einem System. Semantisch bedeutet das, dass ein Service Teil eines Software-Systems ist. Ein System kann zum einen Konsument sein, indem das System Nutzungsverträge mit beliebigen Service-Versionen hält, zum anderen kann ein System Software-Services anbieten, die wiederum von anderen Systemen konsumiert werden dürfen. Diese Anpassung ermöglicht nun die Modellierung von beliebig großen und zusammenhängenden Service-Verbünden, was Voraussetzung für die weitere Konstruktion (in Kapitel 3 und 5) des Abhängigkeitsgraphen ist.

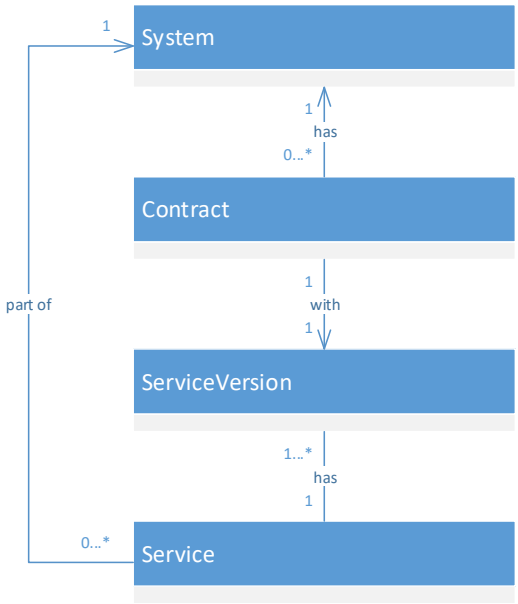


Abbildung 2.2: Dem SGR zu Grunde liegendes Datenmodell

2.5.2 Funktionen

Gemäß der elf zu Grunde liegenden Anforderungen [KSM14] an eine SOA Governance Software, stellt der Prototyp des SOA Governance Repositories [KM16] zum Beispiel Funktionen zur Verwaltung von Service Life Cycles [JNR09] bereit. Jeder im System vorhandenen Service-Version kann einer der Standardzustände des Life Cycles zugewiesen werden. Darüber hinaus besteht die Möglichkeit, weitere benutzerdefinierte Zustände ins System aufzunehmen, um den individuellen Anforderungen eines SOA-Prozesses gerecht zu werden. Basierend auf dem Life-Cycle-Zustand einer Service-Version können Akteure und das System entscheiden, welche Aktionen auf diese Service-Version anwendbar sind. Eine Service-Implementierung im Zustand *retired* eignet sich beispielsweise nicht zur Testzeitplan-Generierung und soll deshalb nicht Teil eines Testplans sein. Des Weiteren bietet das SGR die wichtige Möglichkeit, Konsumenten im System abzubilden. Auf diese Weise behalten alle beteiligten Akteure, die zur Einsicht berechtigt sind, den Überblick über aktiven Konsumenten und deren Nutzungsverhalten. Eine umfangreiche Benutzerverwaltung teilt jedem Benutzer mittels eines Rollensystems vordefinierte Berechtigungen zu. Auf diese Weise kann das SGR gezielt mit Benutzern, die eine bestimmte Rolle haben, interagieren. Die im SGR vordefinierten Rollen sind:

- Administrator
- Owner
- SystemResponsible

- TechnicalContact

Im Falle von Veränderungen der SOA, die einen Akteur beeinflussen können, wird zum Beispiel der zuständige Systemverantwortliche und der technisch zuständige Kontakt über deren notwendiges Eingreifen informiert. Mit diesem System kann die im Unternehmen vorherrschende Struktur gut abgebildet werden. Mehrere Visualisierungen auf dem Dashboard des SGR (siehe Abbildung 2.3) geben Auskunft über den Zustand der SOA. Diese helfen dabei, die Auslastung und Benutzungscharakteristik der SOA sowie den Reifegrad des eingesetzten Prozesses [IA07], einzuschätzen.

2.5.3 Prototyp

Abbildung 2.3 zeigt die Startseite des SGR. Eine Menüleiste mit sieben Tabs zeigt in allen Ansichten der Webanwendung die wichtigsten Einstiegspunkte des Systems. Am oberen rechten Rand befindet sich ein Benutzermenü, das dem angemeldeten Benutzer zur Hinterlegung von Daten wie E-Mail-Adresse, Organisation und Passwort verhilft. Das abgebildete Dashboard stellt konfigurierbare Widgets dar, die dem Benutzer, je nach dessen Berechtigung, einen Überblick über Service-Anfragen, Statistiken, Servernutzung und aufgetretene Ereignisse verschafft.

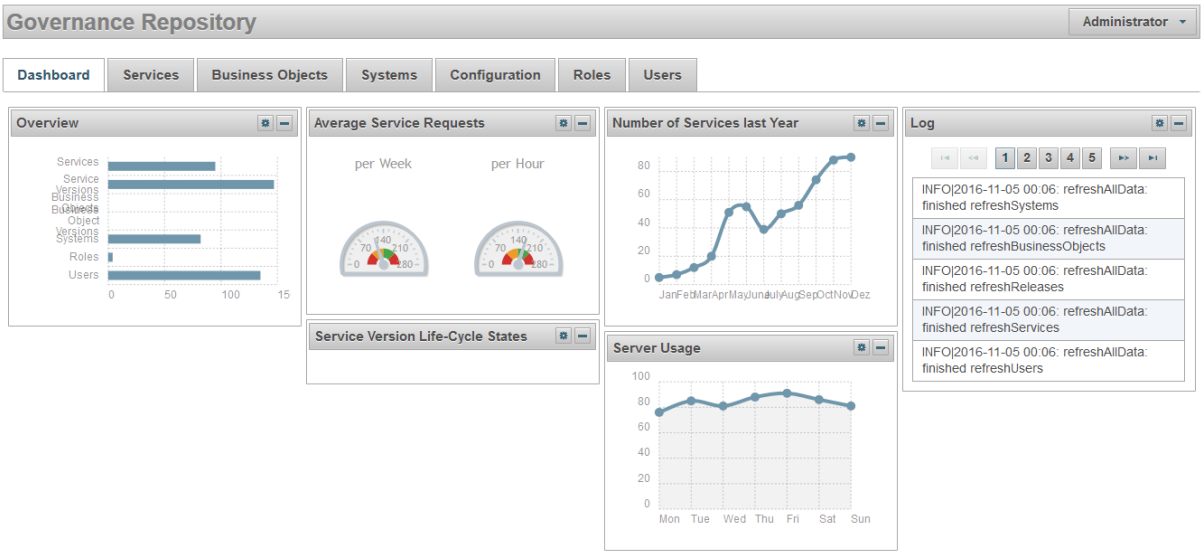


Abbildung 2.3: Dashboard des SGR

3 Konzept der Testzeitplan-Generierung

In diesem Kapitel wird die Eingabe der Abhängigkeiten entworfen. Des Weiteren wird der Aufbau eines Graphen aus den Abhängigkeiten zwischen Service-Anbietern und Konsumenten (siehe Abb. 3.1) konzeptioniert. Dieser Graph dient als Basis für die weitere Verarbeitung und die Generierung der Testzeitpläne. Das dabei entstehende Konzept dient als Grundlage für die Implementierung eines Prototypen.

Dem zu entwickelnden Prototypen liegt eine bestehende Software zu Grunde. Das SOA Governance Repository ermöglicht bereits die Erfassung von Service-Anbietern, Konsumenten und Nutzungsverträgen. Es sollen nun weitere Funktionalitäten implementiert werden, die die Testzeitplan-Generierung ermöglichen.

3.1 Anforderungen

In diesem Abschnitt werden die Anforderungen an den Algorithmus, dessen Eingaben, Arbeitsweise und Ergebnisse festgehalten.

3.1.1 Verbundreleases

In der Praxis werden Services, die Teil einer großen SOA sind, gebündelt veröffentlicht. Zu einem vorausgeplanten Zeitpunkt wird eine Teilmenge der SOA auf einen neuen Stand aktualisiert oder es werden neue Service-Anbieter hinzugefügt bzw. entfernt. Die Entwicklerteams arbeiten dabei gezielt auf ein solches Releasedatum hin.

Zur Testzeitplan-Generierung sollen beliebig viele solcher Releasezeitpunkte im SOA Governance Repository geplant werden können. Zu einem Verbundrelease soll der Benutzer einen Releasenamen, ein dazugehöriges Releasedatum und das Startdatum des Testzeitraums vergeben. Das Releasedatum legt den Zeitpunkt fest, zu dem die Testarbeiten aller Testkandidaten beendet sein müssen. Das Startdatum gibt an, ab welchem Zeitpunkt mit den Testarbeiten begonnen werden kann. Einem Release soll der Benutzer anschließend beliebig viele Service-Versionen hinzufügen können, die zum eingegeben Datum auf eine neue Version aktualisiert werden. Damit ist eine Service-Version Teil eines Releases zu diesem bestimmten Datum. Service-Versionen, die nicht zu einem Verbundrelease hinzugefügt wurden, verbleiben auch nach dem Releasezeitpunkt in ihrer gleichen Version. Es ist nicht nötig (und soll deshalb auch

nicht ermöglicht werden), Service-Konsumenten zu einem Verbundrelease hinzuzufügen, da diese von einer Service-Anbieter-Aktualisierung nicht beeinträchtigt werden. Falls Änderungen an der Schnittstelle eines Service-Anbieters auftreten, sollen all seine Konsumenten separat benachrichtigt werden.

3.1.2 Testaufwand

Die Erstellung eines Testzeitplanes bedarf der Erfassung von Aufwandsschätzungen. In einer Testperiode testet ein System exakt eine Service-Version. Die Dauer dieser Testperiode hängt dabei von mehreren Faktoren ab. Dazu können folgende Gesichtspunkte zählen:

- Umfang der angebotenen Service-Version
- Umfang der tatsächlichen Nutzung
- Komplexität der angebotenen Funktionalitäten
- Testdaten-Verfügbarkeit

Aus diesem Grund soll der geschätzte Testaufwand für jeden Nutzungsvertrag von einem Experten erfasst werden. Als Richtlinie hierfür eignen sich beispielsweise Erfahrungswerte aus vorhergehenden Tests oder Testprojekte mit ähnlichem Umfang. Die vergebenen Zeitspannen sollen als Testdauer für den Testzeitplan verwendet werden. Das kann zu folgenden, problematischen Konstellationen führen:

- Die vergebene Testdauer für eine einzelne Service-Version ist zu lang und überschreitet den Releasezeitpunkt
- Tests, die auf Grund ihrer Abhängigkeiten seriell ausgeführt werden müssen, überschreiten den Releasezeitpunkt
- Einer Service-Nutzung wurde keine Testdauer zugewiesen, obwohl diese Service-Version ein Teil eines Verbundreleases ist

Zum Zeitpunkt der Testzeitplan-Generierung sollen entsprechende Benachrichtigungen ausgegeben werden, falls eine oder mehrere dieser Konstellationen auftreten.

3.1.3 Abhängigkeiten

Damit Service-Versionen sinnvoll in einem Testzeitplan positioniert werden können, müssen die Abhängigkeiten unter ihnen bekannt sein. Der Benutzer soll in einer Prototyp Software die Nutzungsverträge unter Service-Anbietern und Service-Konsumenten abbilden können. So können beliebig komplexe, zusammengesetzte Services modelliert werden. Für die Testreihenfolge sollen die Nutzungsbeziehungen der Services berücksichtigt werden. Ein Service s soll erst dann getestet werden, wenn alle Services $s_{provider}$ getestet wurden, die von s konsumiert

werden. Diese Bedingung stellt sicher, dass zum Zeitpunkt des Tests alle zur Ausführung von s benötigten Service-Aufrufe in ihrer neuesten Version getestet wurden und somit funktionsfähig sind.

Außer den Abhängigkeiten, die durch die Benutzung von Service-Versionen entstehen, gibt es weitere Abhängigkeiten von unterschiedlichen Ressourcentypen. Während der Testphase kann die Testarbeit etwa von den folgenden Ressourcentypen abhängig sein:

- Personal
- Test-Infrastruktur
- Testdaten
- Lizenzen

Die Testausführung einer Service-Version könnte das Wissen eines zuständigen Stakeholders erfordern, der deshalb zur Testzeit anwesend sein muss. Des Weiteren könnten eine bestimmte Anzahl an Mitarbeitern für einen Test eingeplant werden. Außerdem werden häufig spezielle Maschinen der IT-Infrastruktur zur Testausführung benötigt, um die Einsatzumgebung (siehe Abschnitt 2.3.1) zu simulieren. Als Testfälle dienen Test-Eingaben, die manuell erstellt oder durch einen Algorithmus generiert wurden. Diese bilden in Verbindung mit den dazu erwarteten Ergebnissen die notwendigen Testdaten. Mit teuren Software-Lizenzen soll während der Tests sparsam umgegangen werden, damit keine unnötigen Kosten entstehen.

Eine fortgeschrittene Implementierung des SGR-Prototypen soll in der Lage sein, Ressourcen aller genannten Typen zu erfassen. Die Testzeitplan-Generierung soll dann die Möglichkeit bieten, verfügbare Ressourcen effizient zuzuweisen. Dies kann durch die Implementierung eines genetischen Algorithmus [BGW02; CP+08] zur Verteilung von Ressourcen erreicht werden.

3.2 Generierung von Testzeitplänen

Tabelle 3.1 beschreibt das prinzipielle Vorgehen der Testzeitplan-Generierung. Die nachfolgende Implementierung des Algorithmus basiert auf diesem Konzept. Alle notwendigen Schritte werden in den folgenden Abschnitten 3.2.1 bis 3.2.3 detailliert konzipiert. Schritt *5* der Testzeitplan-Generierung wird separat in Kapitel 4 entworfen.

Schritt	Beschreibung	Eingabe	Ausgabe
1	Aufbau des Abhängigkeitsgraphen mit allen relevanten Teilnehmern	Abhängigkeitsgraph mit Release-Informationen	Abhängigkeitsgraph ohne überflüssige Service-Versionen und Systeme
2	Umkehrung der Abhängigkeiten zur Testreihenfolge	Abhängigkeitsgraph aus Schritt 1	Gerichteter Graph mit Testbeziehungen zwischen Systemen und Service-Versionen als Testreihenfolge
3	Sortierung aller Teilbäume gemäß ihren Abhängigkeiten; Fehlermeldung, falls ein Zyklus erkannt wurde, und somit keine Sortierung möglich ist	Abhängigkeitsgraph aus Schritt 2	Abhängigkeitsgraph mit Sortierung durch Rangnummern
4	Erstellung aller Testperioden des Testzeitplans	Testreihenfolge aus Schritt 3, Testaufwandsschätzungen und benötigte Ressourcen	Testperioden mit Datumsinformationen und zugewiesenen Ressourcen
5	Optimierung je nach ausgewählten Optionen	Testperioden des Testzeitplans	Optimierter Testzeitplan

Tabelle 3.1: Generierungsvorgehen

3.2.1 Erstellung des Abhängigkeitsgraphen

Zunächst muss die Form des Abhängigkeitsgraphen und dessen Semantik formalisiert werden. In einem solchen Graphen gibt es drei Arten von Knoten:

- Typ I: Knoten, die ein System als reinen Konsumenten einer oder mehrerer Service-Versionen repräsentieren
- Typ II: Knoten, die ein System als Service-Provider und -Konsumenten repräsentieren
- Typ III: Knoten, die ein System als reinen Service-Provider von einer oder mehreren Service-Versionen repräsentieren

Ein System als reiner Konsument von beliebig vielen Service-Versionen ist in Abbildung 3.1 mit System 1 vertreten. Systeme, die ausschließlich als Konsument auftreten, sind immer ein Wurzelknoten des Abhängigkeitsbaums. Es muss mindestens einen Typ I Wurzelknoten

geben, da es sich andernfalls um einen zyklischen Graphen handelt. Zyklen können zwar im SOA Governance Repository modelliert werden, verhindern allerdings die topologische Sortierung in Abschnitt 3.2.2. Wie Zyklen erkannt und aufgelöst werden können, wird in Abschnitt 3.2.2 beschrieben. Systeme 2, 3 und 4 sind sowohl Konsumenten als auch Provider von Service-Versionen und stellen daher Typ-II-Knoten im Abhängigkeitsgraphen dar. Des Weiteren zeigt Abbildung 3.1 mit System 5 und 6 zwei Knoten, die als reine Service Provider fungieren (Typ III). Die Service-Versionen auf den Blättern des Baumes (System $System5$ und $System6$) können auch ohne ihre zugehörigen Systeme repräsentiert werden, da sie bereits durch den Service und die im Vertrag hinterlegte Service Version eindeutig identifiziert werden können.

Eine gerichtete Kante zwischen zwei Knoten c und sv existiert genau dann, wenn es einen Vertrag c zwischen einem System als Konsumenten und einer Service-Version gibt. Die Richtung der Kante zeigt dabei vom Konsumenten c zur Service-Version sv und stellt damit die Konsumenten-Beziehung dar. System c benutzt Service-Version sv. So ergibt sich zum Beispiel folgender Graph (Abb. 3.1) einer kleinen Service-Landschaft.

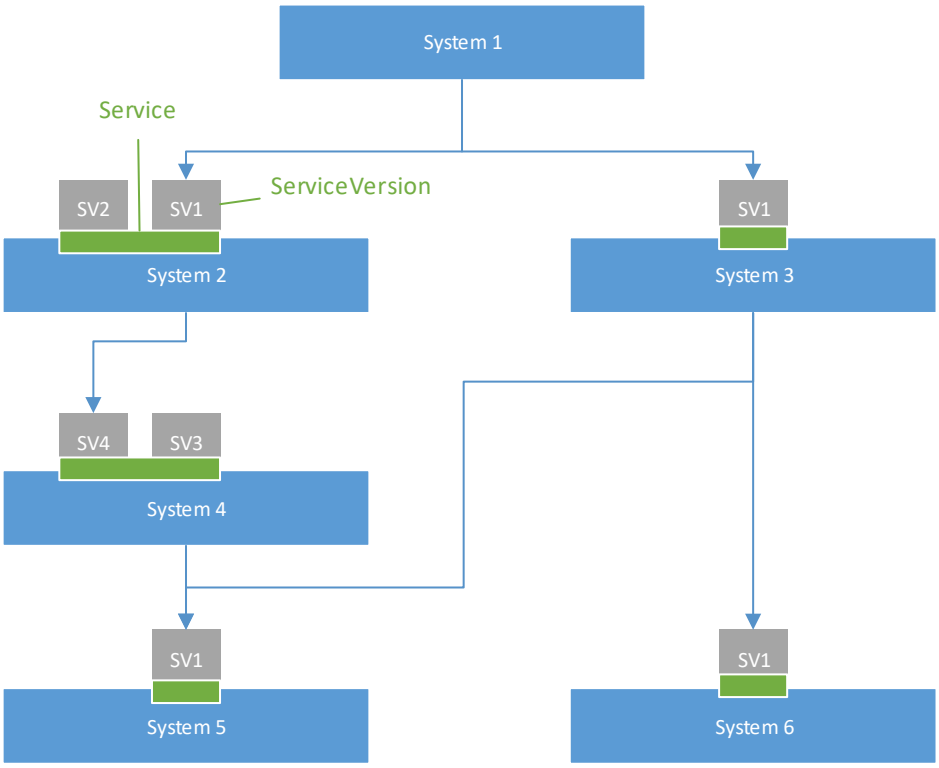


Abbildung 3.1: Systeme nutzen Service-Versionen über Verträge

Ausgehend von den vorhandenen Abhängigkeiten (wie in Abb. 3.1), müssen für die Testreihenfolge alle Kanten umgekehrt werden. Wenn $System1$ die Service-Version $SV1$ verwendet, wird die Kante umgekehrt mit der semantischen Folge: $SV1$ wird von $System1$ getestet. Es ergibt sich ein neuer Graph (siehe Abb. 3.2), der die Reihenfolge der Testbeziehungen darstellt. Gemäß Abbildung 3.2 würden also zunächst die Service-Versionen der Systeme 5 und 6 getestet werden und anschließen entsprechend des Testaufwands mit den nachfolgenden Service-Versionen fortgefahren. Im umgekehrten Graphen haben die Knoten daher folgende Semantik.

- Typ I: Knoten, die ausschließlich die verbundenen Service-Versionen testen
- Typ II: Knoten, die verbundene Service Versionen testen und Service-Versionen bereitstellen, die von verbundenen Systemen getestet werden
- Typ III: Knoten, die Service-Versionen bereitstellen, welche von verbundenen Systemen getestet werden

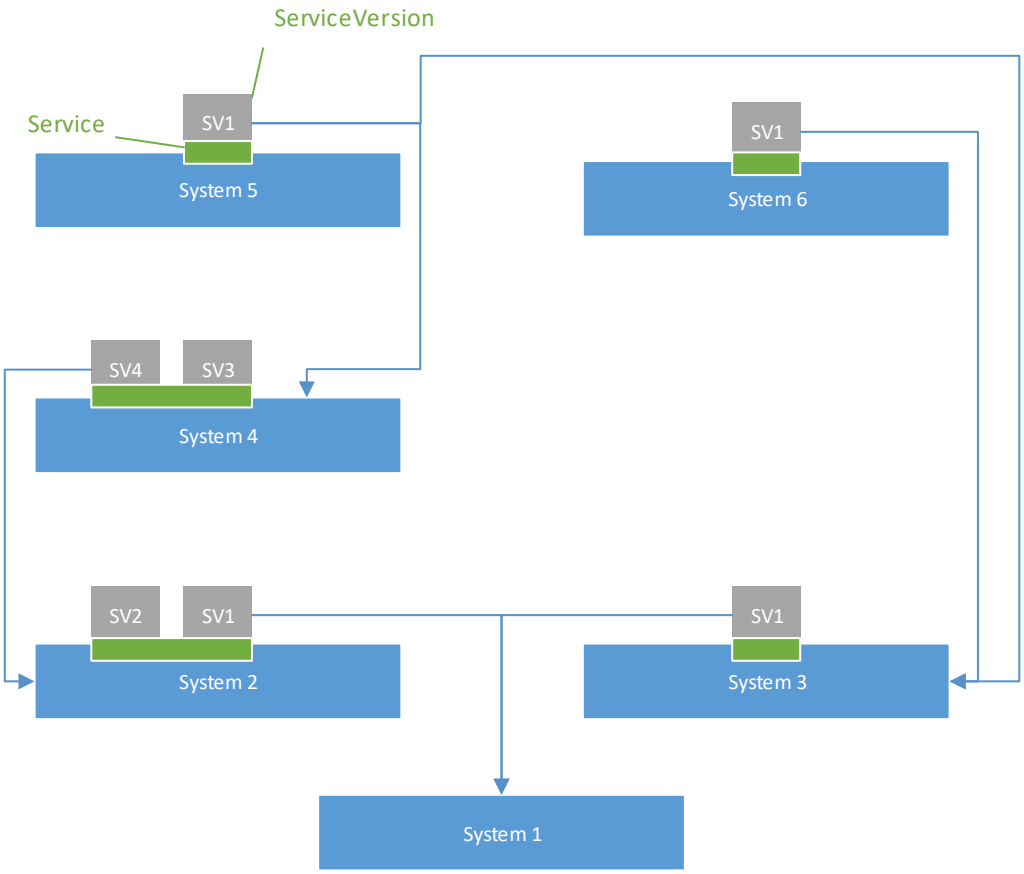


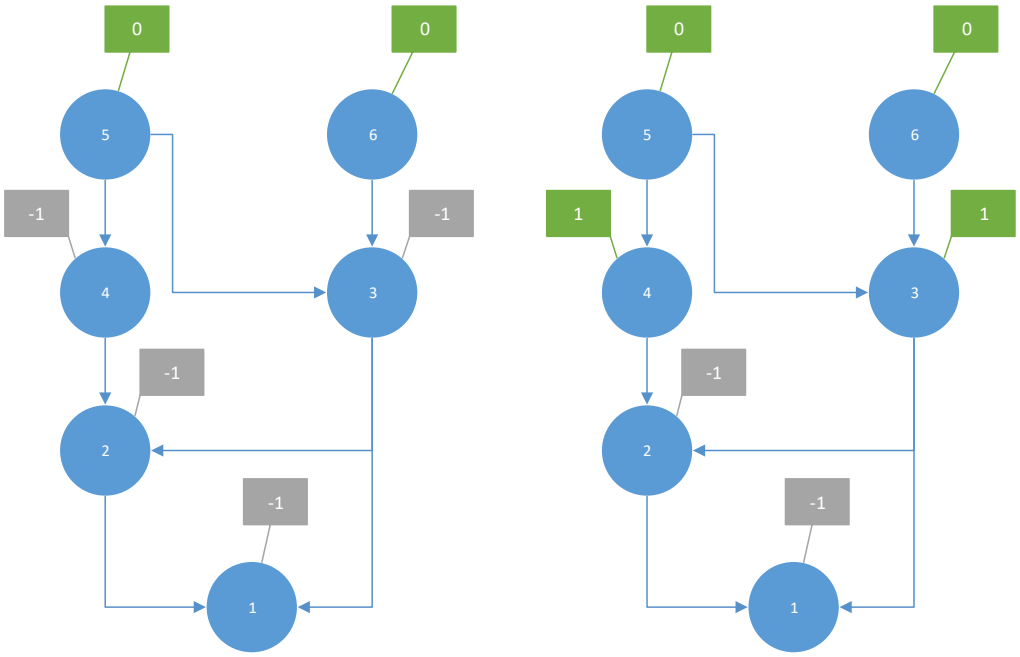
Abbildung 3.2: Testreihenfolge von Systemen und Service-Versionen

3.2.2 Topologische Sortierung

Im nächsten Schritt wird der in Abschnitt 3.2.1 erstellte Graph topologisch sortiert. Im Zuge dessen erhält jeder Knoten eine ganzzahlige Rangnummer r, die angibt, an welcher Stelle des Testzeitplans die Service-Version des jeweiligen Knoten getestet wird. Dies erleichtert im Folgenden die Arbeit mit dem Graphen, da die Reihenfolge nicht bei jedem Zugriff über die Kanten gesucht werden muss. Ein Graph kann aus beliebig vielen Teilbäumen bestehen, die unabhängig voneinander sortiert werden. Im Folgenden wird zur Vereinfachung immer nur ein Teilbaum betrachtet. Für die Sortierung werden zunächst alle Knoten mit der Rangnummer $r = -1$ initialisiert. Anschließend wird der Baum von oben (Wurzeln mit Rangnummer null) aufsteigend sortiert. Für die grafische Veranschaulichung in Abbildung 3.3 werden vereinfachte Darstellungen des Abhängigkeitsgraphen verwendet. Knoten der Typen I, II und III werden unabhängig ihrer Services und Service-Versionen gleich dargestellt. Die Initialisierung der Wurzelknoten (Zeilen $4 - 6$ in Algorithmus 3.1) markiert die Knoten 5 und 6 in Abbildung 3.3a mit $rank = 0$. In der ersten Iteration (Zeilen $7 - 13$) werden Knoten mit $rank = -1$ gesucht, die ausschließlich eingehende Kanten von Knoten mit $rank >= 0$ aufweisen. Bei allen gefundenen Knoten wird $rank = 1$ geschrieben. Abbildung 3.3b zeigt, dass die Knoten 3 und 4 gefunden wurden. Die nächste Iteration der While-Schleife schreibt $rank = 3$ im Knoten 2 (siehe Abb. 3.3c). Insbesondere wird Knoten 1 nicht aktualisiert, weil in Abbildung 3.3b nicht ausschließlich Kanten von Knoten mit positiver Rangnummer bei 1 eingehen. Erst in der dritten Iteration wird die Rangnummer von Knoten 1 mit $rank = 3$ geschrieben (siehe Abb. 3.3d).

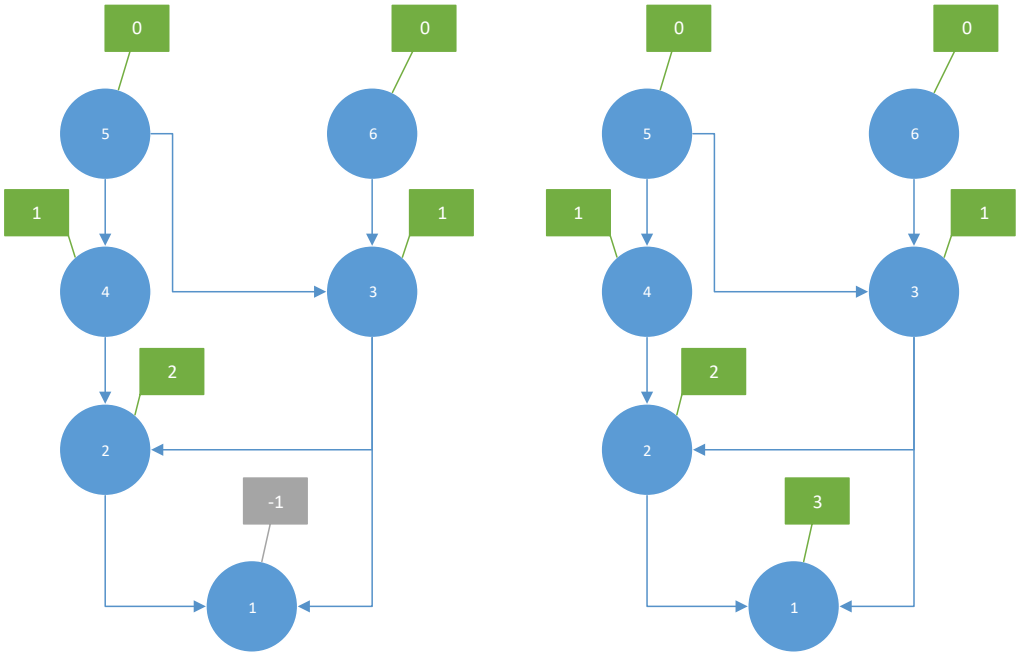
Algorithmus 3.1 Topologische Sortierung

```
1:  for all node ∈ graph do
2:      node.r = −1
3:  end for
4:  for all rootNode ∈ rootNodes do
5:      rootNode.r = 0
6:  end for
7:  rank = 1
8:  while ∀node ∈ graph, ∃node.r < 0 do
9:      for all negativeNode ∈ NegativeConnectedToPositivesOnly do
10:         negativeNode.r = rank
11:     end for
12:     rank + +
13: end while
```



(a) Initialisierung der Wurzelknoten mit $rank = 0$

(b) Iteration mit $rank = 1$



(c) Iteration mit $rank = 2$

(d) Iteration mit $rank = 3$

Abbildung 3.3: Voranschreitender Algorithmus im Beispielgraphen

Zykluserkennung

Besteht im Abhängigkeitsgraphen ein Zyklus, so können mehrere Operationen auf diesen nicht ausgeführt werden. Deswegen sollen Zyklen erkannt und eine entsprechende Information ausgegeben werden. Im Zuge der in Abschnitt 3.2.2 ausgeführten Sortierung soll die Zykluserkennung implementiert werden. Die schrittweise Traversierung des Graphen bietet sich hierfür an, da in jedem Schritt, den die Sortierung durchläuft, mindestens ein Knoten mit einer positiven Rangnummer gesetzt werden muss. Ist das nicht der Fall, wird der Algorithmus der topologischen Sortierung nicht terminieren. In Algorithmus 3.1 genügt es, die Liste *Negative-ConnectedToPositivesOnly* in Zeile *9* zu überprüfen. Ist die Liste leer, kann der Algorithmus nicht terminieren, und es existiert ein Zyklus im Graphen. Die Prozedur endet dann sofort mit der Rückgabe einer Fehlermeldung. Die Auflösung eines Zyklus sollte vom Benutzer erfolgen, da dies nur durch die Änderung der Eingabedaten und ihrer Semantik erreichbar ist. Um den Benutzer auf einen Zyklus hinzuweisen, sollen gefundene Zyklen ausgegeben werden. Um den Zyklus im Graphen auszugeben, kann eine Tiefensuche auf alle Teilgraphen ausgeführt werden. Jeder Knoten, der von der Tiefensuche erreicht wurde, wird dabei markiert. Wird ein Knoten ein zweites Mal erreicht und ist deshalb bereits markiert, wurde zunächst der Zyklus erkannt. Der bereits markierte Knoten wird als erster Knoten in die Liste *foundCycle* aufgenommen. Beim weiteren Fortsetzen der Tiefensuche gehören alle weiteren Knoten zum gefundenen Zyklus und werden in die Liste *foundCycle* aufgenommen. Sobald ein Knoten besucht wird, der bereits ein Teil der Liste *foundCycle* ist, wird die Tiefensuche abgebrochen. Als Hilfestellung, den gefundenen Zyklus aufzulösen, soll dieser dem Benutzer angezeigt werden.

3.2.3 Generierung von Testperioden

Zur Erstellung der Testperioden nutzt der Algorithmus den sortierten Abhängigkeitsgraphen (siehe Abb. 3.2 und 3.3). Beginnend mit dem Knoten, der die kleinste Rangnummer aufweist, werden jeweils alle notwendigen Informationen einer Testperiode erfasst. Dabei wird für jede eingehende Kante eine Testperiode erstellt. So hat $System3$ beispielsweise zwei eingehende Kanten und testet deshalb seine beiden Abhängigkeiten. Insbesondere haben Knoten mit Rangnummer null keine eingehenden Kanten. Das bedeutet, dass Systeme mit Rangnummer null keine Testperiode zum Testen ihrer nicht vorhandenen Abhängigkeiten bekommen.

Eine Testperiode beinhaltet die folgenden Daten:

- Testendes System
- Zu testende Service-Version
- Beginn der Testperiode
- Ende der Testperiode
- Zugewiesene Ressourcen

Sowohl das testende System als auch die zu testende Service-Version können direkt aus der Struktur des in Abschnitt 3.2 aufgebauten Graphen entnommen werden. Beginn und Ende der jeweiligen Testperiode ergeben sich aus den Enden vorheriger Testperioden und dem geschätzten Testaufwand. Der geschätzte Testaufwand ist Teil eines Vertrags, der in einer Kante hinterlegt wird. Somit wird jeder Testperiode eine individuelle zeitliche Dauer zugewiesen. Eine Testperiode soll so früh wie möglich starten. Sind alle vorher geplanten Testperioden abgeschlossen, soll ohne Verzögerung mit der nächsten Testperiode begonnen werden. In Abbildung 3.2 kann $System1$ beispielsweise sofort mit einer Testperiode für die verbundene Service-Version beginnen, sobald eines der Systeme 2 oder 3 alle Testperioden abgeschlossen hat. An dieser Stelle ist es für $System1$ unnötig, auf das Ende aller Testperioden der vorherigen Testperioden zu warten. Die Ressourcenzuweisung erfolgt auf Basis der verfügbaren Ressourcen aller Typen, die in Abschnitt 3.1.3 vorgestellt wurden. Die Testzeitplan-Komponente prüft, welche Ressourcen eines Typs während der jeweiligen Testperiode benötigt werden und weist der Testperiode verfügbare Ressourcen dieses Typs zu. Sollten nicht genügend Ressourcen eines Typs verfügbar sein, muss der Algorithmus zur Ressourcenverteilung der Beginn und das Ende der Testperiode zeitlich verschieben, bis alle benötigten Ressourcen verfügbar sind.

3.3 Darstellung eines Testzeitplans

Dieser Abschnitt stellt die mögliche Darstellungsform eines Testzeitplans vor. Unterschiedliche Generierungsoptionen (deren Konzepte werden im Kapitel 4 vorgestellt) führen dazu, dass die Testzeitplan-Generierung zu einem Verbundrelease unterschiedliche Testzeitpläne erstellen kann. Um verschiedene Testzeitpläne visuell vergleichen zu können, wird in Abschnitt 3.3.2 ein Konzept vorgestellt, welches dies ermöglicht.

3.3.1 Grafische Darstellung eines Testzeitplans

Einer der großen Vorteile von generierten Testzeitplänen ist die parallele Testbarkeit der Services. Der Benutzer der Testzeitplan-Generierung möchte sich nach Ende der Generierung einen Überblick über den entstandenen Testzeitplan verschaffen. Außerdem soll der Benutzer in die Lage versetzt werden, die Brauchbarkeit des Testzeitplans zu hinterfragen. In der Visualisierung des Testzeitplans sollten deswegen folgende Punkte zum Ausdruck kommen:

- Welche Services werden zu welchem Zeitpunkt getestet?
- Welches ist der kritische Pfad?
- Wann ist der Releasezeitpunkt?
- Wie viel Zeit bleibt für den aktuellen Test?

Der kritische Pfad besteht aus lückenlos aneinander gereihten Testperioden. Seine Testperioden bilden in Summe den längsten Pfad von Anfang bis Ende des Testzeitplans. Darüber hinaus wirkt sich die Verzögerung einer Testperiode immer auf die Gesamtdauer des Testzeitplans aus.

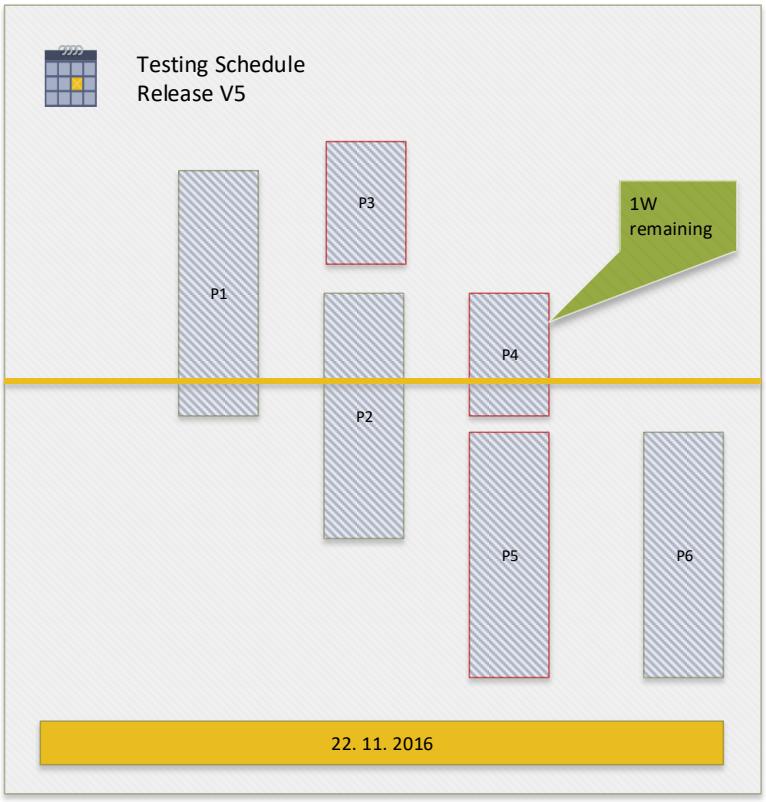


Abbildung 3.4: Prototypische Darstellung eines Testzeitplans

Abbildung 3.4 zeigt eine Möglichkeit, all diese Informationen möglichst komfortabel zu visualisieren. Jede Testperiode wird durch einen Balken Pn dargestellt. Der kritische Pfad, der für die Gesamtdauer des Testzeitplans verantwortlich ist ($P3$, $P4$, $P5$), wird durch rot umrahmte Balken ausgedrückt. Der Releasezeitpunkt wird gut lesbar im gelben Balken, am Ende aller Testperioden, angezeigt. Beim Mouseover-Event einer Testperiode werden in einer grünen Sprechblase weitere Details zur verbleibenden Testzeit dargestellt.

3.3.2 Vergleich mehrerer Testzeitpläne

Die verfügbaren Optionen, Optimierungen und Eingaben zu einem Verbundrelease können zu verschiedenen Testzeitplänen führen. Der Benutzer soll möglichst einfach in der Lage sein, die Unterschiede der Pläne zu erkennen, und den am besten geeigneten Plan auszuwählen.

Visueller Vergleich

Das Konzept des visuellen Vergleichs sieht die Darstellung zweier oder mehrerer Pläne nebeneinander vor. Über die beiden Buttons am oberen Rand eines Testzeitplans soll der dargestellte Zeitplan in einem der Container ausgewählt werden. Der Benutzer wählt denjenigen Testzeitplan aus, der für das Szenario besser geeignet ist. Die Abbildung 3.5 zeigt mit einem blauen Rand um den Container, dass der Benutzer den rechten der beiden angezeigten Testzeitpläne für sein Szenario ausgewählt hat.

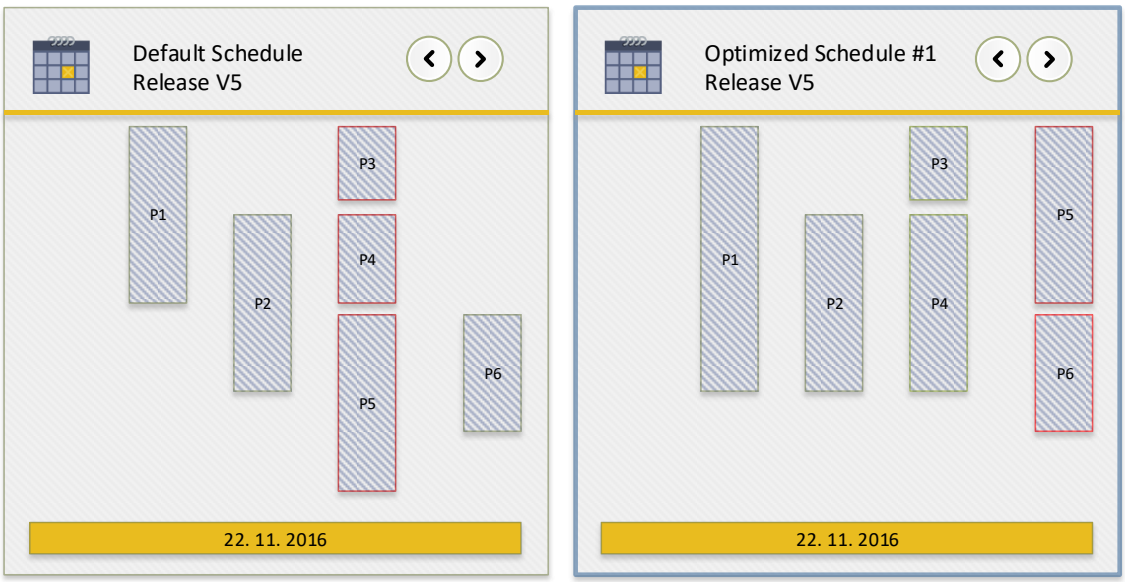


Abbildung 3.5: Prototypische Darstellung des visuellen Testzeitplan-Vergleichs mit zwei Kandidaten

Durch den direkten Vergleich mehrerer Pläne kann der Benutzer die Auswirkungen der gewählten Optionen deutlich erkennen. Die Auswahl des am besten geeigneten Testzeitplans ist eine verantwortungsvolle Aufgabe. Dafür sind weitreichende Kenntnisse zum Testing- und Veröffentlichungsprozess nötig. Im nächsten Abschnitt wird daher eine Vergleichsmethodik vorgestellt, mit Hilfe derer eine automatisierte Entscheidung zur Auswahl eines Testzeitplans getroffen werden kann.

3.3.3 Vergleich der maßgeblichen Kenngrößen

Ein Testzeitplan wird durch mehrere Faktoren charakterisiert, die beispielsweise durch die nachfolgenden Kenngrößen quantifiziert werden können:

1. Gesamt-Testdauer

2. Durchschnittliche Testdauer einer Service-Version
3. Grad der Parallelität
4. Anzahl der Testphasen im kritischen Pfad
5. Anzahl vergebener Ressourcen
6. Verfügbarkeit von Ressourcen zu allen Zeitpunkten

Durch den Vergleich einzelner Kenngrößen mehrerer Testzeitpläne kann eine Entscheidung getroffen werden, welcher Testzeitplan eher den bestehenden Anforderungen entspricht. Die Gewichtung einzelner Faktoren lässt dabei die Individualisierung der Anforderungen zu. Die individuelle Gewichtung folgender Anforderungen führt, durch eine direkte Verknüpfung mit den entsprechenden Kenngrößen, zur gewünschten Anpassung der Entscheidung.

- Schnelle Testdurchführung
- Terminsicherheit
- Ressourcen-Sparsamkeit

Liegt die Priorität bei der Auswahl eines Testzeitplans zum Beispiel zu 50 % bei einem frühen Ende aller Testperioden und zu 40 % bei der Sicherheit, die Termine einhalten zu können, so könnten die in Abbildung 3.6 vorgeschlagenen Gewichte vergeben werden. Der Algorithmus misst dann die Kenngrößen aller generierten Testzeitpläne und vergleicht diese unter Berücksichtigung der vergebenen Gewichte.

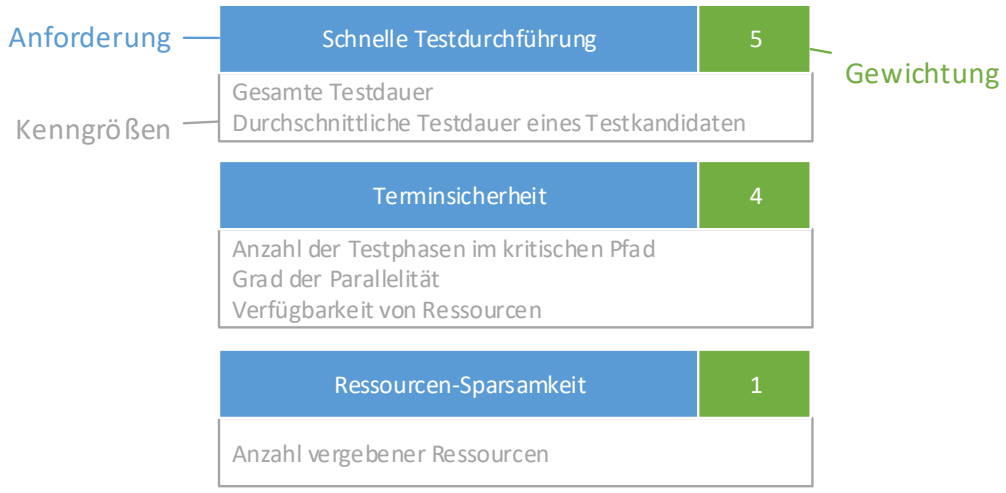


Abbildung 3.6: Beispielhafte Gewichtung von Anforderungen und deren Verknüpfung mit messbaren Kenngrößen

4 Konzept der Optimierungen

In diesem Kapitel werden Konzepte vorgestellt, die zur Optimierung und Individualisierbarkeit der generierten Testzeitpläne führen. Bevor die Generierung des Testzeitplans gestartet wird, soll aus einer Reihe von Optimierungsmöglichkeiten gewählt werden, welche Optimierungen zur Ausführung gebracht werden. Die folgende Tabelle 4.1 ordnet den konzipierten Optimierungsverfahren einen eindeutigen englischen Namen zu, der in der Implementierung zum Einsatz kommen soll.

Englischer Name der Optimierung	Beschreibung
Expanding	Ausdehnung von Testperioden auf die maximal verfügbare Testdauer
EarlyStart	Beginn des Testzeitplans zum frühesten Datum
LateStart	Ende des Testzeitplans zum Releasedatum
Clipping	Trennung des kritischen Pfads

Tabelle 4.1: Übersicht der Optimierungen

4.1 Testperioden expandieren

Die vergebenen Aufwandsschätzungen (siehe Abschnitt 3.1.2) bilden die Grundlage für die Dauer einer Testperiode im Testzeitplan. Durch einige Konstellationen im Abhängigkeitsgraphen kann es allerdings dazu kommen, dass mehr Testzeit zur Verfügung steht, als die geschätzte Testdauer beträgt. Wird eine Service-Version von mehreren Systemen mit unterschiedlichem Testaufwand getestet, kann das Optimierungsverfahren angewendet werden. Im umgekehrten Abhängigkeitsgraphen gehen im beschriebenen Szenario zwei oder mehr Kanten von einer Service-Version aus. Jedes System, das die Service-Version testet, muss auf den Abschluss aller beteiligten Testperioden warten. Erst mit dem Ende aller Testperioden der beteiligten Systeme gilt die Service-Version als getestet. Deshalb kann die Dauer einer beteiligten Testperiode auf die maximale Testdauer aller beteiligten Testperioden erhöht werden. Wenn die hier beschriebene Optimierung zur Anwendung kommt, werden alle Testperioden auf die maximal mögliche Testdauer vergrößert, ohne dass sich die Dauer des gesamten Testzeitplans verändert. In Abbildung 4.1 wirkt sich das Expandieren der Testperioden auf Testperiode $P2$ aus. Der kritische Pfad im Zeitplan verläuft über die rot umrandeten Testperioden von $P1$ über $P3$ zu

$P4$. Da $P2$ nicht Teil des kritischen Pfads ist, und nicht über die Testdauer von $P1$ hinaus expandiert wird, verändert sich die Gesamtdauer des Zeitplans nicht. Durch die Verlängerung einer Testperiode, kann die Beziehung der getesteten Komponenten umfassender getestet werden.

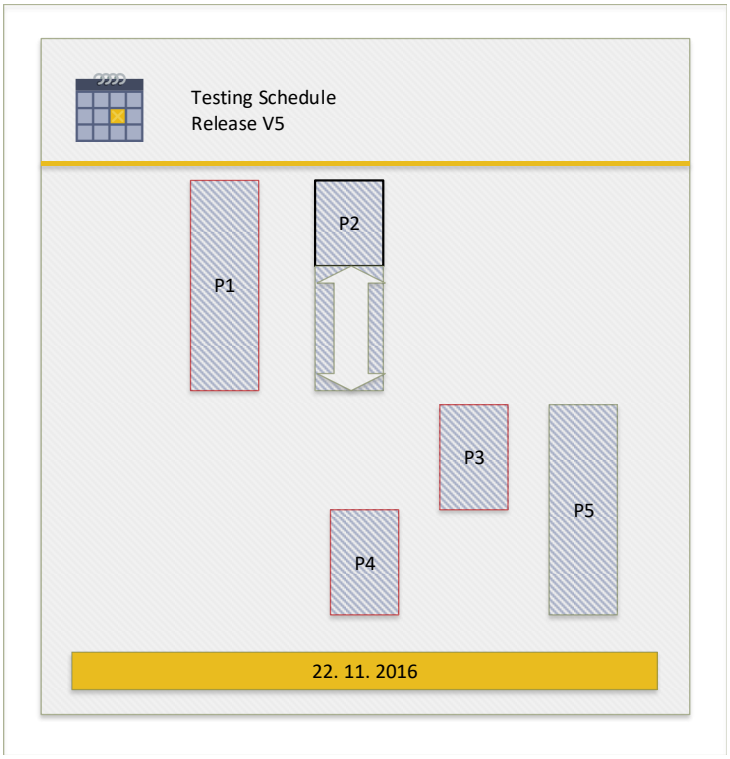


Abbildung 4.1: Expandieren der Testperioden

4.2 Früher Teststart

Bei der Ausführung dieses Verfahrens soll der Testzeitplan mit einem vorgegebenen Test-Startdatum beginnen. Diese Option ermöglicht die Planung der Testphase mit einem explizit festgelegten Startdatum. Je nach der Dauer des gesamten Testzeitplans kann das Ende aller Tests damit sowohl vor als auch nach dem geplanten Releasezeitpunkt liegen.

1. Fall I - Das Ende des Testzeitplans liegt vor dem Releasezeitpunkt:
   Nach Ende des Testverlaufs entsteht ein Puffer-Zeitraum zwischen Testende und Releasezeitpunkt. Eine kurze Puffer-Phase kann in der Praxis von Vorteil sein, falls die tatsächliche Dauer der Testphasen die geplanten Testperioden überschreitet.
2. Fall II - Das Ende des Testzeitplans entspricht exakt dem Releasezeitpunkt:
   Wenn der Testzeitplan exakt eingehalten werden kann, schließen die Testphasen nahtlos an das geplante Verbundrelease an. Dadurch entsteht keine Puffer-Phase, die im Falle des

Testverzugs genutzt werden könnte. Insbesondere entspricht dieser Fall dem Ergebnis der Option aus Abschnitt 4.3.

3. Fall III - Das Ende des Testzeitplans liegt nach dem Releasezeitpunkt:
   Selbst wenn alle Testphasen die geschätzte Testdauer einhalten können, reicht der Testzeitplan nicht aus, um bis zum geplanten Releasezeitpunkt abgearbeitet zu sein. In diesem Fall soll grafisch eine deutlich sichtbare Warnung ausgegeben werden.

Diese Option und die Option aus Abschnitt 4.3 schließen sich in ihrer Anwendung gegenseitig aus, weil die Dauer von Testperioden nicht frei vergeben werden soll.

4.3 Spätes Testende

Diese Option bewirkt, dass ein Testzeitplan immer zum Releasezeitpunkt endet. Dabei kann sich der Beginn des Testzeitplans gemäß der Dauer des Testzeitplans verschieben. Auch hier gibt es drei Fälle, in denen der Beginn der Testphasen unterschiedlich starten kann.

1. Fall I - Der Beginn des Testzeitplans liegt in der Zukunft
   Der Testzeitplan kann regulär zum, vom Algorithmus ermittelten Startzeitpunkt, beginnen und endet zum Releasezeitpunkt. Dabei entsteht keine Puffer-Phase.
2. Fall II - Der Beginn des Testzeitplans liegt auf dem derzeitigen Datum
   Der Testzeitplan sieht den sofortigen Beginn der Testphasen vor, damit ein Ende zum Releasezeitpunkt erreicht werden kann.
3. Fall III - Der Beginn des Testzeitplans liegt in der Vergangenheit
   Die Testarbeit hätte bereits beginnen müssen, um ein Ende zum Releasezeitpunkt erreichen zu können. In diesem Fall soll grafisch eine deutlich sichtbare Warnung ausgegeben werden.

Diese Option und die Option aus Abschnitt 4.2 schließen sich in ihrer Anwendung gegenseitig aus, weil die Dauer von Testperioden nicht frei vergeben werden soll.

4.4 Teilen des kritischen Pfads

Ähnelt der Abhängigkeitsgraph einem degenerierten Baum, also einer geordneten Liste, so empfiehlt sich die Anwendung dieser Optimierung. Ein degenerierter Baum verursacht Testphasen, die größtenteils nur seriell abgearbeitet werden können. Dies führt zu einer relativ langen Gesamtdauer des Testzeitplans. Ein degenerierter Baum wird bei der Anwendung dieser Optimierung in zwei Teilbäume zerlegt. Dieses Vorgehen erhöht die Parallelisierung der Testperioden, wodurch sich die Gesamtdauer des Testzeitplans deutlich verkürzen kann. Jene Abhängigkeit, an welcher der kritische Pfad des Baums geteilt wird, muss am Ende

eines Teilbaums angehängt werden, um die Vollständigkeit des Testzeitplans zu gewährleisten. Abbildung 4.2 zeigt zwei vereinfachte Abhängigkeitsgraphen. Die folgenden Abschnitte behandeln die einzelnen Teilaufgaben beim Teilen eines degenerierten Baums. Die erhöhte Nebenläufigkeit der Testperioden im Vergleich zur seriellen Testausführung führt in der Regel auch zu einem hohen Ressourcenbedarf während der Tests. Wenn viele Testperioden parallel geplant werden, kann das eine stärkere Belastung der Infrastruktur bedeuten. Dieses Problem muss vorab mit dem *Infrastructure Provider* (siehe Abschnitt 2.3.1) eingeplant und kompensiert werden.

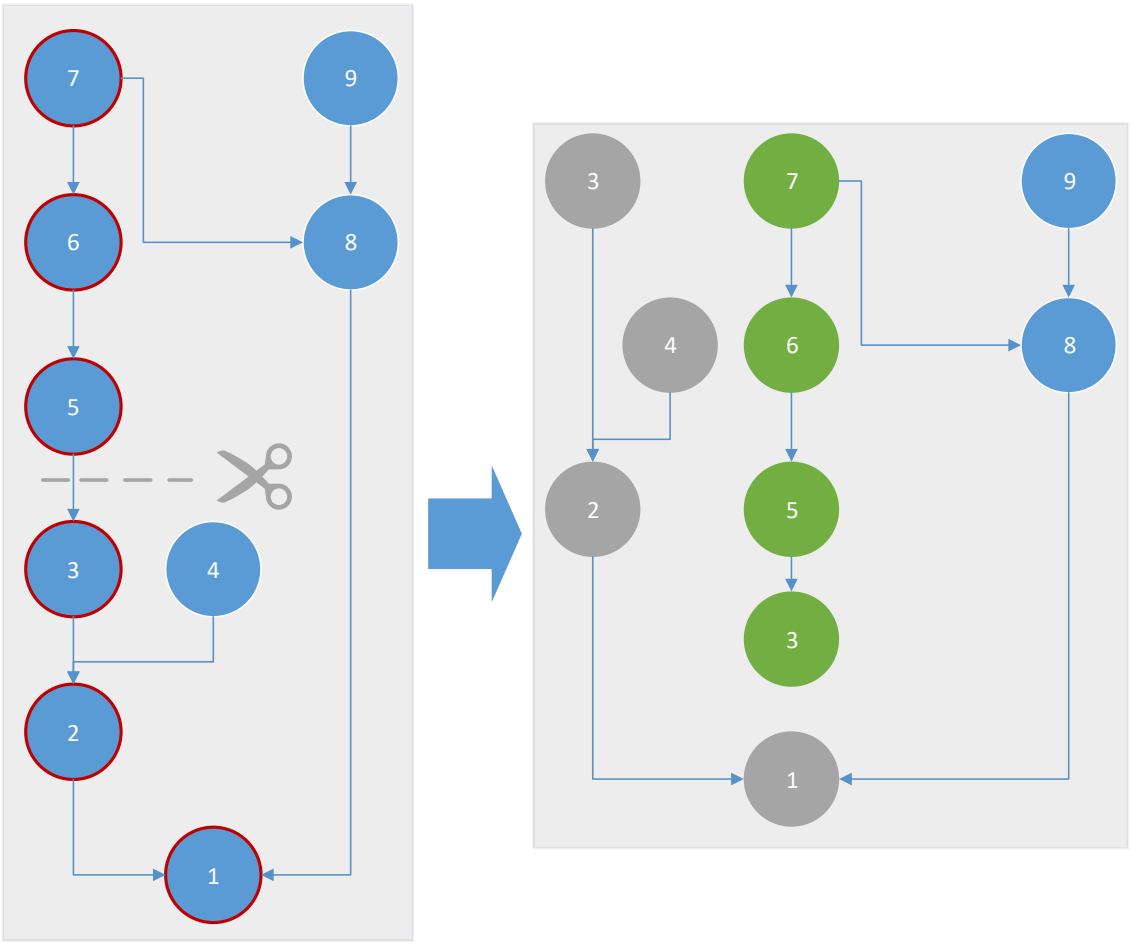


Abbildung 4.2: Teilen des kritischen Pfads

4.4.1 Kriterium zur Anwendung

Die Anwendung der Optimierung hängt vom jeweiligen Szenario ab. Das Durchtrennen des kritischen Pfads ist nur dann von Vorteil, wenn sich dadurch die Gesamtdauer des Testzeitplans

und damit die Gesamtdauer aller Testperioden auf dem kritischen Pfad verkürzt. Den größten Nutzen bietet diese Methode, wenn der kritische Pfad eine wesentlich längere Testdauer als die übrigen Pfade aufweist, und außerdem eine gleichmäßige Verteilung der Testaufwände unter den Testperioden besteht. Erst dann lässt sich der kritische Pfad gut aufteilen. Zunächst soll durch eine Tiefensuche der kritische Pfad und dessen Länge bestimmt werden. Die Länge des Pfads ergibt sich aus der Summe aller Testaufwände im kritischen Pfad (siehe Gleichung 4.1).

$$L_{crit} = \sum_{c \in crit} c_{effort} \tag{4.1}$$

Kriterium zur Anwendung der Optimierung:

$$\sum_{c \in crit} c_{effort} > (\sum_{c \in upper} c_{effort}) + cut_{effort} \tag{4.2}$$

Aus der Teilung entstehen die neuen Teilgraphen $upper$ und $lower$ (grün und grau gefärbt in Abbildung 4.2). Das Kriterium (siehe Ungleichung 4.2) besagt, dass der gesamte Testaufwand des $upper$-Teilgraphen mit zusätzlicher Testperiode für die Schnittstelle (siehe Abschnitt 4.4.3) kleiner als der vorherige kritische Pfad sein muss. L_{crit} sollte möglichst in der Hälfte teilbar sein. Dafür soll die Varianz der Testperioden auf dem kritischen Pfad möglichst klein sein, um eine gute Teilbarkeit des Pfads zu erreichen.

4.4.2 Schnittpunkt ermitteln

Beim Teilen des Pfads wird diejenige Kante aus der Testreihenfolge im kritischen Pfad entfernt, die möglichst nahe am Mittelpunkt der gesamten Testdauer in diesem Pfad liegt. Zur Ermittlung der optimalen Kante werden die Testperioden stückweise aufsummiert, bis die exakte Hälfte der gesamten Testdauer L_{crit} bei der Kante an Stelle i überschritten wurde. Dann wird entschieden, ob Kante i oder $i-1$ näher am exakten Mittelpunkt von L_{crit} liegt, und diese Kante schließlich entfernt.

Für die Beispielgrafik 4.2 kann angenommen werden, dass alle Testperioden $1-9$ exakt gleich lang dauern. In diesem Szenario wird das Kriterium aus Abschnitt 4.4.1 erfüllt, da der kritische Pfad mit seinen sechs Testperioden eine doppelt so lange Testdauer aufweist, wie alle anderen Pfade. Darüber hinaus beträgt die Varianz der Testdauer in diesem Beispiel 0. Der optimale Schnittpunkt des kritischen Pfads liegt folglich exakt zwischen den Testperioden 3 und 5.

4.4.3 Schnittpunkt testen

Dadurch, dass der untere Teilbaum (grauer Teilgraph in Abb. 4.2) vom kritischen Pfad abgetrennt wird, kann die Testarbeit sofort mit dem unteren Teilgraphen beginnen. Die abgetrennte Abhängigkeit wird zum Testbeginn also zunächst vernachlässigt, um eine hohe Parallelität der

Testperioden zu erreichen. Trotzdem soll natürlich die Interaktion der betroffenen Systeme getestet werden. Aus diesem Grund wird am Ende der Tests des oberen Teilgraphen (grüner Teilgraph in Abb. 4.2) die erste Testperiode des unteren Teilgraphen angehängt.

4.4.4 Terminierung der Optimierung

Nach einem Durchlauf der beschriebenen Optimierung ergibt sich ein veränderter kritischer Pfad. Dieser neue kritische Pfad kann wieder geteilt werden, wenn daraus gemäß Abschnitt 4.4.1 abermals eine Verkürzung des Testplans erreichbar ist. Je öfter die Optimierung ausgeführt wird, desto mehr Testperioden kommen, wie in Abschnitt 4.4.3 behandelt, zum Testplan hinzu.

5 Implementierung der vorgestellten Konzepte

Dieses Kapitel stellt die prototypische Implementierung des Algorithmus zur Testzeitplan-Generierung vor und präsentiert Screenshots der grafischen Oberfläche. Die Komponenten wurden Teil des im Model-View-ViewModel (MVVM) Entwurfsmuster geschriebenen SOA Governance Repository. Die automatisierte Testzeitplan-Generierung ist zu einem festen Bestandteil des SGR geworden, das PrimeFaces zur Darstellung der Oberfläche im Webbrowser verwendet. Der Algorithmus erhält als Eingaben die im Kapitel 3.1 aufgeführten Daten. Aus den Verträgen zwischen Systemen und Service-Versionen erstellt der Algorithmus zunächst gemäß des Ablaufkonzepts in 3.2 einen Abhängigkeitsgraphen. Im Anschluss wird der erstellte Graph mittels topologischer Sortierung sortiert. Aus dem sortierten Graphen können schließlich Testperioden erstellt werden, die die tatsächlichen Testzeiten einer Service-Version darstellen. Im Folgenden werden diese Schritte des Algorithmus detailliert erläutert.

5.1 Architektur des SGR

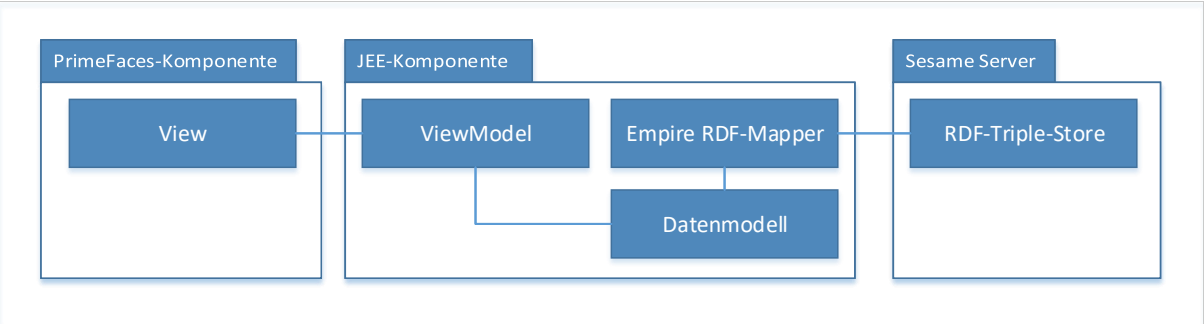


Abbildung 5.1: Komponenten der SGR-Architektur

Abbildung 5.1 zeigt eine grobe Ansicht der SGR-Architektur. Zur Datenhaltung dient der Sesame Server, der ein RDF-Repository zur Speicherung von Triple-Werten bereitstellt. Diese Triple werden vom Empire RDF-Mapper bidirektional in Java-Objekte umgewandelt. Das Datenmodell umfasst unter anderem die notwendigen Klassen zur Verwaltung von Services und Systemen. Es wurde um die Klassen in Abschnitt 5.1.1 erweitert. In der ViewModel-Komponente werden

Funktionalitäten zur Datenmanipulation implementiert. Die View-Komponente ermöglicht schließlich die browserbasierte Benutzerinteraktion.

Die Testzeitplan-Generierung wurde als Komponente in der Schichtenarchitektur [KM16] des SGR implementiert. Alle Java-Packages werden durch ihre Bezeichner eindeutig einem der drei MVVM Komponenten zugeordnet. Das Package *govRep.model.System* beinhaltet beispielsweise alle notwendigen Model-Klassen, die zur Verwaltung und Persistenz von Systemen, Verträgen und den Vertragseigenschaften nötig sind. Weiterhin existieren Packages zur Modellierung von Benutzern, Benutzerrollen, Services, Business-Objekten, etc. Die meisten Model-Klassen werden mit Hilfe des Empire RDF-Mappers[1] in einem Triplestore gespeichert und von dort auch abgerufen [Miy15].

Die View-Komponenten der Architektur werden als PrimeFaces-Benutzerschnittstellen[2] in xhtml-Dateien implementiert. PrimeFaces ist eine Open-Source-Erweiterung der JavaServer Faces. Die Datei *services.xhtml* stellt beispielsweise Ansichten und Dialoge zur Verwaltung der Services bereit. Weitere View-Komponenten existieren unter anderem für den Benutzer-Login, für die Benutzer-, Rollen-, Vertrags- und Business-Objekt-Verwaltung.

Funktionalitäten, wie das Hinzufügen von Services, werden in ViewModel-Komponenten der Architektur implementiert. Diese stellen auch Variablen zur Datenbindung mit View-Komponenten bereit. Im Package *govRep.viewModel.service* finden sich beispielsweise alle View-Model-Klassen, die zur Erstellung und Löschung von Services nötig sind. Es existieren weitere View-Model-Klassen zur Arbeit mit Rollen, Systemen, Benutzern, Business-Objekten, etc.

5.1.1 Erweiterungen

Zur Modellierung von Test-Schedules wurde das Datenmodell der Architektur um folgende Model-Klassen erweitert:

- *govRep.model.testScheduling.DependencyNode*
- *govRep.model.testScheduling.Edge*
- *govRep.model.testScheduling.Schedule*
- *govRep.model.testScheduling.ScheduleOption*
- *govRep.model.testScheduling.TestPeriod*
- *govRep.model.serviceManagement.Release*

[1] http://github.com/mhgrove/Empire
[2] http://www.primefaces.org/

Die Klassen *DependencyNode* und *Edge* werden gemeinsam verwendet, um den Abhängigkeitsgraphen von Systemen und Service-Versionen zu modellieren. Dieser besteht aus einer Adjazenzliste, bei der Objekte von *DependencyNode* ihre benachbarten Knoten über eine Liste von Edge-Objekten erreichen können. Die Klassen *Schedule* und *ScheduleOption* modellieren einen Test-Schedule mit den vom Benutzer ausgewählten Daten und Optionen. Ein *Schedule* hält nach der Testzeitplan-Generierung eine Liste von *TestPeriod*-Objekten, die die Testphasen von Service-Versionen repräsentieren. Die *Release*-Klasse modelliert ein Verbundrelease mit allen notwendigen Informationen und Service-Versionen, die Teil des Verbundreleases sind.

Des Weiteren enthält das Package *govRep.mode.testScheduling.optimizations* alle verfügbaren Optimierungen der Testzeitplan-Generierung (siehe Kapitel 4). Jede Optimierungsoption ist abgeleitet von *ScheduleOption* und implementiert seine individuelle Optimierungsmethode. Dadurch erben Optimierungen mehrere wichtige Attribute, die durch ihre Implementierung überschrieben werden können.

Die neuen View-Komponenten *configuration.xhtml* und *selectedrelease.xhtml* ermöglichen die vollständige Verwaltung von Verbundreleases. Dazu arbeiten sie mit den neuen View-Model-Komponenten im Package *govRep.viewModel.release* zusammen. Der ausgelagerte Dialog *generatescheduledialog.xhtml* wird in mehreren Views verwendet, um die Generierung eines Schedules anzustoßen.

5.1.2 Datenmodell

Das typische Vorgehen im Prototypen sieht zunächst die Erstellung und Konfiguration eines Verbundreleases vor. Zu einem solchen Release fügt der Benutzer des SOA Governance Repositories alle Service-Versionen hinzu, die eine neue Version veröffentlichen sollen. Für ein Release können dann Testing-Schedules generiert werden, die durch Testperioden realisiert werden. Diese Datenstruktur wird in Abbildung 5.2 modelliert. Alle Objekte der abgebildeten Klassenstruktur werden in einer Datenbank gespeichert.

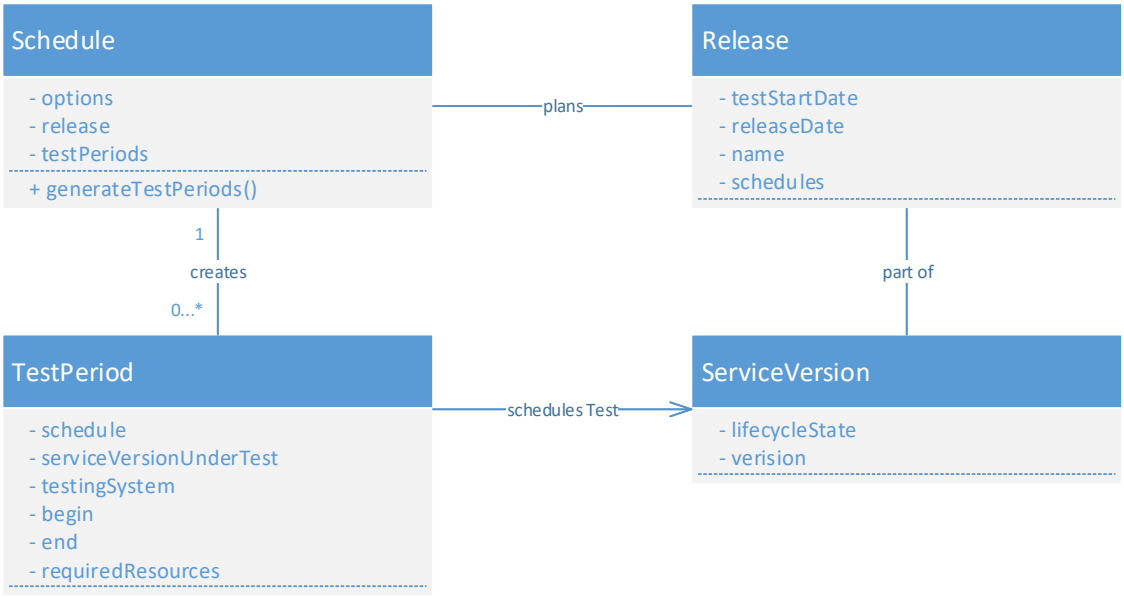


Abbildung 5.2: Model-Klassen der Implementierung

5.2 Optimierungsgerüst

Abbildung 5.3 zeigt das Gerüst der umgesetzten Optimierungen. Alle Optimierungen sind Unterklassen von *ScheduleOption* und erben demnach die abgebildeten Eigenschaften. Darüber hinaus implementiert jede Optimierung die Methode $applyToSchedule()$, in der die jeweils nötigen Optimierungsverfahren ausgeführt werden. Alle Optimierungen erben eine Standard-Nummer als *stage*. Diese Nummer gibt an, zu welcher Optimierungsphase die jeweilige Implementierung durchgeführt wird (eine nähere Betrachtung findet in Abschnitt 5.4 statt). *Expanding*, *EarlyStart* und *LateStart* können zum Ende der Generierung ausgeführt werden. Es ist einfacher, *Clipping* zu einem früheren Zeitpunkt auszuführen, damit die Änderungen am Abhängigkeitsgraphen in Effekt treten.

5.3 Oberfläche des Prototypen

In diesem Abschnitt werden Teile der entstandenen PrimeFaces-Oberfläche präsentiert.

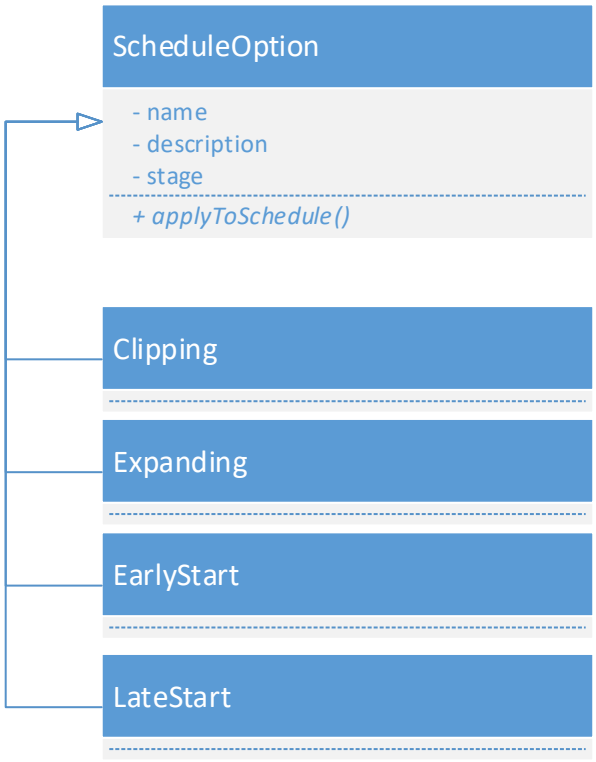


Abbildung 5.3: Klassendiagramm der Optimierungen

5.3.1 Verbundreleases

Mit Hilfe des Dialogs in Abbildung 5.4 erstellt der Benutzer eine neue Release-Instanz im System. Es muss ein Name eingegeben werden. Des Weiteren besteht die Möglichkeit, einen Text über die genaueren Absichten, Pläne und Randbedingungen in ein Beschreibungsfeld einzugeben. Über eine Kalenderdarstellung sollen jeweils ein Datum für den frühesten Starttermin der Testphasen und ein Datum für den geplanten Releasezeitpunkt ausgewählt werden. Mit dem Klick auf $Save$ wird das Objekt instanziiert und in der Datenbank abgespeichert.

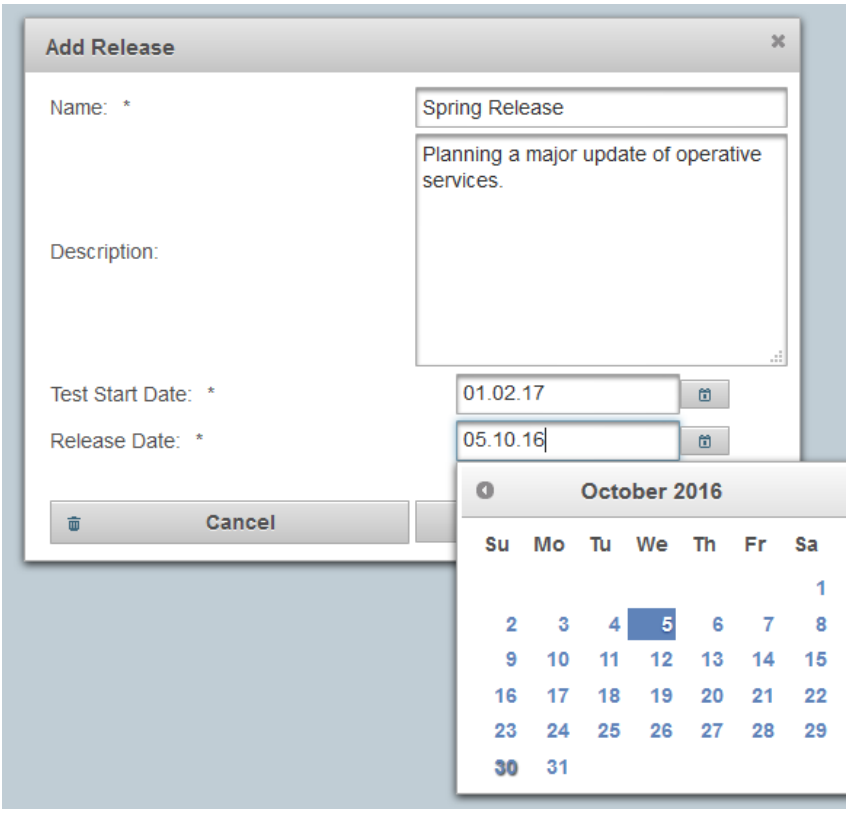


Abbildung 5.4: Dialog zum Hinzufügen eines Verbundreleases

Um Service-Versionen zu einem Verbundrelease hinzuzufügen, wählt der Benutzer in einer Liste (siehe Abbildung 5.5) die gewünschten Service-Versionen aus und fügt sie mit einem Klick auf *Add* zum Release hinzu. Ein Filter-Textfeld ermöglicht die schnelle Auswahl, wenn viele Einträge im System existieren. Eine Übersicht der Service-Versionen, die zum Releasezeitpunkt veröffentlicht werden sollen, findet sich in tabellarischer Form oberhalb des Filter-Textfeldes. Die rechte Spalte bietet Schaltflächen zum Entfernen der Einträge. Alle Änderungen in dieser Ansicht werden sofort in die Datenbank übertragen.

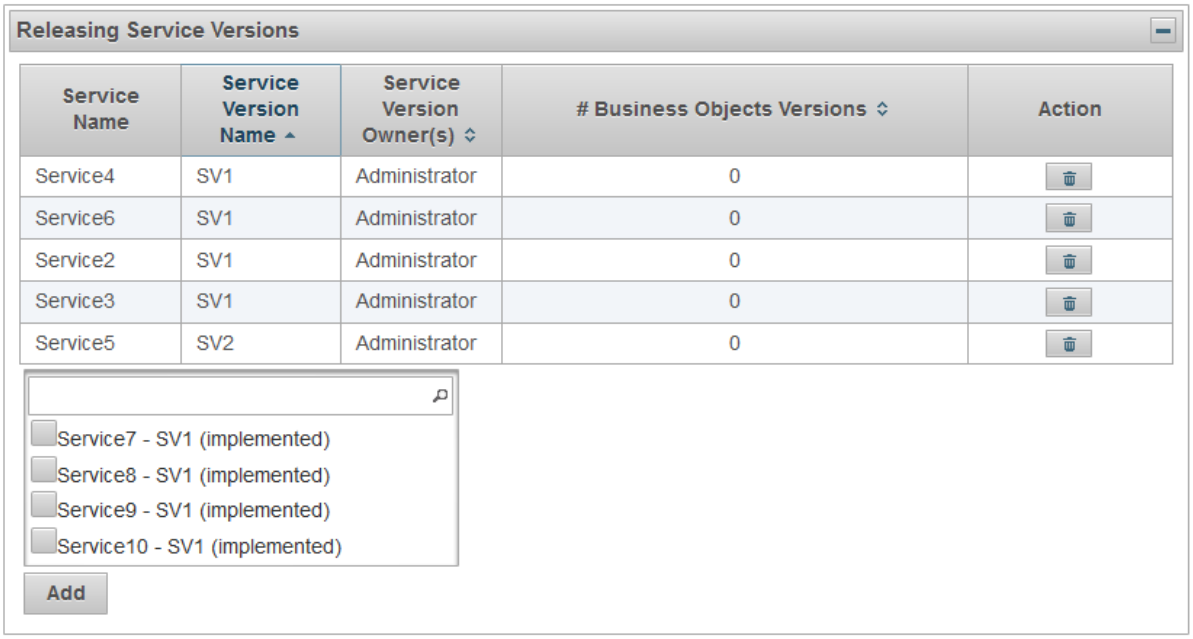


Abbildung 5.5: Auflistung von Service-Versionen, die Teil eines Verbundreleases sein sollen

5.3.2 Testzeitpläne

In diesem Abschnitt wird die Interaktion und Visualisierung der Test-Schedules vorgestellt. Abbildung 5.6 zeigt den Dialog, der den Algorithmus zur Testzeitplan-Generierung startet. Als einzige Pflichteingabe soll hier ein Name für den Zeitplan vergeben werden. Dieser dient später zur Unterscheidung mehrerer Zeitpläne. Darüber hinaus werden in diesem Dialog alle gewünschten Optimierungen ausgewählt. Die gewählten Optimierungen werden mit Klick auf *Generate* instanziiert und in einer Liste an den Algorithmus übergeben.

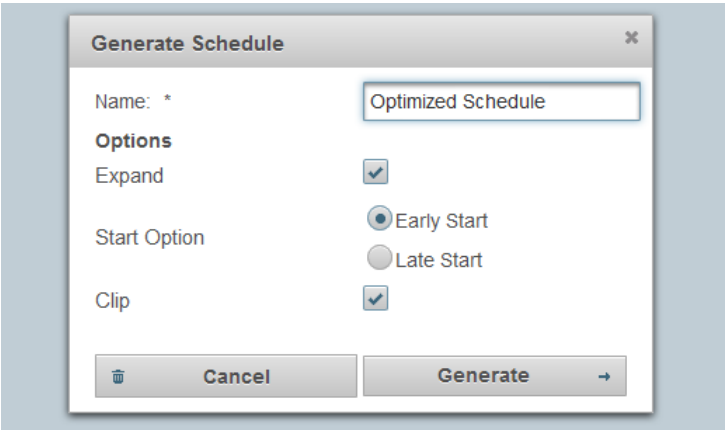


Abbildung 5.6: Dialog zum Generieren von Testzeitplänen

Ist der Vorgang zur Generierung beendet, zeigt ein Hinweisfenster an, ob der zeitliche Rahmen für den generierten Zeitplan eingehalten werden konnte. Die Abbildung 5.7 zeigt, dass die Beispiel-Ausführung einen Zeitplan generiert hat, der über den Releasezeitpunkt hinaus läuft. Die Deadline kann in diesem Fall nicht eingehalten werden.

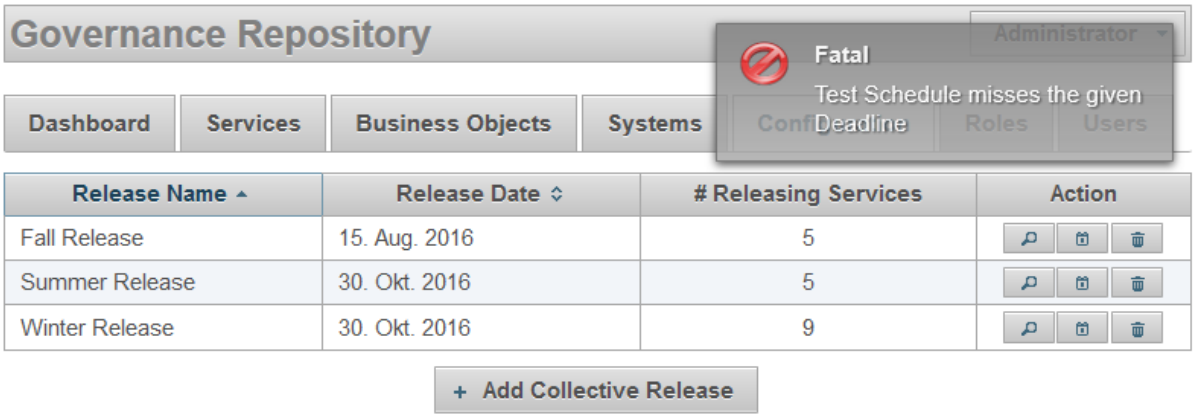


Abbildung 5.7: Fehlermeldung beim Verpassen des Releasezeitpunktes

Abbildung 5.8 zeigt außer dem Standard-Navigationsmenü zwei übereinander angeordnete Container. Der obere zeigt die Stammdaten des gewählten Releases an. Die abgebildeten Informationen können in diesem Bereich editiert und gespeichert werden. Der untere Container zeigt die Visualisierung eines Zeitplans. Der Container wird nur dann angezeigt, wenn zum gewählten Release bereits ein Test Schedule generiert wurde. In diesem Fall endet der Zeitplan zum 21.09.2016 und erfüllt damit die Deadline mit dem Releasezeitpunkt am 31.10.2016. Ein gelbes Hinweisfenster zeigt an, wenn sich Abhängigkeiten oder sonstige Daten, die eine Auswirkung auf den Zeitplan haben können, geändert haben. In diesem Fall wurde ein neuer Vertrag mit einer dem Release zugeordneten Service-Version abgeschlossen. Deswegen sollte eine erneute Generierung des Zeitplans angestoßen werden. Ein horizontales Balkendiagramm zeigt die Beteiligten einer Testphase über dem jeweiligen Datum an. Ein Tooltip, der beim *MouseOver* über einem Balken angezeigt wird, zeigt das genaue Start- und Enddatum der jeweiligen Testphase an.

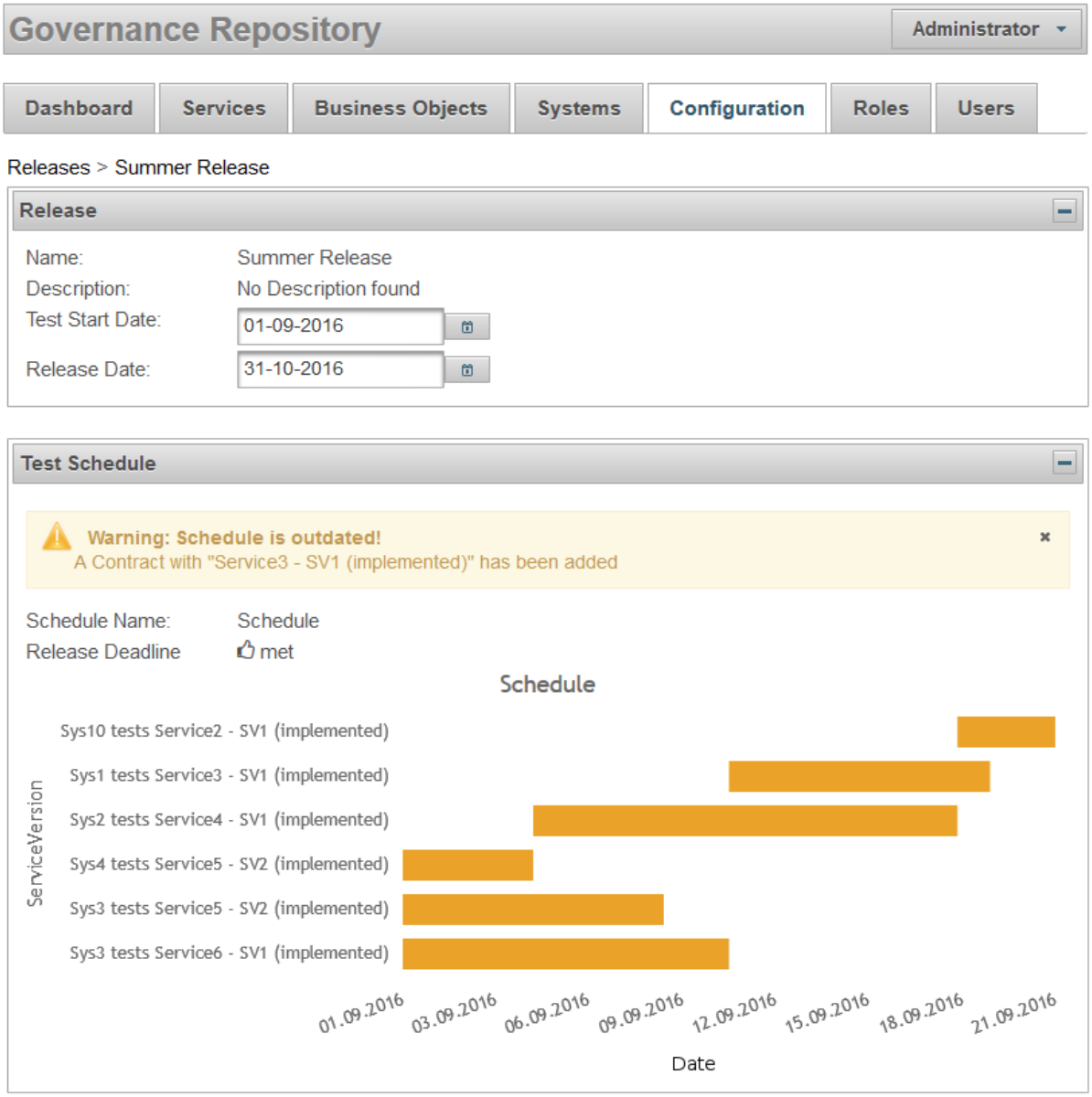


Abbildung 5.8: Darstellung eines generierten Testzeitplans

5.4 Optimierungszeitpunkt

Beim Start der Generierung werden alle im Dialog (siehe Abb. 5.6) ausgewählten Optimierungen instanziiert. Alle ausgewählten Instanzen werden in der folgenden Liste (Listing 5.1) gehalten. Diese wird als entsprechendes *RdfProperty* annotiert, wodurch sie als Teil eines Testzeitplans in der RDF-Datenbank gespeichert werden kann.

Listing 5.1: Liste der auf einen Testzeitplan anzuwendenden Optimierungen

```
/** Holds and persists the chosen optimizations to be applied during
    generation of the schedule
*/
@OneToMany(fetch = FetchType.EAGER)
@RdfProperty("sgr:scheduleOptions")
public List<ScheduleOption> scheduleOptions = new
    ArrayList<ScheduleOption>();
```

Der Zeitpunkt, wann eine Optimierung durchgeführt wird, hängt von der *stage*-Variable ab, die von der Klasse *ScheduleOption* mit dem Standardwert *10* vererbt wird. Im Code der Generierung können an beliebigen Stellen Optimierungen mit einer eigenen *stage*-Nummer durchgeführt werden. Der Wertebereich dieser Variable beträgt $W_{stage} = \{n \in \mathbb{N} \mid 0 < n < 11\}$. Der Wert *stage = 10* entspricht dabei der Ausführung nach Ende der Testperiodengenerierung. Optimierungen wie *Clipping*, die Änderungen am Abhängigkeitsgraphen durchführen sollen, überschreiben die vererbte *stage*-Variable in ihrem Konstruktor mit einem kleineren Wert, um vor der Testperiodengenerierung ausgeführt zu werden. Der Code-Auszug 5.2 zeigt die Ausführung der Optimierungen zu einem bestimmten Zeitpunkt.

Listing 5.2: Anwendung von Optimierungen auf einen Testzeitplan

```
/**
* This method runs the optimizations which correspond to the provided
    stage number.
* @param stage
*       current stage of the algorithm
*/
private void optimizeSchedule(int stage) {
        for (ScheduleOption option : this.scheduleOptions){
                if(stage == option.getStage()){
                        System.out.println("Applying Option: " +
                            option.getName());
                        option.applyToSchedule((this));
                }
        }
}
```

5.5 Implementierung der Optimierungen

Dieser Abschnitt zeigt auf wie die implementierten Optimierungen im SGR umgesetzt wurden. Dabei werden die Kernfunktionalitäten anhand von Java-Code-Abschnitten erklärt.

5.5.1 Testperioden expandieren

Das Expandieren von Testperioden maximiert die Testdauer von Testperioden auf den größtmöglichen Wert. Dazu werden die bereits instanziierten Testperioden abgeändert. Nach Beendigung der Testperiodengenerierung ohne Optimierungen entspricht die Dauer einer Testperiode immer dem im Vertrag angegebenen Testaufwand. Der Testaufwand wird in Stunden eingegeben, woraus während der Generierung die benötigten Tage berechnet werden. Dafür sind acht Personenstunden pro Tag vorgesehen. Ein beispielhafter Testaufwand von *50* Stunden wird folglich zu *6,25* Tagen. Dieser Wert wird schließlich auf ganze Tage gerundet, damit die Berechnung der Kalendertage nicht unnötig verkompliziert wird. Sollte im Vertrag mit einer Service-Version kein Testaufwand hinterlegt sein, wird ein Standardwert von sieben Tagen als voreingestellte Testdauer verwendet. Testen mehrere Konsumenten dieselbe Service-Version eines Services, können die Endpunkte der Testperioden auf den spätesten beteiligten Endpunkt gesetzt werden. Das ist möglich, da alle weiterführenden Tests in der Testreihenfolge (siehe Abschnitt 3.2.1 und 3.2.2) eine höhere Rangnummer erhalten und somit auf das Ende der vorhergehenden Testperiode warten müssen. Die Implementierung durchsucht alle Testperioden auf das späteste Testende *max* einer Service-Version. Anderen Testperioden, die dieselbe Service-Version eines Services testen, wird ebenfalls *max* als Testende zugewiesen. Diese Änderungen müssen auch in die Datenbank übertragen werden. Dafür wird die *update*-Methode des Testperioden-Objekts ausgeführt. Das folgende Code-Fragment 5.3 zeigt die Zuweisung und Aktualisierung einer Testperiode *p* mit dem zuvor ermittelten End-Datum *max*.

Listing 5.3: Ausdehnung von Testperioden

```
//setting all periods of the service version to max end date.
for (TestPeriod p : schedule.getTestPeriods()){
        if(period.getServiceVersion() == p.getServiceVersion()){
                p.setEnd(max);
                // update testperiod which is already in DB
                HttpSession session = (HttpSession)
                   FacesContext.getCurrentInstance().getExternalContext()
                        .getSession(false);
                User operator;
                try {
                        operator = Authentication.getAuthenticator()
                                .getOperator(session);
                        p.update(operator);
                }catch(Exception e){
                        e.printStackTrace();
                }
        }
}
```

5.5.2 Früher Teststart

Ein früher Teststart bedeutet, dass der Testzeitplan zu einem benutzerdefinierten Startdatum beginnt. Dieses Datum gibt der Benutzer beim Anlegen eines Releases (siehe Abschnitt 5.3.1) ein. Die beiden Einstellungen *früher Teststart* und *spätes Testende* (siehe Abschnitt 5.5.3) schließen sich exklusiv aus. Die Implementierung der hier vorgestellten Variante wird als Normalfall angesehen und ist deshalb Teil des Generierungsprozesses und nicht als Optimierung implementiert. Die hier vorgestellte Vorgehensweise wird folglich bei jeder Generierung ausgeführt.

Im Laufe der Ausführung werden den Kanten im Abhängigkeitsgraphen Start- und Enddaten zugewiesen. Eine Kante repräsentiert einen Vertrag zwischen einem System und einer Service-Version. Diese Interaktion gilt es in einer Testperiode zu testen. Zunächst werden alle ausgehenden Kanten von Knoten mit Rangnummer *0* mit dem vorgegebenen Startdatum initialisiert. Das Enddatum ergibt sich durch die Addition von Startdatum und der berechneten Testdauer (siehe Abschnitt 5.5.1). Die Klasse *java.util.Calendar* erleichtert das Addieren von Tagen zu einem angegebenen Datum. Der nachfolgende Code-Auszug 5.4 zeigt die Methode zur Berechnung aller fortlaufenden Daten. Alle Verträge, die von Knoten mit Rangnummer *0* ausgehen, können unmittelbar nach dem globalen Startdatum ihre Testarbeit aufnehmen, da diese Systeme keine weiteren Service-Versionen testen müssen. Nachdem diese Initialisierung vorgenommen wurde, werden alle weiteren Knoten des Graphen gemäß ihrer Rangnummer aufsteigend abgearbeitet. Für alle weiteren Knoten werden alle ihre Kanten durchlaufen. Das Startdatum, das einer Kante zugewiesen wird, ist das Enddatum des aktuellen Knotens. Das Enddatum ist wiederum die Summe aus Startdatum und Testaufwand. Für jede Kante wird dabei ein Objekt der Klasse *TestPeriod* instanziiert und in der Datenbank erstellt. Bevor eine Testperiode in die Datenbank gespeichert wird, werden Start- und Enddatum der Testperiode gesetzt. Außerdem muss die, während der Testperiode zu testende Service-Version und das testende System gespeichert werden, um den Kontext der Testperiode beizubehalten.

Listing 5.4: Addition des Testaufwands zum gegebenen Kalenderdatum

```
/**
* Adds days to the provided startDate. Each day consists of 8 working hours
*
* @param startDate
*   the base date
* @param testEffort
*   hours of work to do
* @return date
*       date after the working hours
*/
private Date addTestEffort(Date startDate, float testEffort) {
        Calendar cal = Calendar.getInstance();
        cal.setTime(startDate);
```

```
        // add the required amount of testing days, rounded to ceiling
        if (testEffort == 0.0f) {
                // use default test effort value (7 days)
                cal.add(Calendar.DATE, 56 / 8);
        } else {
                cal.add(Calendar.DATE, (int) Math.ceil((double) testEffort
                   / 8));
        }
        return cal.getTime();
}
```

5.5.3 Spätes Testende

Ein Testzeitplan mit einem späten Testende sieht die Beendigung der Testaktivitäten einen Tag vor dem Releasedatum vor. Dafür werden alle bestehenden Testperioden gleichermaßen verschoben und in der Datenbank aktualisiert. Wenn die Optimierung vom Benutzer gewählt wurde, wird sie am Ende der Testzeitplan-Generierung ausgeführt. Im ersten Schritt wird die Differenz ermittelt, um die alle Perioden verschoben werden müssen. Dabei handelt es sich um die Differenz zwischen dem spätesten Enddatum aller Testperioden und dem Releasedatum. Im Anschluss werden die Daten aller Testperioden mit der in Abschnitt 5.5.2 vorgestellten Methode verschoben. Der folgende Code-Auszug 5.5 ermittelt die Testperiode *last* mit dem spätesten Enddatum, die zur Bestimmung der Differenz zum Releasedatum verwendet wird.

Listing 5.5: Suche nach dem spätesten Ende einer Testperiode

```
//find last ending period
TestPeriod last = schedule.getTestPeriods().get(0);
for(TestPeriod p : schedule.getTestPeriods()){
        if(p.getEnd().getTime() > last.getEnd().getTime()){
                last = p;
        }
}
```

5.5.4 Teilen des kritischen Pfads

Das Teilen des kritischen Pfads soll zu einer Verkürzung der Gesamtdauer eines Testzeitplans führen. Dazu nimmt die Optimierung Änderungen am Abhängigkeitsgraphen vor, bevor aus dessen Struktur die Testperioden generiert werden. Der exakte Ausführungszeitpunkt ist nachdem die Knoten-Kanten-Struktur des Graphen als Adjazenzliste aufgebaut wurde (nach Schritt 2 in 3.1). Die Implementierung der Optimierung besteht aus drei Phasen:

1. Bestimmung des kritischen Pfads

2. Vergleich mit anderen Pfaden bzw. Entscheidung über die Anwendung der Optimierung
3. Teilen des Pfads und Duplizieren der Schnittstelle

Die folgenden Abschnitte erklären, auf welche Weise und in welchem Maße das Konzept aus Abschnitt 4.4 realisiert wurde.

Finden des kritischen Pfads

Im ersten Schritt der Optimierung wird der Abhängigkeitsgraph mittels rekursiver Tiefensuche (Listing 5.6) durchlaufen, um den kritischen Pfad zu finden. Zum Zeitpunkt der Ausführung dieser Optimierung stehen noch keine Sortierinformationen bereit. Um die Tiefensuche von den oberen Knoten des Graphen starten zu können, werden zunächst die Wurzelknoten gesucht. Da der kritische Pfad von jedem Wurzelknoten ausgehen kann, wird von jedem Wurzelknoten eine Tiefensuche gestartet. Insbesondere ist es auch möglich, dass die Struktur mehrere disjunkte Teilgraphen aufweist. Dadurch wird schnell klar, dass ausgehend von allen Wurzelknoten nach dem kritischen Pfad gesucht werden muss. Dabei werden alle besuchten Kanten eines Pfads in einer Liste abgelegt. Während der Tiefensuche wird der längste gefundene Pfad in einer separaten Variable *criticalPath* gehalten. Jede Rekursion der Tiefensuche aktualisiert den durchlaufenen Pfad *currentPath* und vergleicht diesen mit dem bisher gefundenen längsten Pfad. Der Vergleich zweier Pfade summiert die Testaufwände beider Pfade auf und vergleicht die beiden Summen. Je nach Ergebnis gibt die Vergleichsmethode *-1, 0* oder *1* zurück. Am Ende der Tiefensuche liegt der Pfad mit dem größten Gesamt-Testaufwand als Ergebnis vor. Die folgende Java-Funktion (Listing 5.6) zeigt die rekursive Implementierung der Tiefensuche mit der Suche nach dem kritischen Pfad.

Listing 5.6: Tiefensuche zur Ermittlung des kritischen Pfads

```
/**
* Recursive DFS function
*
* @param currentEdge
*       the edge, that is currently visited by the DFS
* @param currentPath
*       the list of edges, which are part of the current path
*/
private void visit(Edge currentEdge, List<Edge> currentPath){
        currentPath.add(currentEdge);
        switch(comparePaths(currentPath, criticalPath)){
                case -1: // current Path is shorter than criticalPath.
                        break;
                case 0 : // current Path is equal to the criticalPath.
                        break;
```

```
                case 1 : // current Path is greater than the criticalPath.
                    update criticalPath
                        criticalPath = currentPath;
        }

        // for all edges continue DFS
        if(currentEdge.getTargetNode().getEdges() != null){
                for(Edge edge : currentEdge.getTargetNode().getEdges()){
                        visit(edge, currentPath);
                }
        }else{
                pathDepths.add(currentPath.size());
                return;
        }
}
```

Entscheidung über die Anwendung

Die Teilung des kritischen Pfads macht nur dann Sinn, wenn dadurch tatsächlich eine Verkürzung der Gesamtdauer eines Testzeitplans erreicht wird. Die dazu notwendigen Voraussetzungen wurden bereits in Abschnitt 4.4.1 vorgestellt. Je mehr Testaufwand der kritische Pfad im Vergleich zu allen anderen Pfaden umfasst, desto größer ist der Vorteil, den die Anwendung der Optimierung erzielt. Der exakte zeitliche Gewinn der Optimierung besteht in der Differenz beider kritischer Pfade, die der Graph vor und nach der Optimierung aufweist. Die prototypische Implementierung führt die Optimierung genau dann aus, wenn der kritische Pfad mehr als doppelt so lang ist wie der zweitlängste Pfad im Graphen. Dazu werden die Längen aller Pfade während der Tiefensuche in der Liste *pathDepths* gespeichert.

Durchführung der Teilung

Wurde die optimale Trennstelle für die Teilung gefunden, so wird die betroffene Kante aus dem Graphen zunächst entfernt. Um die Testbeziehung der beiden betroffenen Komponenten nicht zu vernachlässigen, wird der betroffene Knoten dupliziert und an das Ende der Testperioden angehängt. Der folgende Code-Auszug 5.7 zeigt, wie die Teilung des Graphen in Java umgesetzt wurde.

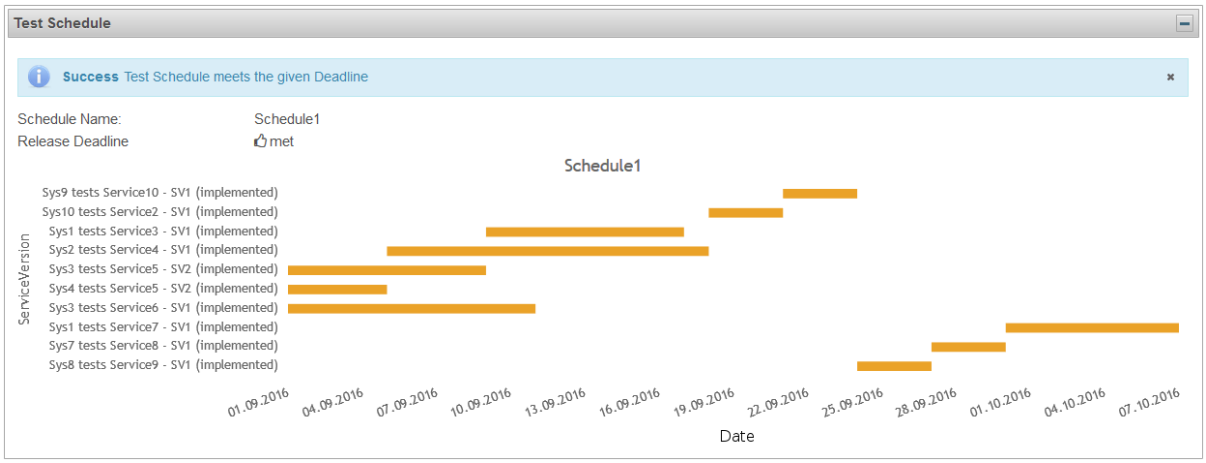
Listing 5.7: Teilung des ermittelten Pfads

```
if(currentNode.getEdges().remove(edgeToRemove)){
        // duplicate the bottom node from the critical path's first half
           and attach it to the top node of the critical path's second half
```

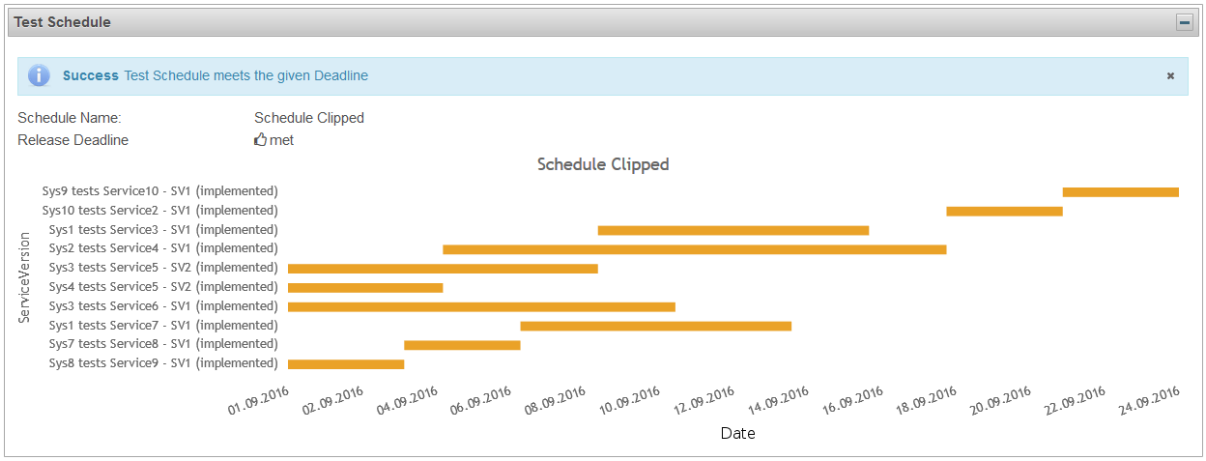
```
        DependencyNode bottomNode = criticalPath.get(criticalPath.size() /
            2).getTargetNode();
        newNode = new DependencyNode();
        newNode.setServiceVersion(bottomNode.getServiceVersion());
        newNode.setSystem(bottomNode.getSystem());

        Edge edge = new Edge();
        edge.setTargetNode(newNode);
        edge.setContract(edgeToRemove.getContract());
        criticalPath.get(criticalPath.size() /
            2-1).getTargetNode().addEdge(edge);
}
```

Die Abbildung 5.9a zeigt einen Testzeitplan, der ohne jegliche Optimierungen generiert wurde. Das beispielhafte Szenario umfasst neun Service-Versionen und acht Systeme. Auf Grund ihrer Nutzungsverträge ergibt sich in diesem Fall ein hochgradig serieller Testzeitplan. Sieben der Testperioden sind direkt voneinander abhängig. Der gesamte Testzeitraum soll vom *01.09.2016* bis zum *07.10.2016* dauern. Abbildung 5.9b zeigt dasselbe Szenario mit der Anwendung der vorgestellten Optimierung. Das Enddatum des Testzeitplans kann dadurch vom *07.10.2016* auf den *24.09.2016* verbessert werden. Durch die parallele Ausführung der oberen und unteren Hälfte des kritischen Pfads konnte der Testzeitplan um *13* Tage verkürzt werden.



(a) Hochgradig serieller Testzeitplan ohne Clipping



(b) Gleiche Konfiguration mit Clipping

Abbildung 5.9: Vergleich zweier Testzeitpläne ohne und mit Optimierungsausführung

5.6 Komplexität der Optimierungen

Die folgende Tabelle bietet eine Übersicht über die Laufzeit-Komplexität der implementierten Optimierungen. Damit kann die Skalierbarkeit der Konzepte eingeschätzt werden.

Optimierung	Komplexität	Begründung
Testperioden expandieren (Abschnitt 5.5.1)	$O(\|P\|)$	Das Verfahren durchläuft einmal alle Testperioden P.
Früher Teststart (Abschnitt 5.5.2)	$O(\|V\| + \|E\|)$	Zur Vergabe des Start- und Endzeitpunkts von Testperioden werden alle Knoten V und Kanten E im Abhängigkeitsgraphen ausgewertet.
Spätes Testende (Abschnitt 5.5.3)	$O(\|P\|)$	Die zeitliche Verschiebung betrifft alle Testperioden P.
Teilen des kritischen Pfads (Abschnitt 5.5.4)	$O(\|V\| + \|E\|)$	Die Tiefensuche durchläuft die Menge V aller Knoten und E aller Kanten im Abhängigkeitsgraphen.

Tabelle 5.1: Komplexität der implementierten Optimierungen

6 Fallstudie

In diesem Kapitel wird die durchgeführte Fallstudie vorgestellt. Ziel ist es, die Leistungsfähigkeit des Prototypen bei der Anwendung mit einem großen und realitätsnahen Datensatz zu prüfen. Können kleine Testzeitpläne noch ohne Systemunterstützung auf Papier entworfen werden, soll der Prototyp die Anwendung bei einem großen Verbundrelease zeigen. Als maßgeblich für die Komplexität der Erstellung des Zeitplans gelten die Anzahl der beteiligten Systeme, Services, Service-Versionen und ganz besonders die Verträge der Systeme mit Service-Versionen. Im folgenden Anwendungsfall soll ein Testingenieur die Testausführungen für ein Verbundrelease planen. Die komplexe Struktur der vorliegenden SOA (siehe Abschnitte 6.1 und 6.1.1) erfordert die Hilfe der automatisierten Testzeitplan-Generierung. Die manuelle Erstellung des Zeitplans birgt ein hohes Fehlerrisiko und dauert zu lange. Der Testingenieur erstellt im SGR ein Verbundrelease mit den beiden Daten, die den Testzeitraum eingrenzen sollen. Im nächsten Schritt fügt er Service-Versionen zum Verbundrelease hinzu, die getestet werden sollen. Als weiteren Schritt startet er die Testzeitplan-Generierung mit allen verfügbaren Optimierungen. Nachdem die Generierung abgeschlossen ist, müssen folgende Fragen beantwortet werden:

- Hält der Testzeitplan die geforderten Daten ein?
- Ist der Testzeitplan umsetzbar?
- Soll der Testzeitplan verwendet werden?

Die Auswertung dieser Fragen erfolgt in Abschnitt 6.3.

6.1 Verwendung eines realen Datensatzes

Der verwendete Datensatz entstammt der SOA-Umgebung eines Automobilherstellers. Die Daten wurden als Excel-Datei aus dem aktuellen Service-Repository des Herstellers exportiert und anonymisiert zur Verfügung gestellt. Die bereitgestellte Architektur umfasst insgesamt *82* Systeme. Zu jedem System gibt es eine oder zwei verantwortliche Personen. Außerdem kann es einen separaten Service-Verantwortlichen geben. Systeme stellen bis zu *17* verschiedene Services und *24* Service-Versionen bereit. Ein System konsumiert bis zu *16* Service-Versionen. Schätzungen zum Testaufwand stehen nicht zur Verfügung. Aus diesem Grund werden für den Testaufwand Zufallswerte zwischen drei und zwölf Personen-Tagen zu jeweils acht Arbeitsstunden angenommen. 147 Service-Versionen existieren in dieser SOA. Bis auf zehn, befinden

sich alle im Lebenszyklus-Zustand *Veröffentlicht*. Neun weitere gelten als *Veraltet* und eine als *Angekündigt*.

6.1.1 Gewähltes Verbundrelease

Ungeachtet ihres Lebenszyklus-Zustands wurden zufällig einige Service-Versionen als Teil eines Verbundreleases ausgewählt. Von den 147 verfügbaren Service-Versionen wurden 21 in ein Verbundrelease aufgenommen. Das entspricht 14,29 % der Service-Versionen, die im Szenario eine neue Version veröffentlichen sollen. In Abbildung 6.1 werden die vereinfachten Abhängigkeiten zwischen Systemen und ausgewählten Service-Versionen dargestellt. Sie bilden sieben unabhängige Teilgraphen, die während der Testzeitplan-Generierung verarbeitet werden. Die Teilgraphen sind in Abbildung 6.1 farblich voneinander abgetrennt.

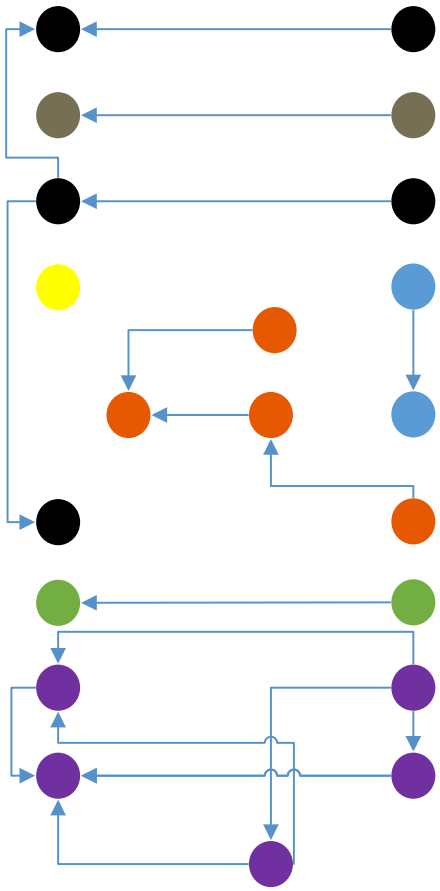


Abbildung 6.1: Vereinfachte und anonymisierte Darstellung der Abhängigkeiten zwischen den beteiligten Systemen und Service-Versionen

6.2 Generierung des Testzeitplans

Die Generierung eines Testzeitplans führt unter Verwendung von zufälligem Testaufwand zu einem Testzeitplan wie in Abbildung 6.2 dargestellt.

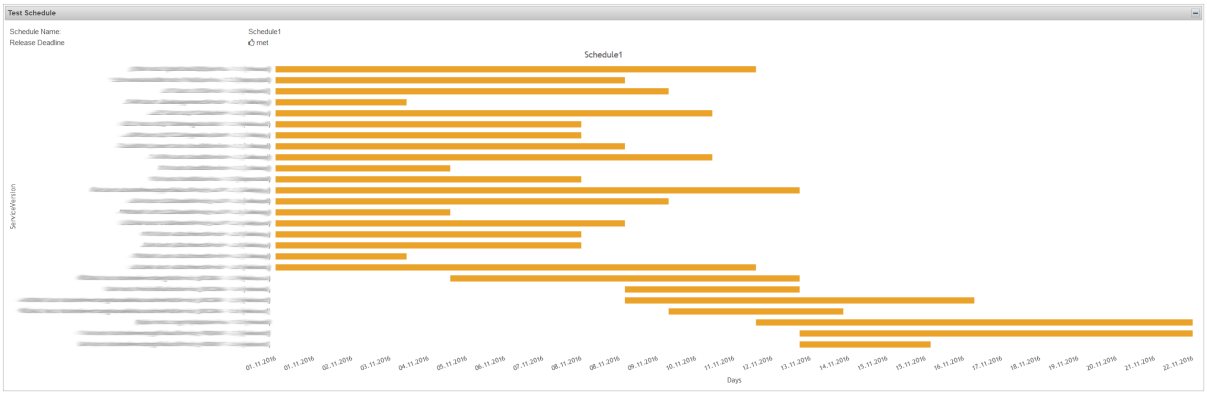


Abbildung 6.2: Generierter Testzeitplan unter Verwendung randomisierter Testdauern

6.2.1 Angewandte Optimierungen

Das Expandieren der Testperioden erzielt bei jeder Ausführung unterschiedliche Ergebnisse, da der Testaufwand jedes Mal zufällig gewählt wird. Im Abhängigkeitsgraphen in Abbildung 6.1 tritt das Muster, bei dem die Optimierung effektiv ausgeführt wird, insgesamt vier Mal auf. Dabei handelt es sich um Knoten, die mehr als eine eingehende Kante aufweisen. Im umgekehrten Abhängigkeitsgraphen wird schnell klar, dass bei diesem Muster mehrere Systeme die gleiche Service-Version testen. Immer dann können Testperioden maximiert werden.

Auch das *LateStart*-Verfahren unterscheidet sich auf Grund des zufällig generierten Testaufwands in seiner Ausführung. Die Anwendung auf den in Abbildung 6.2 dargestellten Testzeitplan verschiebt alle Testperioden um *sieben* Tage in die Zukunft. Das Ende des Testzeitplans wird somit vom *22.11.2016* auf den *29.11.2016*, also einen Tag vor den Releasezeitpunkt am *30.11.2016* verschoben.

Das Verfahren zur Teilung des kritischen Pfades führt die Tiefensuche im Graph aus. Aus der Anordnung der Knoten in Abbildung 6.1 und der Testperioden in Abbildung 6.2 ist ersichtlich, dass der Baum keinen hervorstechenden, besonders langen Pfad birgt. Die Wahrscheinlichkeit, dass durch die Trennung des kritischen Pfades eine Verbesserung erzielt werden kann, ist deshalb gering und hängt ebenfalls vom zugewiesenen Testaufwand ab.

6.2.2 Performance-Messungen

Zur Zeitmessung wurden zwischen den Generierungsschritten Code-Segmente zur Subtraktion der gemessenen Zeitwerte eingefügt. Außerdem wurden die Ergebnisse direkt in eine Textdatei geschrieben. Tabelle 6.1 zeigt die verwendete Konfiguration der Maschine, welche die Tests ausgeführte.

Komponente	Wert
Prozessor	Intel i7-2600 @ 3.40 GHz
Arbeitsspeicher	8 GB DDR3 - 1333 MHz
Betriebssystem	Windows 10 64-Bit
Java VM	Version 1.8 32-Bit
OpenRDF Sesame	Version 4.1.2
Sesame Repository Typ	Java Native Store

Tabelle 6.1: Zur Testausführung genutzte Hard- und Software

Die Testzeitplan-Generierung wurde fünf mal ausgeführt. Dafür wurden die drei Optimierungen *Expanding, Clipping* und *LateStart* aktiviert. Das Verfahren für *EarlyStart* muss nicht separat gemessen werden, da es bereits als Standard-Variante nach der Generierung gilt. Somit benötigt die Ausführung keine zusätzliche Zeit, wenn die Option gewählt wird. Tabelle 6.2 zeigt die Mittelwerte der gemessenen Zeiten aller fünf Ausführungen.

Vorgang	Dauer	Datenbankzugriff
Graph-Sortierung	0,29 ms	nein
Erstellung der Testperioden	286,59 ms	ja
Graph-Ausgabe in die Konsole	1,06 ms	nein
Gesamtdauer der Generierung ohne Optimierungen	286,89 ms	ja
Expandieren der Testperioden	2.930,00 ms	ja
Später Teststart	3.012,89 ms	ja
Teilen des kritischen Pfades	0,52 ms	nein
Gesamtdauer mit Optimierungen	**6.231,36 ms**	**ja**

Tabelle 6.2: Zeitmessungen der Ausführung

Die Gesamtdauer der Generierung setzt sich aus den ersten beiden Werten der Tabelle zusammen. Maßgeblich für die Gesamtdauer des Generierungsprozesses ist die Dauer für die Erstellung der Testperioden mit 286 ms. Wesentlich länger dauern die Optimierungen, die alle Testperioden in der Datenbank abändern müssen (*Expanding* und *LateStart*). Das Teilen des kritischen Pfades dauert mit 0,52 ms nur sehr kurz. Die ausgeführte Tiefensuche findet in diesem Szenario keinen ungewöhnlich langen Pfad im Baum und beendet die Optimierung

relativ schnell. Außerdem führt diese Optimierung keine Änderungen im Datenbanksystem aus.

6.3 Auswertung und Bewertung des generierten Testzeitplans

Das erste Indiz über die Umsetzbarkeit des generierten Testzeitplans ist die Tatsache, ob ein Testzeitplan das vorgegebene Releasedatum einhalten kann oder nicht. Kann das Releasedatum vom Testzeitplan nicht eingehalten werden, kann der Testzeitraum, Testumfang oder der Testaufwand abgeändert werden. Die Beispielausführung in Abbildung 6.2 sieht das Ende des Testzeitraums am *22.11.2016* vor und hält damit den Releasezeitpunkt vom *30.11.2016* ein. Des Weiteren liegt es im Ermessen des Testingenieurs, ob die Ressourcen zur Testausführung aufgebracht werden können. Der generierte Testzeitplan sieht zu Beginn des Testzeitraums eine Vielzahl von Testperioden zur gleichen Zeit vor. Daraus ergibt sich ein besonders großer Bedarf an Ressourcen zu Beginn der Testaktivitäten.

7 Zusammenfassung und Ausblick

In diesem Kapitel werden die wichtigsten Gesichtspunkte der Arbeit noch einmal zusammengefasst. Die anschließenden Erweiterungsmöglichkeiten zeigen Gebiete auf, die in zukünftige Arbeiten aufgenommen werden können. Außerdem fasst ein Fazit einige der gewonnenen Erkenntnisse kurz zusammen.

7.1 Zusammenfassung

In Kapitel 2 wurden besondere Charakteristika einer SOA besprochen. Besonders die dynamische und anpassungsfähige Natur einer SOA machen Tests einer solchen zu einer großen Herausforderung. Um das Verständnis aller Beteiligten im Softwareprozess beim Umgang mit einer SOA zu leiten, wird die Notwendigkeit des SOA-Governance-Konzepts deutlich. Die Einführung einer solchen Architektur umfasst wesentlich mehr als die Implementierung der Geschäftslogik in Web Services. Das SOA Governance Repository aus Abschnitt 2.5 ist eine zentrale Verwaltungsstelle zur Steuerung aller teilhabenden Gesichtspunkte. Der Prototyp leitet Akteure mit einer modernen Oberfläche und einem flexiblen Backend beispielsweise beim Life Cycle Management, der Verwaltung von Service-Versionen und der Vernetzung aller beteiligten Stakeholder. In Kapitel 3 wird die automatisierte Generierung von Testzeitplänen einer SOA konzipiert. Die generierbaren Zeitpläne gelten als notwendige Grundlage zur gut koordinierten Durchführung der Integrationstests einer SOA. Dafür werden Systeme, Services und Service-Versionen in einem Abhängigkeitsgraphen abgebildet, dessen Sortierung schließlich die Testreihenfolge der interagierenden Komponenten bestimmt. Auf diese Weise garantiert der Algorithmus, dass bei korrekter terminlicher Einhaltung, alle Abhängigkeiten einer Komponente schon vor seiner Testzeit getestet wurden. Im weiteren Verlauf der Arbeit (Kapitel 4) werden Optimierungen erarbeitet, die den Testzeitplan für bestimmte Anforderungen anpassen können. Das Expandieren von Testperioden sorgt dafür, dass eine Testperiode den maximal verfügbaren Zeitraum zugewiesen bekommt. Das Teilen des kritischen Testpfads kann hingegen die gesamte Dauer des Testzeitplans verkürzen, indem die Nebenläufigkeit der Testperioden verstärkt wird. Kapitel 5 stellt die konkrete Umsetzung der Konzepte als Teil des SOA Governance Repositories vor. Dafür waren Anpassungen in vielen Bereichen des Prototyps notwendig. Beispielsweise wurde die Dateneingabe des Testaufwands und die Zuordnung eines Services zu einem System umgesetzt. Im MVVM-Muster wurde die Verwaltung von Verbundreleases, die Generierung und Darstellung der Testzeitpläne implementiert. Des

Weiteren wird der implementierte Prototyp in Kapitel 6 mit realen Testdaten geprüft. Dabei stehen die Aspekte der automatisierten Testzeitplan-Generierung im Vordergrund.

7.2 Erweiterungsmöglichkeiten

Dieser Abschnitt stellt vier Erweiterungsmöglichkeiten der Testzeitplan-Generierung vor.

7.2.1 Generische Testzeitplan-Generierung als eigenständiger Service

Die vielseitigen Einsatzmöglichkeiten für den Algorithmus (siehe Abschnitt 7.3.1) legen nahe, eine generische Version dessen zu implementieren. Dabei sollten keine SOA-spezifischen, sondern möglichst generische Begriffe und Datenmodelle verwendet werden. Außerdem würde es sich anbieten, eine solche Implementierung als Web Service bereit zu stellen, um mit einer einheitlichen und klar definierten Schnittstelle den vielseitigen Einsatzdomänen gerecht zu werden.

7.2.2 Standardwert für die Testleistung eliminieren

Die Umrechnung vom Testaufwand in Stunden zu Kalendertagen sollte überarbeitet werden. In die Implementierung werden fest acht Personenstunden für einen Arbeitstag angenommen. Diese Umrechnung berücksichtigt nicht, dass mehrere Teammitglieder an den Testarbeiten einer Testperiode mitarbeiten können. Außerdem werden ebenfalls Wochenenden und Feiertage bei der Errechnung der Kalendertage vernachlässigt. Denkbar ist eine umfassende Verwaltung von Testing-Teams, denen Benutzer des SGR zugeordnet werden können. Der Testaufwand einer Testperiode kann auf diese Weise dynamisch unter den beteiligten Testing-Teams aufgeteilt werden.

7.2.3 Interaktive Darstellung des Abhängigkeitsgraphen

Zur Validierung der entstehenden Testzeitpläne muss das Ergebnis häufig mit dem zu Grunde liegenden Abhängigkeitsgraphen abgeglichen werden. Die manuelle Variante, den Graphen aufzuzeichnen, ist fehleranfällig und zeitintensiv. Da eine geeignete Datenstruktur während der Ausführung bereits aufgebaut wird (siehe Abschnitt 3.2.1), bietet es sich an, den Abhängigkeitsgraphen mit Hilfe dieser zu visualisieren. Eine interaktive Visualisierung könnte sogar die Verwaltung von Systemen, Services, Service-Versionen und deren Abhängigkeiten (Verträge) erlauben.

7.2.4 Weitere Optimierungsmöglichkeiten

Sollten sich bei der Nutzung der Komponente aus Kapitel 5 neue Anforderungen an einen Zeitplan ergeben, können diese leicht im existierenden Prototypen umgesetzt werden. Die modulare Art und Weise, wie Optimierungen entwickelt und angewendet werden, erlaubt freie Änderungen zu jedem Zeitpunkt des automatisierten Vorgangs. Die im Rahmen dieser Arbeit umgesetzten Optimierungsverfahren aus Kapitel 4 werden im Prototypen jeweils in einer separaten Klasse implementiert. Zusammen mit der Vererbung von Standardwerten der Oberklasse *ScheduleOption*, dienen die bestehenden Optimierungen auch als Leitlinie zur Entwicklung weiterer Optimierungen. An dieser Stelle wäre auch ein Plug-In-System denkbar, das zur Laufzeit die vorhandenen Optimierungen laden kann. So können diese leicht ausgetauscht und angepasst werden.

7.3 Ausblick

Die folgenden Abschnitte geben einen Ausblick auf den weiterführenden Kontext des entwickelten Konzepts.

7.3.1 Testzeitplan-Generierung in anderen Domänen

Obwohl die Konzeption und Umsetzung des Algorithmus im Hinblick auf den Einsatz mit einer SOA entstanden ist, kann dieser grundsätzlich auch in anderen Domänen verwendet werden. In [Wic15] werden beispielsweise die Abhängigkeiten von Steuerungskomponenten eines Roboters untersucht. Durch deren Analyse kann eine bestimmte Strategie zur Ausführung der Module (zur Auswertung von Bilddaten, Bewegungssensoren oder anderen kognitiven Reizen) umgesetzt werden. Auch bei herkömmlicher komponentenbasierter Anwendungssoftware müssen komplizierte Integrationstests geplant werden. Entwickler arbeiten dabei häufig eigenständig an der Umsetzung von zuvor definierten Arbeitspaketen, die später von einem Integrator zusammengefügt werden müssen. Sind die Abhängigkeiten der Pakete untereinander bekannt, so könnte der Algorithmus die Integrationsarbeit sinnvoll strukturieren und die Anzahl notwendiger Stub-Implementierungen minimalisieren. Auch außerhalb der digitalen Welt gibt es etliche Gebiete, in denen die Integration von Hardware-Komponenten geplant werden muss. Als Beispiel lässt sich der Test mehrerer Fahrzeugkomponenten in einem PKW aufführen. Bevor die Klimaautomatik für den Fahrzeuginnenraum getestet werden kann, muss zunächst ein funktionierendes Lüftungssystem vorhanden sein. Dieses benötigt wiederum eine konstante Stromquelle und eine gewisse Abgeschlossenheit des Innenraums, beispielsweise durch Türen, Front- und Heckscheibe. Je mehr Komponenten beteiligt sind und voneinander abhängen, desto schwieriger und aufwändiger wird die manuelle Planung. Hier könnte eine generische und domänenübergreifende Implementierung des Algorithmus hilfreich sein.

7.3.2 Reaktiver dynamischer Zeitplan

Testzeitpläne stellen den geplanten Ablauf von Tests in der Theorie dar. Ein Zeitplan beinhaltet keine Informationen darüber, was in der Praxis bei Nicht-Einhaltung von Terminen geschehen soll. In der Praxis können im Hinblick auf eine Testperiode vier Fälle auftreten, die den Start- und Endzeitpunkt einer Testperiode beeinflussen können.

1. Der Endzeitpunkt einer Testperiode kann aus unterschiedlichen Gründen nicht eingehalten werden.
2. Die für eine Testperiode vorgesehenen Arbeiten konnten schon vor Ende der Testperiode erfolgreich abgeschlossen werden.
3. Der Endzeitpunkt einer im Voraus eingeplanten Testperiode kann nicht eingehalten werden.
4. Die Testarbeiten einer im Voraus eingeplanten Testperiode können früher als zum Endzeitpunkt der Testperiode abgeschlossen werden.

Die derzeitige Implementierung der Testzeitpläne sieht keine Möglichkeit vor, den Zeitplan beim Eintreten einer der Fälle anzupassen. Vielmehr verständigen sich die Stakeholder auf die Umsetzung eines Zeitplans und gehen von seiner dauerhaften Gültigkeit aus. Die gute Vernetzung und die Möglichkeit, betroffene Stakeholder über technische Änderungen zu informieren, qualifizieren das SOA Governance Repository für den Einsatz dynamischer Testzeitpläne. Für die genannten Fälle muss dann spezifiziert werden, wie auf das Eintreten eines Falles reagiert wird. Für die Fälle *1* und *3* könnten nachfolgende Testperioden um die Dauer der Verzögerung aufgeschoben werden, wenn zum Releasezeitpunkt ohnehin noch ein Pufferzeitraum besteht. Ist dies nicht der Fall, könnten darauffolgende Testperioden die Verzögerung durch ihre gleichverteilte Verkürzung kompensieren. Im Gegenteil dazu, können frei werdende Zeitreserven in den Fällen *2* und *4* nachfolgenden Testperioden zu Gute kommen, indem die zusätzliche Zeitspanne gleichmäßig auf nachfolgende Testperioden verteilt wird.

7.4 Fazit

Das Testen einer SOA mit vielen Komponenten stellt für das Testpersonal eine große Herausforderung dar. Können Tests von eigenständigen Services noch mit etablierten Testmethoden (Unit-Tests) durchgeführt werden, müssen die Tests von Service-Kompositionen mit umfangreichen Integrations- und Systemtests akribisch geplant werden. Nur so kann die koordinierte Interaktion von Services Stück für Stück aufgebaut und getestet werden. Im Unternehmensumfeld sollten SOA-Governance-Lösungen eingesetzt werden, um die Kontrolle über die SOA zu behalten. Sie verwalten und dokumentieren Services, Service-Konsumenten und die Verträge untereinander. Insbesondere die hohe Dynamik, die sich durch die Bindung an eine Service-Implementierung zur Laufzeit ergibt, steht häufig mit der gewünschten Zuverlässigkeit und

Reproduzierbarkeit der Ausführung in Konflikt. Im Rahmen der SOA-Governance wird deshalb der Fokus auf die bewusste und feste Einplanung von Verträgen zwischen Service-Konsumenten und Service-Versionen gerichtet. Das vorgestellte Testzeitplan-Generierungsverfahren nutzt die Kenntnis über die bestehenden Abhängigkeiten einer SOA aus, um einen maßgeschneiderten Testzeitplan zu generieren. Besonders große Architekturen mit zahlreichen Systemen und Services profitieren stark von diesem automatisierten Vorgang. Im Gegensatz zu Tests, die eine große Service-Komposition im Ganzen testen, verkleinert das vorgestellte Konzept die Problemstellung durch Tests einzelner Interaktionen. Dadurch wird die Fehler-Ursache-Analyse wesentlich vereinfacht, da in jeder Testperiode nur eine einzelne ungetestete Komponente beteiligt ist. Darüber hinaus wurden Optimierungen entwickelt, die einen generierten Testzeitplan im Hinblick auf bestimmte Anforderungen verbessern können. Die durchgeführte Fallstudie konnte den praktischen Nutzen der Implementierung im SGR unter Beweis stellen. Dafür wurde ein zufälliges Verbundrelease erzeugt und unter Verwendung aller Optimierungsmöglichkeiten ein durchführbarer Testzeitplan generiert.

Literaturverzeichnis

[Bar+09] C. Bartolini, A. Bertolino, S. Elbaum, E. Marchetti. „Whitening SOA Testing“. In: *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*. ESEC/FSE '09. Amsterdam, The Netherlands: ACM, 2009, S. 161–170. ISBN: 978-1-60558-001-2. DOI: 10.1145/1595696.1595721. URL: http://doi.acm.org/10.1145/1595696.1595721 (zitiert auf S. 13).

[Bar+11] C. Bartolini, A. Bertolino, F. Lonetti, E. Marchetti. „Approaches to functional, structural and security SOA testing“. In: *Performance and Dependability in Service Computing: Concepts, Techniques and Research Directions. Accepted for publication* (2011), S. 381–383. DOI: 10.4018/978-1-60960-794-4.ch017. URL: http://www.igi-global.com/viewtitlesample.aspx?id=55527 (zitiert auf S. 16).

[BCD07] S. Basu, F. Casati, F. Daniel. „Web Service Dependency Discovery Tool for SOA Management“. In: *IEEE International Conference on Services Computing (SCC 2007)*. 2007, S. 684–685. DOI: 10.1109/SCC.2007.130 (zitiert auf S. 17).

[BGW02] S. Büttcher, M. Gröbner, P. Wilke. *Entwurf und Implementierung eines Genetischen Algorithmus zur Erstellung von Zeitplänen*. November. 2002 (zitiert auf S. 12, 25).

[Bor+10] B. Borisov, V. Pavlov, D. Petrova-Antonova, S. Ilieva. „Framework for Testing Service Compositions“. In: *2011 13th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing* 00 (2010), S. 557–560. URL: http://doi.ieeecomputersociety.org/10.1109/SYNASC.2010.45 (zitiert auf S. 16).

[BP09] A. Bertolino, A. Polini. „SOA Test Governance: Enabling service integration testing across organization and technology borders“. In: *IEEE International Conference on Software Testing, Verification, and Validation Workshops, ICSTW 2009* (2009), S. 277–286. DOI: 10.1109/ICSTW.2009.39 (zitiert auf S. 18).

[BP12] P. Bhuyan, C. Prakash. *A Survey of Regression Testing in SOA*. April. 2012, S. 22–25. DOI: 10.5120/6371-8779 (zitiert auf S. 16).

[CDP06] G. Canfora, M. Di Penta. „Testing Services and Service-Centric Systems: Challenges and Opportunities“. In: *IT Professional* 8.2 (März 2006), S. 10–17. ISSN: 1520-9202. DOI: 10.1109/MITP.2006.51. URL: http://dx.doi.org/10.1109/MITP.2006.51 (zitiert auf S. 13, 14).

[CDP09] G. Canfora, M. Di Penta. „Service-Oriented Architectures Testing: A Survey“. In: *Software Engineering: International Summer Schools, ISSSE 2006-2008, Salerno, Italy, Revised Tutorial Lectures*. Hrsg. von A. De Lucia, F. Ferrucci. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, S. 78–105. ISBN: 978-3-540-95888-8. DOI: 10.1007/978-3-540-95888-8_4. URL: http://dx.doi.org/10.1007/978-3-540-95888-8_4 (zitiert auf S. 16).

[CP+08] A. Cerdeira-Pena, L. Carpente, A. Fariña, D. Seco. „New approaches for the school timetabling problem“. In: *7th Mexican International Conference on Artificial Intelligence - Proceedings of the Special Session, MICAI 2008* (2008), S. 261–267. DOI: 10.1109/MICAI.2008.19 (zitiert auf S. 25).

[Cum09] F. A. Cummins. *Service-Oriented Architecture*. 2009, S. 27–73. ISBN: 1591407990. DOI: 10.1016/B978-0-12-374445-6.00002-9. URL: http://linkinghub.elsevier.com/retrieve/pii/B9780123744456000029 (zitiert auf S. 11).

[DLDB09] P. De Leusse, T. Dimitrakos, D. Brossard. „A governance model for SOA“. In: *2009 IEEE International Conference on Web Services, ICWS 2009* (2009), S. 1020–1027. DOI: 10.1109/ICWS.2009.132 (zitiert auf S. 17).

[Erl04] T. Erl. *Service Orientated Architecture - A Field Guide to Integrating XML and Web Services*. 2004, S. 1 –558 (zitiert auf S. 11).

[Erl05] T. Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2005. ISBN: 0131858580 (zitiert auf S. 9, 11).

[GG07] L. Goasduff, Gartner. *Bad Technical Implementations and Lack of Governance Increase Risks of Failure in SOA Projects*. 2007 (zitiert auf S. 17).

[GW02] M. Groebner, P. Wilke. „A General View on Timetabling Problems“. In: *Proceedings of the 4th international conference on the Practice and Theory of Automated Timetabling*. Hrsg. von E Burke, P. DeCausmaecker. Belgium. Bd. 1. KaHo St. Lieven, 2002, S. 221–227. ISBN: 9080609617 (zitiert auf S. 12).

[Had12] N. Hadizadeh. *Stakeholder involvement in SOA: analyzing service identification as co-design*. 2012 (zitiert auf S. 14).

[HL05] R. Heckel, M. Lohmann. „Towards Contract-based Testing of Web Services“. In: *Electronic Notes in Theoretical Computer Science* 116 (2005), S. 145–156. URL: http://www.sciencedirect.com/science/article/pii/S1571066104052831 (zitiert auf S. 13).

[IA07] S. Inaganti, S. Aravamudan. „SOA Maturity Model“. In: *BPTrends* April (2007), S. 1–23. DOI: 10.1007/978-3-319-02453-0 (zitiert auf S. 21).

[JNR09] C. Janiesch, M. Niemann, N. Repp. *Towards a service governance framework for the internet of services*. 2009, S. 1–13 (zitiert auf S. 20).

[JPW13] S. Jehan, I. Pill, F. Wotawa. „Functional SOA testing based on constraints“. In: *2013 8th International Workshop on Automation of Software Test, AST 2013 - Proceedings* (2013), S. 33–39. DOI: 10.1109/IWAST.2013.6595788 (zitiert auf S. 16).

[KG12] P. Kalamegam, Z. Godandapani. „A survey on testing SOA built using web services“. In: *International Journal of Software Engineering and its Applications* 6.4 (2012), S. 91–104. ISSN: 17389984 (zitiert auf S. 12, 13, 15).

[KM16] J. Königsberger, B. Mitschang. „A Semantically-enabled SOA Governance Repository“. In: *Proceedings of the 2016 IEEE 17th International Conference on Information Reuse and Integration*. Hrsg. von I. C. Society. IEEE Computer Society Conference Publishing Services, 2016, S. 423–432. ISBN: 978-1-5090-3207-5. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2016-35&engl= (zitiert auf S. 18–20, 44).

[KSM14] J. Königsberger, S. Silcher, B. Mitschang. „SOA-GovMM: A meta model for a comprehensive SOA governance repository“. In: *Proceedings of the 2014 IEEE 15th International Conference on Information Reuse and Integration, IEEE IRI 2014* (2014), S. 187–194. DOI: 10.1109/IRI.2014.7051889 (zitiert auf S. 17, 19, 20).

[Mei+12] L. Mei, K. Zhai, B. Jiang, W. K. Chan, T. H. Tse. „Preemptive regression test scheduling strategies: A new testing approach to thriving on the volatile service environments“. In: *Proceedings - International Computer Software and Applications Conference* (2012), S. 72–81. ISSN: 07303157. DOI: 10.1109/COMPSAC.2012.17 (zitiert auf S. 16).

[Miy15] V. Y. Miyai. *RDF-based Data Model for a SOA Governance Repository*. 2015 (zitiert auf S. 44).

[Moh+12] R. K. Mohanty, B. K. Pattanayak, B. Puthal, D. P. Mohapatra. „a Road Map To Regression Testing of Service-Oriented Architecture (SOA) Based Applications“. In: *Journal of Theoretical and Applied Information Technology* 36.1 (2012) (zitiert auf S. 16).

[PGFT11] M. Palacios, J. García-Fanjul, J. Tuya. „Testing in Service Oriented Architectures with dynamic binding: A mapping study“. In: *Information and Software Technology* 53.3 (2011), S. 171–189. ISSN: 09505849. DOI: 10.1016/j.infsof.2010.11.014 (zitiert auf S. 13).

[RMI07] L Ribarov, I Manova, S Ilieva. „Testing in a service-oriented world“. In: *Proceedings of the International Conference on Information Technologies InfoTech2007* September (2007), S. 1–10 (zitiert auf S. 16).

[Ros08] M. Rosen. *Applied SOA Service-Oriented Architecture and Design Strategies*. 2008, S. 699. ISBN: 9780470223659 (zitiert auf S. 11).

[San+05] N. Sangal, E. Jordan, V. Sinha, D. Jackson. „Using dependency models to manage complex software architecture“. In: *ACM SIGPLAN Notices* 40.10 (2005), S. 167. ISSN: 03621340. DOI: 10.1145/1103845.1094824 (zitiert auf S. 17).

[Tra+00] J. B. Tran, M. W. Godfrey, E. H. S. Lee, R. C. Holt. „Architecture Repair of Open Source Software“. In: *International Workshop on Program Comprehension (IWPC'00)* (2000) (zitiert auf S. 17).

[Wic15] M. Wichner. „Abhängigkeitserkennung und parallele Ausführung von Code-Modulen im FUmanoid-Framework“. Bachelor Thesis. Freie Universität Berlin, 2015 (zitiert auf S. 69).

[Wot+13] F. Wotawa, M. Schulz, I. Pill, S. Jehan, P. Leitner, W. Hummer, S. Schulte, P. Hoenisch, S. Dustdar. „Fifty shades of grey in SOA testing“. In: *Proceedings - IEEE 6th International Conference on Software Testing, Verification and Validation Workshops, ICSTW 2013* (2013), S. 154–157. DOI: 10.1109/ICSTW.2013.26 (zitiert auf S. 13).

[Zho+07a] J. Zhou, D. Pakkala, J. Perälä, E. Niemelä, J. Riekki, M. Ylianttila. „Dependency-aware Service Oriented Architecture and Service Composition“. In: *Web Services, 2007. ICWS 2007. IEEE International Conference on* Icws (2007), S. 1146–1149. DOI: 10.1109/ICWS.2007.71 (zitiert auf S. 13).

[Zho+07b] J. Zhou, E. Niemelä, J. Perälä, D. Pakkala. „Web service in context and dependency-aware service composition“. In: *Proceedings of The 2nd IEEE Asia-Pacific Services Computing Conference, APSCC 2007* (2007), S. 349–355. DOI: 10.1109/APSCC.2007.4414481 (zitiert auf S. 13).

Alle URLs wurden zuletzt am 20.11.2016 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift