Freie Universität Berlin

Bachelorarbeit am Institut für Informatik der Freien Universität Berlin,

Arbeitsgruppe Software Engineering

Entwicklung eines generischen Sequenzgenerators und seine Verwendung im Bereich modellbasierten Testens

Sascha Schreckenbach
Matrikelnummer: 4494464
sascha.schreckenbach@piketec.com

Betreuerin an der Freien Universität: Prof. Dr. Ina Schieferdecker
Betreuer bei PikeTec: Dr. Eckard Bringmann und Dr. Jens Lüdemann
Zweitgutachterin: Prof. Dr. Katinka Wolter

Berlin, 23.09.2015

Zusammenfassung

Sequenzgeneratoren erzeugen aus Mengen von Elementen Folgen beziehungsweise Sequenzen dieser Elemente. Sind die Elemente in Modellen organisiert, werden diese auf der Suche nach gültigen Sequenzen durchmustert. Verwendung finden solche Generatoren beim Modellbasierten Testen, kurz MBT.

MBT ist ein Sammelbegriff für Methoden zur modellgestützen Automatisierung verschiedener, auf den Softwaretest bezogener Aktivitäten. Das Ableiten von Testfällen aus Testmodellen ist eine dieser Aktivitäten.

Bei der Planung und den ersten Experimenten zur Entwicklung eines neuen Testfallgenerators für das MBT-Werkzeug TPT (Time Partition Testing) entstand die Idee zu einem allgemeinen, intuitiven Ansatz für einen generischen Sequenzgenerator. Dieser sollte das eigentliche Zusammensetzen der Sequenzen in den Vordergrund rücken und das verwendete Modell, das spezielle Vorgehen bei dessen Durchmusterung und die Generierungsziele hinter schlanken und eingängigen Schnittstellen verbergen. Man versprach sich davon eine Möglichkeit, auf einfache Weise mit verschieden Modellen und Vorgehensweisen experimentieren zu können, um eine gute Basis für den angestrebten Testfallgenerator zu finden.

Die Entwicklung dieses generischen Sequenzgenerators ist Gegenstand dieser Arbeit.

Eidesstattliche Erklärung

Ich versichere hiermit an Eides Statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

23.09.2015

Sascha Schreckenbach

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Einordnung in den großen Kontext Softwareentwicklung und Softwaretest . . . 1

1.2 Motivation zu dieser Arbeit . . . 3

1.3 Problemstellung . . . 3

1.4 State of the Art . . . 4

1.5 Aufbau dieser Arbeit . . . 4

2 Konzept **6**

2.1 Überblick . . . 6

2.2 Erste Problemzerlegung anhand eines einfachen, zustandslosen Generierungsalgorithmus . . . 6

2.3 Diskussion zweier Erweiterungen des Generierungsalgorithmus . . . 9

2.4 Zustandsbasierter Generierungsalgorithmus . . . 10

2.5 Modell . . . 12

2.6 Zielvorgaben . . . 12

2.7 Strategie . . . 13

2.8 Inneres Abbruchkriterium . . . 13

2.9 Äußeres Abbruchkriterium . . . 14

2.10 Brauchbarkeit von Sequenzen . . . 14

3 Der generische Sequenzgenerator **15**

3.1 Überblick . . . 15

3.2 Architektur und Entwurf . . . 15

3.3 Das Interface `IModel` . . . 16

3.4 Das Interface `IStrategy` . . . 17

3.5 Das Interface `IObjective` . . . 17

3.6 Das Interface `IContainer` . . . 17

3.7 Die Klasse `Generator` . . . 18

3.8 Veranschaulichende Beispiele . . . 18

3.8.1 Beispiel 1: zufällige Sequenzen bestimmter Länge und Zahl . . . 18

3.8.2 Beispiel 2: Deterministischer endlicher Automat „Geordnete Wörter“ 19
3.8.3 Beispiel 3: parabolische Irrfahrt in $\mathbb{Z}$ 21

4 Komplexe, praxisrelevante Beispiele 24
4.1 Überblick 24
4.2 Beispiel 4: Online-Testing-Algorithmus nach T. Kanstrén und M. Chechik 24
4.3 Beispiel 5: Steplist-Generator in TPT 28
4.4 Anwendungsbeispiel „Fensterputzen“ in TPT 32

5 Fazit 35

6 Ausblick 36

Danksagung 37

A Sequenzgenerator - Alle Klassen und Interfaces 38

B Beispiel 1 - Alle Klassen 42

C Beispiel 2 - Alle Klassen 45

D Beispiel 3 - Alle Klassen 48

E Beispiel 4 - Ausgewählte Klassen 51

Abbildungsverzeichnis

1.1 Sequenzgenerierung 2

2.1 Der zustandslose Algorithmus als Flussdiagramm 7
2.2 Prinzipieller Aufbau des Sequenzgenerators 9
2.3 Der Generierungsalgorithmus als Flussdiagramm 11

3.1 Klassendiagramm des Sequenzgenerators 16
3.2 `IModel` 16
3.3 `IStrategy` 17
3.4 `IObjectives` 17
3.5 `IContainer` 17
3.6 `Generator` 18
3.7 Markov-Kette „parabolische Irrfahrt in $\mathbb{Z}$“ 22

4.1 DEA mit der Sprache $L = \{w \in \{A, B, C, D\}^* : 0 < |w| <= 5\}$ 26
4.2 Sequenzdiagramm zur Synchronisierung von `Strategy4` und `Explorer` mit einem `BarrierContainer` 27
4.3 „Hierarchical automatons“-Example aus TPT 8 (Pre-Release) 29
4.4 Prinzip der verwendeten erweiterten Zustandsautomaten . . . 30
4.5 Erweiterter endlicher Zustandsautomat „Fensterputzen“ als Graph 32
4.6 Modellierung des Automaten „Fensterputzen“ in TPT 33
4.7 Testlets, Varianten und generierte Testfälle in TPT 34
4.8 Eine generierte Schrittliste 34

Kapitel 1

Einleitung

1.1 Einordnung in den großen Kontext Softwareentwicklung und Softwaretest

Computer halten stetig Einzug in unser Leben. Sie dringen in beinahe jeden Bereich vor, um uns zu helfen, zu unterhalten oder zu überwachen, und sie werden dabei immer leistungfähiger, handlicher und billiger. Scheinbar wird, wo immer es geht, irgendwann ein Chip platziert, mal um unser Leben einfacher und sicherer, mal teurer und transparenter zu machen. Für die edelsten und für die fragwürdigsten Zwecke werden Computer eingesetzt. Der Grund dafür ist ihre Vielseitigkeit.

Noch vielseitiger als ihre Einsatzmöglichkeiten sind die Programme, also die Software[1], die auf ihnen laufen, um die gestellten Aufgaben zu erledigen. Für viele Probleme und Ansprüche gibt es mehrere Lösungen und fast all diese Lösungen haben eines gemeinsam: Sie haben Mängel. Mängel führen zu Fehlverhalten und Fehlverhalten verursacht Schaden. Die Möglichkeiten, durch fehlerhafte Software Schaden anzurichten, sind ebenso mannigfaltig wie ihr Nutzen, wenn sie funktioniert. Im besten Fall kommt ein Mangel überhaupt nicht zum Tragen, im schlimmsten kostet er Leben.

Um Mängel oder Defekte zu finden bevor sie zum Versagen einer Software führen, wird getestet, und zwar je nach Qualitätsanforderungen unterschiedlich gründlich. Wie fast alles heutzutage unterliegt auch der Softwaretest ökonomischen Zwängen, denn es hängt vom Verhältnis zwischen Kosten und Nutzen ab, wieviel ein Software-Unternehmen in die Qualitätssicherung investiert. In vielen Bereichen wurden inzwischen Normen und Standards etabliert, um ein angemessenes Qualitätslevel zu halten.

[1] Der Software-Begriff ist wesentlich weiter gefasst und nicht nur auf Programme beschränkt. In [1] findet man folgenden Eintrag: „**software.** Computer programs, procedures, and possibly associated documentation and data pertaining to the operation of a computer system.“

Je höher die Ansprüche an die Zuverlässigkeit einer Software sind, umso mehr muss getestet werden. Dazu wird eine Reihe von Testfällen ausgeführt, während derer die Software oder ein Teil davon unter gegebenen Randbedingungen mit bestimmten Eingaben stimuliert wird, um Voraussagen über Ausgaben oder Verhalten zu prüfen. Mit steigender Komplexität der Systeme steigt auch die Zahl der möglichen Testfälle, so dass es schnell unmöglich wird, alles daran zu testen. Das ist zum Glück auch selten nötig. Neben der eigenen Intuition gibt es zahlreiche Methoden zur Ableitung einer Menge von Testfällen - man spricht dabei oft von Testsets oder, im größeren Maßstab, von Testsuiten - für ein System, mit Unterschieden in Granularität und Fokus.

Der Begriff *modellbasiertes Testen* (MBT) beinhaltet verschiedene Verfahren zum Ableiten von Testfällen aus Modellen. So repräsentieren sogenannte Testmodelle ganze Teilmengen der möglichen Testfälle und es lassen sich leicht konkrete Testfälle aus ihnen ableiten. An dieser Stelle kommen auch Testfallgeneratoren ins Spiel, die, wenn im Modell alles Nötige enthalten ist, diese Ableitung automatisieren. Anstatt viele mehr oder weniger gleichartige Testfälle selbst zu schreiben, muss ein Tester also lediglich ein Modell entwerfen, das die Idee hinter diesen Testfällen abbildet und benötigte Testdaten enthält oder berechnet. Das spart Zeit und schafft Transparenz. Am Modell lässt sich häufig leichter ablesen, welche Bereiche es abdeckt, als an einer großen Menge konkreter Testfälle.

Testfälle beinhalten eine Handlungsanweisung zur Durchführung des Tests. Das ist in der Regel eine Folge von Testschritten, einfache, atomare Handlungen, die zum Beispiel das Testobjekt stimulieren, es prüfen oder Berechnungen anstellen. Welche Testschritte es gibt und in welcher Reihenfolge sie durchgeführt werden sollen, kann in Testmodellen formuliert werden. Ein Testfallgenerator erzeugt dann daraus Folgen von Testschritten und darauf aufbauend Testfälle. Die Hauptfunktion eines solchen Generators ist es also, in Modellen organisierte Dinge aneinanderzureihen, Sequenzen daraus zu bilden.

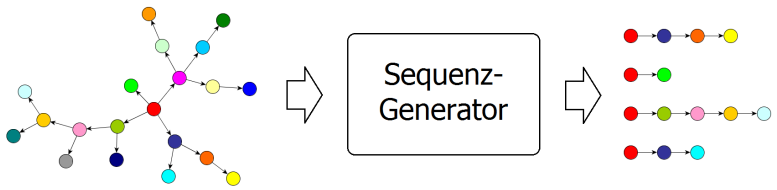

Abbildung 1.1: Sequenzgenerierung

Um diese Sequenzgenerierung wird es in der vorliegenden Arbeit gehen. Es wird ein Werkzeug entwickelt, das es ermöglicht, alle möglichen Dinge, die

auf verschiedene Weise in Modellen organisiert sind, aneinanderzureihen.

1.2 Motivation zu dieser Arbeit

Vor einiger Zeit unterstützte ich eine Kollegin bei ihrer Masterarbeit[2], in der es darum ging, eine Idee zu einem neuen Testfallgenerator für das MBT-Werkzeug TPT, das von der PikeTec GmbH[3] entwickelt und vertrieben wird, auszuarbeiten und zu evaluieren. Die Generierung sollte auf Basis der TPT-Automaten mit Hilfe einer darauf spezialisierten Open-Source-Software, dem GraphWalker[4], erfolgen. Dabei handelt es sich ebenfalls um ein MBT-Werkzeug, das seinen Fokus auf der Erzeugung von Testsequenzen aus erweiterten endlichen Zustandsautomaten hat. Aus verschiedenen Gründen scheiterte die Umsetzung dieser Idee in TPT vorerst und wir erarbeiteten einen neuen Ansatz: die Generierung von Schrittlisten.

Das zu diesem Zeitpunkt noch recht neue TPT-Feature der *Steplist* bietet die Möglichkeit, einfache oder komplexe Testschritte sequenziell oder parallel auszuführen. Es sollten nun aus einer gegebenen Menge von Testschritten eine Anzahl von Schrittlisten generiert werden. Die Reihenfolge der Testschritte sollte durch ein Testmodell geregelt werden, das gleichzeitig die Menge aller möglichen Schrittlisten repräsentiert. Für die Generierung sollten bestimmte Abdeckungsziele[2] definiert werden können, die der Generator dann mit möglichst wenigen, möglichst kurzen Schrittlisten erreichen sollte.

Diese noch recht unkonkreten Anforderungen warfen bereits die wichtigsten Fragen auf: nach einer geeigneten Art von Modell, nach einer Möglichkeit Generierungsziele zu formulieren, nach einer flexiblen Implementierung für ein effektives Vorgehen bei der Generierung und schließlich nach der Generierung selbst. Es stellte sich als gute Idee heraus, diese Probleme getrennt anzugehen. Zusammen mit der Erkenntnis, dass sich die Schrittlistengenerierung auf eine Sequenzgenerierung herunterbrechen lässt, führte das zu der Baukastenlösung, die in dieser Arbeit vorgestellt wird. Die Anwendung in TPT wird im letzten Beispiel in Abschnitt 4.3 beschrieben.

1.3 Problemstellung

Es soll ein „generischer Sequenzgenerator“ implementiert werden, der durch seinen Aufbau eine zweckmäßige, intuitive Problemzerlegung abbildet. Es sollen sich damit Mengen von Sequenzen aus beliebigen Elementen nach flexiblen Vorgaben generieren lassen. Diese Vorgaben sind insbesondere verschiedene Arten von Modellen, aber auch unterschiedliche Herangehensweisen beziehungsweise Strategien beim Treffen von Entscheidungen. Es soll die

[2] zum Beispiel die Verwendung jedes Testschrittes

Möglichkeit geben, die generierten Sequenzen zu bewerten und diese Bewertungen in die Generierung einfließen zu lassen.

Der Generator wird in der Programmiersprache Java geschrieben. Dafür gibt es mehrere Gründe. Erstens ist es die verwendete Sprache in seiner Entstehungsdomäne, dem MBT-Werkzeug TPT. Zweitens eignet sich Java als objektorientierte Sprache hervorragend, um den Generator und seine Bestandteile zu modellieren. Drittens fühlt sich der Autor damit am wohlsten.

1.4 State of the Art

Es gibt zahlreiche Ansätze zur Automatisierung von Aktivitäten in verschiedenen Phasen des Testprozesses. Im Jahr 2007 wurden im Rahmen einer systematischen Studie[5] über 400 wissenschaftliche Publikationen zum Thema erfasst, von denen etwa die Hälfte MBT-Methoden beschreiben. Die Autoren waren auf der Suche nach den wichtigsten Ansätzen und deren wichtigsten Eigenschaften. Beispielsweise standen die Tool-Unterstützung, der Automatisierungsgrad und die verwendeten Modelle im Fokus. Im Ergebnis ist unter anderem zu lesen:

> MBT approaches are usually not integrated with the software development process. The models used for tests generation are not integrated with artifacts from the software development process, or were defined exclusively by a MBT approach. Integration tools must be used to support this problem;

und

> Requirements to use a MBT approach include knowledge about the modeling language, testing coverage criteria, generated output format, supporting tools make the usage difficult/unfeasible; these need to be minimized;

Zu beiden Punkten kann die vorliegende Arbeit ein Beitrag leisten.

1.5 Aufbau dieser Arbeit

Die vorliegende Arbeit besteht aus sechs Kapiteln, von denen das zweite, dritte und vierte den Hauptteil bilden, während die letzten beiden, **Fazit** und **Ausblick**, für Nachbetrachtungen vorgesehen sind. Dieser Abschnitt beendet das erste Kapitel, die **Einleitung**.

Die drei Kapitel des Hauptteils, befassen sich mit abnehmendem Abstraktionsgrad mit dem Aufbau und der Implementierung des generischen Sequenzgenerators, dem Hauptgegenstand dieser Arbeit, und von konkreten

Beispielen. In Kapitel 2 - **Konzept** werden anhand eines vereinfachten Generierungsalgorithmus die Rollen *Generator*, *Modell*, *Strategie* und *Zielvorgaben* für die spätere Implementierung identifiziert. Anschließend wird der Algorithmus auf die endgültige Form erweitert. Die Implementierung erfolgt in Kapitel 3 - **Der generische Sequenzgenerator**. Passend zum Konzept werden Klassen, Interfaces und Methoden entworfen und anschließend in Java umgesetzt. Die durch die Klassen repräsentierten Rollen und deren Zusammenspiel werden am Ende des Kapitels in drei einfachen Beispielen veranschaulicht. In Kapitel 4 - **Komplexe, praxisrelevante Beispiele** wird in zwei komplexen Anwendungen die Praxistauglichkeit des generischen Sequenzgenerators auf die Probe gestellt. Die erste ist ein optimierender Online-Testing-Algorithmus aus [6]. Die zweite ist der Steplist-Generator von TPT.

Das Kapitel 5 - **Fazit** enthält eine kurze Stellungnahme zu den in dieser Arbeit erzielten Ergebnissen und Kapitel 6 - **Ausblick** gibt Auskunft darüber, was noch zu tun ist beziehungsweise getan werden kann.

Nach einer kurzen Danksagung folgen die Anhänge A bis E, die den vollständigen Code des generischen Sequenzgenerators und der drei ersten Beispiele sowie die interessanten Teile von Beispiel 4 enthalten. Mit Ausnahme von Anhang E ist der Code komplett unkommentiert. Er ist experimenteller Natur und wird in der vorliegenden Form nicht weiter verwendet. Da er recht übersichtlich ist und in Kapitel 3 ausführlich besprochen wird, konnte auf Kommentare verzichtet werden.

Kapitel 2

Konzept

2.1 Überblick

Dieses Kapitel bildet den Anfang des Hauptteils. In seinem ersten Abschnitt wird an einem einfachen Algorithmus gezeigt, auf welche Art Sequenzen zusammengesetzt werden sollen und es werden alle Beteiligten, *Modell*, *Strategie* und die *Zielvorgaben*, in ihren Rollen identifiziert. Dann wird die Erweiterung des Algorithmus um die Verwendung von Zuständen motiviert und umgesetzt. Anschließend werden erst die Rollen und danach drei Algorithmusdetails, die Zusammensetzungen des *inneren* und des *äußeren Abbruchkriteriums* sowie die Beurteilung der *Brauchbarkeit einer Sequenz*, genauer erklärt.

2.2 Erste Problemzerlegung anhand eines einfachen, zustandslosen Generierungsalgorithmus

Die einfachste und intuitivste Art Sequenzen zusammenzustellen ist, mit dem ersten Element zu beginnen und der Reihe nach weitere hinzuzufügen bis man fertig ist. Diese Vorgehensweise ist leicht beherrschbar und nicht zuletzt deshalb sinnvoll, weil viele Testmodelle auf gerichteten Graphen basieren, in denen Sequenzen aus den Knoten oder Kanten entlang der Pfade von einem Start- zu einem Endknoten gebildet werden.

Der geplante Algorithmus wird in zwei verschachtelten Schleifen eine Reihe von Sequenzen erzeugen. Im Rumpf der äußeren Schleife werden solange neue Sequenzen gestartet, bis ihr Abbruchkriterium erfüllt ist. In der inneren Schleife wird den Sequenzen Element für Element hinzugefügt. Auch hier gibt es ein Abbruchkriterium, das die Schleife und damit die gerade bearbeitete Sequenz beendet. In Listing 2.1 sind die Schritte dieses vereinfachten Generierungsalgorithmus in informeller Form dargestellt. Abbildung 2.1 zeigt einen passenden Programmablaufplan.

```
Eingabe: ein Container, in dem die fertigen Sequenzen
   abgelegt werden
Ausgabe: der Container mit den generierten Sequenzen
1 wenn genügend Sequenzen generiert worden sind,
   beenden
2 eine neue Sequenz starten
3 wenn die aktuelle Sequenz fertig ist, Schritt 7
4 alle Elemente, die als nächstes hinzugefügt werden
   können, ermitteln
5 aus diesen Elementen eines auswählen
6 dieses Element der aktuellen Sequenz hinzufügen und
   mit Schritt 3 fortfahren
7 brauchbare Sequenzen im Container ablegen, andere
   verwerfen, dann Schritt 1
```

Listing 2.1: zustandsloser Algorithmus

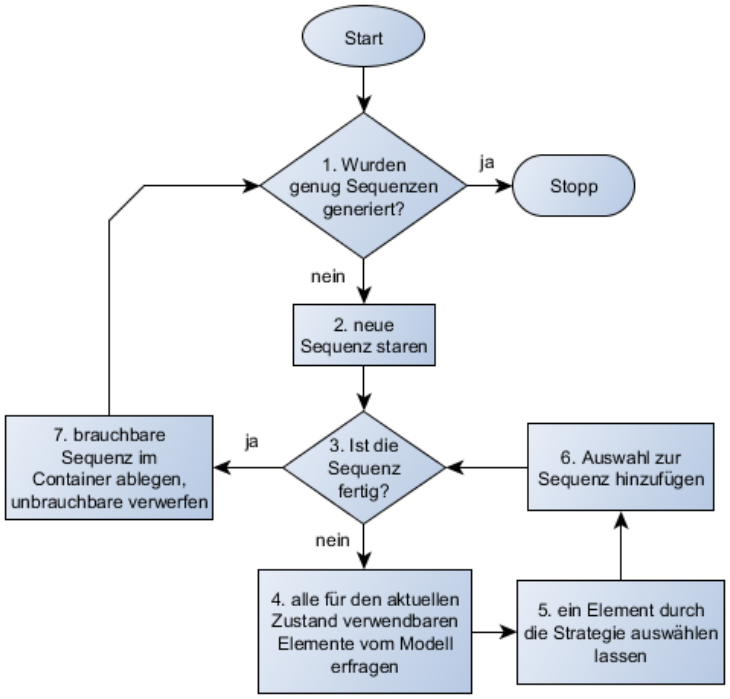

Abbildung 2.1: Der zustandslose Algorithmus als Flussdiagramm

Während die Anweisungen in den Schritten 2 und 6 vergleichsweise einfach sind, müssen an anderer Stelle verschiedene, teils komplexe Entscheidungen getroffen werden. Die fallen in unterschiedliche Zuständigkeiten. Einiges liegt im Bereich des Testmodells, anderes hängt eher mit den bei der Generierung verfolgten Strategien und Zielen zusammen:

Schritt 1 - Wurden genug Sequenzen generiert bzw. gibt es Gründe, keine weiteren Sequenzen zu generieren? Die Antwort auf diese Frage hängt fast ausschließlich von den für die Generierung angegebenen Zielvorgaben ab. Häufig wird eine auf eine bestimmte Art hinreichende Abdeckung des Modells angestrebt. Manchmal geht es darum, ob durch weitere Sequenzen noch irgendein Fortschritt zu erwarten ist. Dann ist die Entscheidung eher strategischer Natur.

Schritt 3 - Wann ist eine Sequenz fertig? Das Ende einer Sequenz kann sich aus dem Modell ergeben, wenn es z.B. keine weiteren Elemente liefert oder ein akzeptierender Zustand erreicht wird. Es kann aber auch eine Maximallänge vorgegeben sein oder es besteht einfach keine Notwendigkeit eine Sequenz zu verlängern.

Schritt 4 - Welche Elemente dürfen einer Sequenz hinzugefügt werden? Das hängt vom Modell und meist auch von der bisher generierten Sequenz ab - letzteres immer dann, wenn das Modell einen inneren Zustand hat, der sich bei jedem Schritt bzw. mit jedem neuen Element ändern kann. Der Algorithmus wird später so angepasst, dass Zustände eines Modells berücksichtigt werden können.

Schritt 5 - Welches ist das zum Hinzufügen geeignetste Element? Je nachdem, was beim Generieren erreicht werden soll, sind dafür verschiedene Kriterien ausschlaggebend. Beispielsweise wird in Graphen oft eine hohe Pfadvariation angestrebt, so dass Wiederholungen möglichst vermieden werden sollen. In anderen Fällen, z.B. bei Markov-Ketten, geschieht die Auswahl dagegen zufällig.

Schritt 7 - Ist eine beendete Sequenz brauchbar? Eine generierte Sequenz kann aus verschiedenen Gründen unbrauchbar oder ungültig sein. Ursachen dafür können im Modell oder in den Zielvorgaben liegen.

Nach diesen Überlegungen sind die an einer Generierung zu beteiligenden Module schnell identifiziert. Es wird ein *Modell* benötigt, das die Elemente bereitstellt. Welches Element zu welchem Zeitpunkt zu einer Sequenz hinzugefügt wird, hängt von der *Strategie* ab, die verfolgt wird, um bestimmte *Zielvorgaben* zu erreichen. Neben dem *Generator* selbst, in dem der beschriebene Algorithmus abläuft, sind das die drei Hauptbeteiligten bei einer Sequenzgenerierung: *Modell*, *Strategie* und die *Zielvorgaben*. Einige Schritte können nun eindeutig den Modulen zugeordnet werden. Schritt 4, die Bereitstellung der Elemente, fällt in die Zuständigkeit des *Modells* und Schritt 5, die Wahl eines bestimmten Elements zur Fortsetzung einer Sequenz, ist Sache der *Strategie*. Ob in Schritt 7 eine Sequenz als brauchbar eingestuft wird, wird an den *Zielvorgaben* gemessen. Auf die Abbruchkriterien in den Schritten 1 und 3 nehmen dagegen mehrere Module Einfluss. Eine ausführli-

chere Diskussion dieser beiden Punkte wird weiter unten in den Abschnitten 2.8 und 2.9 geführt.

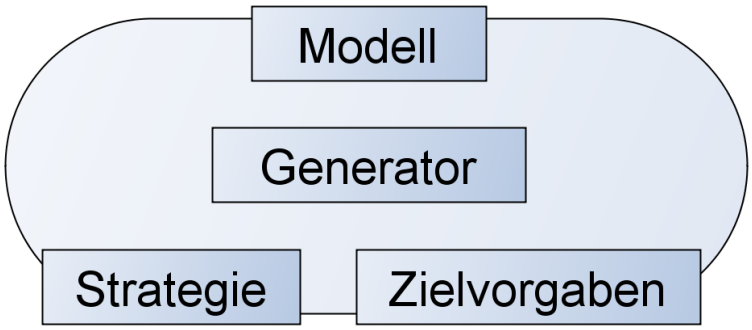

Abbildung 2.2: Prinzipieller Aufbau des Sequenzgenerators

2.3 Diskussion zweier Erweiterungen des Generierungsalgorithmus

Nachdem die Module identifiziert sind, wird nun der Algorithmus in seine endgültige Form gebracht.

Die erste Erweiterung betrifft Schritt 5, also die Wahl des Elements, das anschließend an die Sequenz angefügt wird. Diese Entscheidung fällt in den Bereich der *Strategie*. Davon ausgehend, dass immer mindestens ein Element zur Auswahl steht - warum, wird in Abschnitt 2.8 gezeigt - ist noch offen, was passiert, wenn die Strategie kein Element liefert. Dieses Verhalten könnte man in Schritt 3 sicherlich mit einigem Aufwand voraussehen und die Sequenz vorsorglich beenden. Für den folgenden Algorithmus wird aber festgelegt, dass die Sequenz im beschriebenen Fall zwischen den Schritten 5 und 6 endet. Das innere Abbruchkriterium wird dabei umgangen.

In der zweiten Erweiterung geht es um die in Schritt 4 vom Modell vorgegebene Auswahl der Kandidaten für das nächste Element einer Sequenz. Diese hängt häufig vom bisherigen Verlauf der Generierung ab, ein Umstand, der in Listing 2.1 nicht berücksichtigt wird. Der Algorithmus soll nun so erweitert werden, dass der Generator vor jedem Schritt in einen Zustand gebracht wird, an dem sich das Modell orientiert. Die Berechnung dieses Zustandes erfolgt durch das Modell. Der neue Zustand ergibt sich dabei aus dem alten und dem zuletzt zur Sequenz hinzugefügten Element. Außerdem gibt das Modell einen Startzustand vor. Die in der Folge für die Sequenz verwendbaren Elemente hängen dann vom aktuellen Zustand des Generators ab.

Im folgenden Abschnitt werden die nötigen Anpassungen am Algorithmus vorgenommen.

2.4 Zustandsbasierter Generierungsalgorithmus

Der erweiterte Algorithmus ist in Listing 2.2 abgedruckt und der zugehörige Programmablaufplan ist in Abbildung 2.3 zu sehen. Es wurden die Schritte 3, 7 und 9 hinzugefügt und Schritt 5 wurde erweitert.

```
Eingabe: ein Container, zur Ablage der fertigen
   Sequenzen sowie Modell, Strategie und Zielvorgaben
Ausgabe: der Container mit den generierten Sequenzen
1  wenn genügend Sequenzen generiert worden sind,
   beenden
2  eine neue Sequenz starten
3  vom Modell den Ausgangszustand erfragen und als
   aktuellen Zustand des Generators setzen
4  wenn die aktuelle Sequenz fertig ist, Schritt 10
5  vom Modell alle im aktuellen Zustand verwendbaren
   Elemente erfragen
6  aus diesen Elementen durch die Strategie eines
   auswählen lassen
7  Falls kein Element ausgewählt wurde, Schritt 10
8  dieses Element der aktuellen Sequenz hinzufügen
9  den neuen Zustand vom Modell erfragen und als
   aktuellen Zustand setzen und mit Schritt 4
   fortfahren
10 brauchbare Sequenzen im Container ablegen, andere
   verwerfen, dann Schritt 1
```

Listing 2.2: Generierungsalgorithmus

Die Generierung läuft nun etwa so ab: Der Generator startet mit einem leeren Container für die Sequenzen, einem Modell, einer Strategie und bestimmten Zielvorgaben. Vor dem Start einer neuen Sequenz entscheidet er, ob es nötig ist, eine weitere zu generieren. Künftig wird in diesem Zusammenhang vom „äußeren Abbruchkriterium“ gesprochen. Wenn es erfüllt ist, bricht die Generierung ab und der Container wird ausgegeben. Ansonsten startet er eine neue Sequenz und begibt sich in den vom Modell angegebenen Startzustand. Bevor versucht wird, der aktuellen Sequenz ein weiteres Element hinzuzufügen, überprüft der Generator, ob eine Fortsetzung nötig, gewollt oder überhaupt möglich ist. Die Komplexität dieser Entscheidung wird im Folgenden hinter der Bezeichnung „inneres Abbruchkriterium“ verborgen. Ist das innere Abbruchkriterium erfüllt, wird die Sequenz beendet. Der Generator prüft, ob sie brauchbar ist und legt sie gegebenenfalls im Container ab. Anschließend wird unter Berücksichtigung des äußeren Abbruchkriteriums eine neue Sequenz gestartet. Ist aber das innere Abbruchkriterium nicht

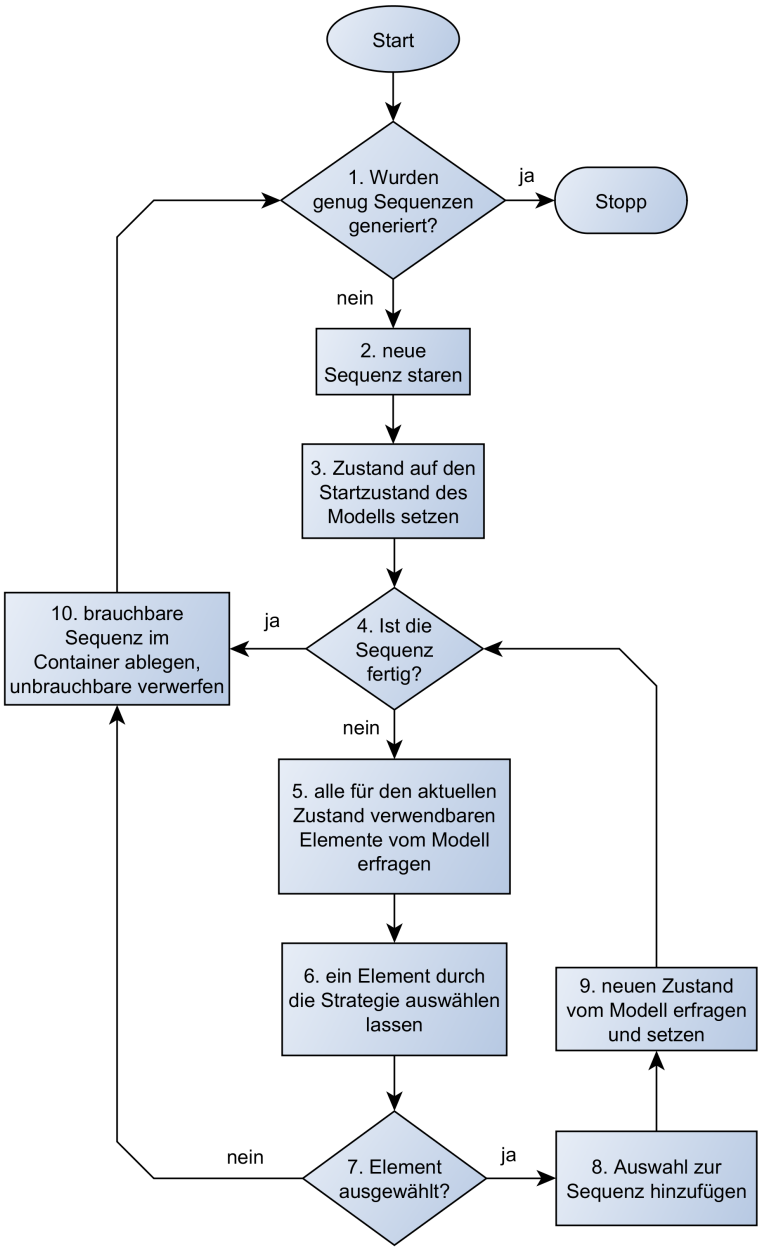

Abbildung 2.3: Der Generierungsalgorithmus als Flussdiagramm

erfüllt, besorgt sich der Generator vom Modell die Auswahl von Elementen, die im aktuellen Zustand als nächstes in Frage kommen, lässt anschließend die Strategie eines davon auswählen und fügt es der Sequenz hinzu. Das Modell berechnet nun aus dem aktuellen Zustand und dem hinzugefügten Element den neuen Zustand des Generators. Der Generator prüft daraufhin

wieder das innere Abbruchkriterium, um eventuell ein weiteres Element anzuhängen. Für den Fall, dass die Strategie kein Element auswählt, bricht die Sequenz sofort ab, wird ebenfalls auf Brauchbarkeit geprüft usw.

Die folgenden Abschnitte fassen die Rollen und Fähigkeiten der drei Module *Modell*, *Zielvorgaben* und *Strategie* zusammen und beschreiben zusätzliche Funktionen, die für die Abbruchkriterien und die Beurteilung fertiger Sequenzen wichtig sind.

2.5 Modell

Das Modell stellt die für die Sequenzen verwendeten Elemente zur Verfügung, organisiert ihre Reihenfolge und berechnet den Zustand des Generators. Dazu liefert es einen Startzustand und bestimmt, welche Elemente für einen bestimmten Zustand zur Verfügung stehen. Für einen gegebenen Zustand und ein Element liefert es einen folgenden Zustand.

Zusätzlich gibt es Auskunft darüber, ob für einen Zustand Elemente geliefert werden können und es kann außerdem Zustände als „akzeptierend“ auszeichnen. Beides ist für das innere Abbruchkriterium von Interesse, während Letzteres auch bei der Beurteilung der Brauchbarkeit einer Sequenz einfließt.

Wenn das Modell einen Zustand als akzeptierend einstuft, sind alle Sequenzen, deren Generierung den Generator in diesen Zustand überführen, aus Modellsicht akzeptabel.

Durch dieses Verhalten definiert das Modell neben seinem Startzustand implizit auch eine Menge von (erreichbaren) Zuständen. Die für einen Zustand angebotenen Elemente bilden ein Eingabealphabet und die Berechnung der Folgezustände, eine Zustandsübergangsfunktion. Die akzeptierenden Zustände bilden eine Menge von Endzuständen. So ein Modell simuliert also einen deterministischen Automaten. Eine Definition für deteministische endliche Automaten ist in Unterabschnitt 3.8.2 zu finden. Die Zustandsmenge muss hier allerdings nicht zwingend endlich sein. Im Folgenden soll das mathematische Modell jedoch vor der Intuition zurücktreten.

2.6 Zielvorgaben

Die Zielvorgaben beeinflussen die Generierung über die Abbruchkriterien und sieben unbrauchbare Sequenzen aus. Es gibt Ziele, die sich auf die Generierung einer einzelnen Sequenz beziehen und Ziele, deren Fokus auf allen fertig generierten und brauchbaren Sequenzen liegt.

Für das innere Abbruchkriterium wird entschieden, ob eine Sequenz im gegenwärtigen Stand das entsprechende Ziel erreicht oder, falls nicht, über-

haupt noch erreichen kann. Im äußeren Abbruchkriterium erfolgt diese Beurteilung bezüglich der Ziele über die bisher generierten Sequenzen. Des Weiteren kann eine Sequenz zurückgewiesen werden, wenn sie keinen Beitrag zu den Zielvorgaben leistet.

2.7 Strategie

Die Strategie ist allein verantwortlich für Variation bei der Sequenzgenerierung. Wenn es für das nächste hinzuzufügende Element mehrere Möglichkeiten gibt, ist es sie, die diese Wahl trifft. Für die Abbruchkriterien kann die Strategie veranlassen, eine Sequenz oder die ganze Sitzung zu stoppen. Sie kann aber auch empfehlen, eine Sequenz fortzusetzen.

2.8 Inneres Abbruchkriterium

Jetzt liegen alle Informationen vor, um über die Abbruchkriterien zu sprechen. Das innere ist das kompliziertere von beiden und betrifft die innere Schleife, also die Generierung einer einzelnen Sequenz. Ist es erfüllt, wird die Sequenz nicht fortgesetzt und gilt als beendet. Dafür kann es mehrere verschieden starke Gründe geben. Es wird grundsätzlich folgender Ansatz verfolgt: Eine Sequenz wird nur fortgesetzt, wenn nichts Wichtiges dagegen spricht und eine Verbesserung möglich ist, sonst wird sie beendet. Damit soll erreicht werden, dass sie nicht unnötig lang wird.

Wichtige Gründe zum Beenden einer Sequenz sind:

A: Die Zielvorgaben können nicht erfüllt werden.

B: Das Modell liefert keine Elemente für den aktuellen Zustand.

C: Die Strategie erzwingt das Beenden.

Liegt einer dieser Sachverhalte vor, ist das Kriterium erfüllt. Falls nicht, wird die Sequenz nur beendet, wenn jede der folgenden Bedingungen erfüllt ist:

D: Das Modell zeichnet den aktuellen Zustand als akzeptierend aus.

E: Die Zielvorgaben für diese Sequenz sind erfüllt.

F: Die Strategie rät nicht zur Fortsetzung.

Zusammengefasst ergibt sich für das innere Abbruchkriterium K_i also folgende zusammengesetzte Aussage:

$$K_i = A \vee B \vee C \vee (D \wedge E \wedge F)$$

Eine Sequenz wird beendet, wenn die Zielvorgaben nicht erfüllt werden können, das Modell keine Elemente liefert, die Strategie es erzwingt oder der aktuelle Zustand akzeptierend ist, die Sequenz die Zielvorgaben erfüllt und die Strategie kein Fortsetzen empfiehlt.

2.9 Äußeres Abbruchkriterium

Das äußere Abbruchkriterium beendet, wenn es erfüllt ist, die Generierungssitzung ganz. Es wird anschließend keine weitere Sequenz erzeugt. Das ist der Fall, wenn eine der folgenden drei Bedingungen erfüllt ist:

G: Die Strategie erzwingt den Abbruch der Sitzung.

H: Die Zielvorgaben für diese Sitzung sind erfüllt.

I: Die Zielvorgaben für diese Sitzung können nicht erfüllt werden.

Das ergibt

$$K_a = G \vee H \vee I$$

für das äußere Abbruchkriterium K_a. Die Generierungssitzung wird beendet, wenn die Strategie es veranlasst oder die Zielvorgaben entweder erfüllt oder nicht zu erreichen sind.

2.10 Brauchbarkeit von Sequenzen

Am Ende jeder Sequenz wird geprüft, ob sie für die Ausgabe zu gebrauchen ist. Hinter dieser nebulösen Formulierung verbirgt sich die Forderung, dass nach der Fertigstellung einer Sequenz das Modell den Zustand des Generators als akzeptierend auszeichnen und die Sequenz einen Beitrag zu den Zielvorgaben leisten soll. Ist eines nicht der Fall, wird die Sequenz verworfen.

Kapitel 3

Der generische Sequenzgenerator

3.1 Überblick

Nach den konzeptionellen Vorbereitungen folgt nun die Implementierung des Sequenzgenerators in Java. Bevor ab dem dritten Abschnitt die Klassen und Interfaces einzeln betrachtet werden, bietet der zweite bereits einen Blick auf das Gesamtbild. Das Kapitel endet mit einigen einfachen Beispielen.

3.2 Architektur und Entwurf

Die in Kapitel 2 diskutierten Bestandteile des Sequenzgenerators *Generator*, *Modell*, *Strategie* und *Zielvorgaben* werden eins zu eins nach Java in die Klasse `Generator<S,E>`[1] und die Interfaces `IModel<S,E>`, `IStrategy<S,E>` und `IObjectives<S,E>` übertragen. Sequenzen werden von `java.util.List<E>`-Objekten repräsentiert und der Container für die Sequenzen durch ein eigenes Interface `IContainer<E>`, das `java.util.Collection<List<E>>` erweitert.

Die generischen Typparameter `S` und `E` stehen für Klassen, die die Zustände (State) und die Elemente repräsentieren.

Das Klassendiagramm in Abbildung 3.1 zeigt den fertigen Entwurf mit allen Klassen, Interfaces und ihren Methoden und Feldern. Der Code dazu ist in Anhang A zu finden.

Es fällt auf, dass `Generator` die einzige Klasse ist. Alles andere sind Interfaces, die zwar teilweise Default-Implementierungen einiger ihrer Methoden enthalten, aber sonst keinerlei Verhalten implementieren. Auf diese Weise

[1] Wie üblich werden an die englische Sprache angelehnte Bezeichner verwendet. Das führende „`I`“ unterscheidet Interface von Klassen.

wird eine hohe Konfigurierbarkeit der Sequenzgeneratoren erreicht. Implementierungen dieser Interfaces sind unter Berücksichtigung der generischen Typparameter nach Belieben austauschbar und somit Modelle, Strategien und Zielvorgaben weitgehend frei kombinierbar.

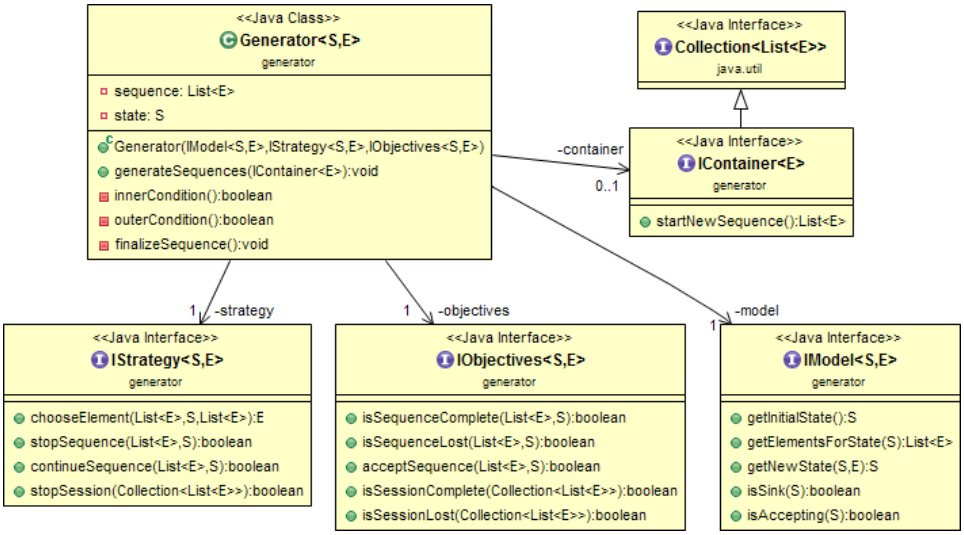

Abbildung 3.1: Klassendiagramm des Sequenzgenerators

3.3 Das Interface `IModel`

Implementierungen von `IModel` übernehmen im Generator die Rolle des Modells und bieten Zugriff auf Elemente und Zustände. Die Methode `getInitialState()` liefert den Startzustand, `getElementsForState(S)` die für einen Zustand verwendbaren Elemente und `getNewState(S, E)` den Zustand, der aus einem Zustand und dem daraus verwendeten Element folgt. Um vorauszusagen, ob das Modell für einen Zustand Elemente liefert, wird `isSink(S)` verwendet. Die Defaultimplementierung gibt `false` zurück. Ihr Einfluss auf das innere Abbruchkriterium wird damit neutralisiert. Die letzte Methode `isAccepting(S)` entscheidet, ob ein Zustand akzeptierend ist. Per Default `true` wird auch hier ein neutrales Verhalten im äußeren Abbruchkriterium und bei der Bewertung der Brauchbarkeit einer Sequenz erreicht.

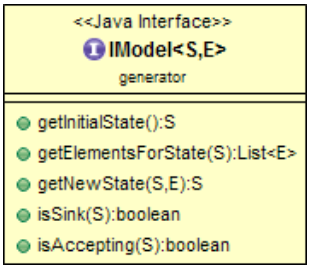

Abb. 3.2: `IModel`

3.4 Das Interface IStrategy

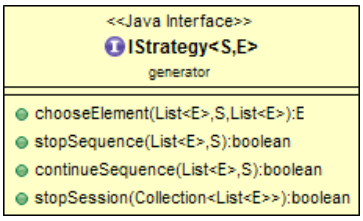

Abb. 3.3: IStrategy

IStrategy-Objekte repräsentieren die Strategie ihres Generators. In der Methode chooseElement(List, S, List) wird aus einer Liste von Elementen dasjenige ausgewählt, welches zur Sequenz hinzugefügt werden soll. Wenn keine Wahl getroffen wurde, wird die Konstante null zurückgegeben. Die Sequenz wird dann beendet. stopSequence(List, S) und stopSession(Collection) beenden eine Sequenz oder die ganze Generierungssitzung wenn sie true liefern. continueSequence(List, S) empfiehlt, falls true, das Fortsetzen einer Sequenz. Diese letzten drei Methoden liefern per Default false und halten sich damit aus den Abbruchkriterien heraus.

3.5 Das Interface IObjective

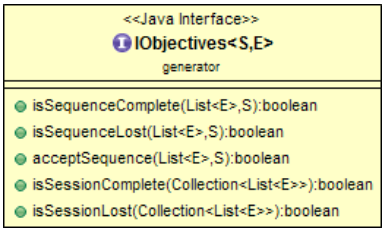

Abb. 3.4: IObjectives

IObjectives übernimmt die Rolle der Zielvorgaben für die Generierung. Die Methode isSequenceComplete(List, S) gibt vor jeder Erweiterung einer Sequenz an, ob sie bereits diese Vorgaben erfüllt. Dagegen liefert isSequenceLost(List, S) true, wenn erkannt wird, dass eine Sequenz die Vorgaben nicht mehr erreichen kann. acceptSequence(List, S) wird aufgerufen, um zu entscheiden, ob eine beendete Sequenz akzeptiert wird oder verworfen werden sollte. Zwischen der Generierung zweier Sequenzen wird mit isSessionComplete(Collection) und isSessionLost(Collection) geprüft, ob die Vorgaben bezüglich der insgesamt zu generierenden Sequenzen erfüllt oder noch zu erreichen sind.

3.6 Das Interface IContainer

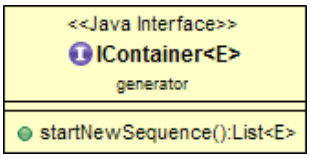

Abb. 3.5: IContainer

Ein IContainer ist der Container, in dem der Generator die fertigen und brauchbaren Sequenzen ablegt. Außerdem startet er dort mit Hilfe von startNewSequence() neue Sequenzen. Das hat den Vorteil, dass verschiedene Implementierungen von java.util.List verwendet werden können. Wofür das gut ist, wird im Beispiel in Abschnitt 4.2 deutlich.

3.7 Die Klasse `Generator`

Die Klasse `Generator` schließlich entspricht dem *Generator* aus Kapitel 2. Im Konstruktor werden `IModel`, `IStrategy` und `IObjectives` einmalig übergeben und können nicht mehr ausgetauscht werden. Das bedeutet, dass so ein `Generator` grundsätzlich nicht für Mehrfachverwendung ausgelegt ist. Für jede Generierung muss ein neues `Generator`-Objekt erzeugt werden.

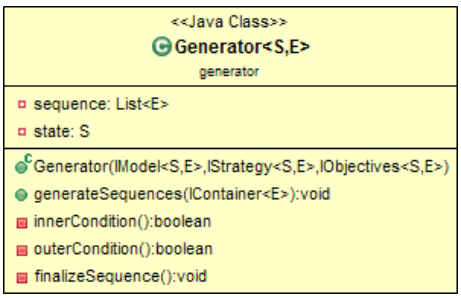

Abb. 3.6: `Generator`

Der in Listing 2.2 beschriebene Algorithmus ist hier in der öffentlichen Methode `generateSequences(IContainer)` implementiert. Als Parameter wird der Container übergeben, in dem die generierten Sequenzen abgelegt werden. Der Rückgabetyp ist `void`, weil dem Aufrufer das `IContainer`-Objekt bereits bekannt ist. Die beiden Abbruchkriterien sind in die Methoden `innerCondition()` und `outerCondition()` ausgelagert. Ihre Implementierungen entsprechen den Formeln in Abschnitt 2.8 und 2.9. Wenn die innere Schleife terminiert ist, wird in `finalizeSequence()` die generierte Sequenz, wie in Abschnitt 2.10 beschrieben, auf Brauchbarkeit geprüft und gegebenenfalls im Container abgelegt.

3.8 Veranschaulichende Beispiele

3.8.1 Beispiel 1: zufällige Sequenzen bestimmter Länge und Zahl

Das erste Beispiel zeigt einen Generator, der eine bestimmte Anzahl Sequenzen von bestimmter Länge aus `java.lang.Character`-Objekten - den Buchstaben `A`, `B` und `C` - erzeugt. Als Modell wird die `IModel`-Implementierung `SimpleUnconstrainedModel` verwendet. Mit dem Konstruktor `SimpleUnconstrainedModel(Element...)` wird das Modell mit einer Zahl an Elementen initialisiert. Durch `getElementsForState()` werden sie komplett als Liste zurückgegeben. `SimpleUnconstrainedModel` ist zustandlos, das heißt, `getInitialState()` und `getNewState()` liefern immer dieselbe Konstante. Die verwendete Strategie `SimpleRandomizedStrategy` wählt in `chooseElementForState()` per Zufall eines aus den übergebenen Elementen aus und gibt es zurück. `MinLengthAndCountObjectives` überschreibt die Methoden `isSequenceComplete()` und `isSessionComplete()` von `IObjectives` so, dass sie `true` liefern, sobald eine Sequenz eine Mindestlänge bzw. die Zahl der Sequenzen eine Mindestgröße erreicht hat.

Da lediglich `MinLengthAndCountObjectives` Methoden, die die Abbruchkriterien des Generators beeinflussen, überschreibt, werden Länge und Anzahl der Sequenzen ausschließlich dort geregelt. Modell und Strategie haben keinen Einfluss darauf.

In Klasse `Beispiel1` werden je ein `SimpleUnconstrainedModel`, `SimpleRandomizedStrategy` und `MinLengthAndCountObjectives` erzeugt. Ersterem werden `A`, `B` und `C` im Konstruktor übergeben, Letzterem die Parameter `4` als gewünschte Sequenzlänge und `10` als Zahl der zu generierenden Sequenzen. Anschließend werden diese drei Objekte dem `Generator`-Konstruktor übergeben. Auf dem so erzeugten `Generator`-Objekt wird die Methode `generateSequences()` aufgerufen. Als Container dient ein `SetOfLinkedListsContainer`, eine `IContainer`-Implementierung, die `java.util.HashSet` erweitert und für die Sequenzen `LinkedList`-Objekte verwendet.

Nach der Generierung wird der Inhalt des Containers über die Standardausgabe zur Kontrolle ausgegeben. Erwartungsgemäß erscheinen zehn Sequenzen der Länge vier, zusammengesetzt aus den Buchstaben `A`, `B` und `C`.

3.8.2 Beispiel 2: Deterministischer endlicher Automat „Geordnete Wörter“

Ein deterministischer endlicher Automat (DEA) ist ein Tupel $(Q, \Sigma, \delta, q_0, F)$, wobei Q eine endliche Menge von Zuständen, Σ ein Eingabealphabet, $\delta : Q \times \Sigma \to Q$ eine Übergangsfunktion, $q_0 \in Q$ ein Startzustand und schließlich $F \subset Q$ eine Menge von sogenannten akzeptierenden Zuständen sind. Beginnend mit dem Startzustand q_0 führen geeignete Eingaben von Symbolen aus Σ entsprechend der Funktion δ in andere Zustände aus Q. Ein Folge von Symbolen wird als Wort bezeichnet. Endet der Automat nach Eingabe eines Wortes in einem akzeptierenden Zustand aus F, akzeptiert der Automat dieses Wort. Die Menge aller Wörter, die ein DEA akzeptiert, nennt man seine Sprache. Mehr zu deterministischen endlichen Automaten findet sich in [7].

Für das zweite Beispiel wird als Modell folgender deterministischer endlicher Automat $D = (Q, \Sigma, \delta, q_0, F)$ verwendet:

$$
\begin{aligned}
\Sigma = &\quad \{A, \ldots, Z\} \\
Q = &\quad \Sigma + \{@\} \\
\delta : &\quad \{(q, a) \in (Q \times \Sigma) | a > q\} \to Q \\
&\quad (q, a) \mapsto a \\
q_0 = &\quad @ \\
F = &\quad \Sigma
\end{aligned}
$$

Die in der dritten Zeile verwendete Ordnungsrelation $>$ entspricht der Ordnung der Symbole in der ASCII-Zeichenkodierung. Die Buchstaben sind also natürlich geordnet und es gilt $A > @$. D akzeptiert alle Wörter aus $\{w \in \Sigma^* | \forall i, j \in \mathbb{N} \setminus \{0\} : i < j \leq |w| \Rightarrow w_i < w_j\}$. Seine Sprache besteht also aus Wörtern verschiedener, von links nach rechts alphabetisch sortierter Buchstaben.

Der Generator soll eine Menge von Sequenzen generieren, die solchen Wörtern entsprechen. Außerdem soll jeder Buchstabe in genau einer Sequenz vorkommen. Da im Modell Elemente und Zustände gleichartig sind, wird `Character` für beide Typparameter verwendet.

Die `IModel`-Implementierung, die D abbildet, liefert als Startzustand den `Character '@'` und als Elemente für einen Zustand alle Buchstaben, deren `Character` größer ist als derjenige, der diesen Zustand repräsentiert. `getNewState()` gibt als neuen Zustand den zuletzt verwendeten `Character` zurück.

Als Strategie wird eine Erweiterung der `SimpleRandomizedStrategy` aus Beispiel 1 verwendet, die `ExplorativeRandomizedStrategy`. Diese Strategie hält zwei Sets `elementsUsed` und `elementsFound`. Beide werden in Methode `chooseElement()` gepflegt. In `elementsUsed` werden die Elemente gespeichert, die die Methode bereits zurückgegeben hat, in `elementsFound` diejenigen, die bisher zur Auswahl standen. Bevor aus den vom Generator gelieferten Möglichkeiten ein Element ausgewählt wird, werden alle bereits verwendeten, also die aus `elementsUsed`, aussortiert. Auf diese Weise wird kein Element mehrfach verwendet. `continueSequence()` liefert immer `true` und `stopSession()` immer dann, wenn alle gefundenen Elemente verwendet wurden. Um zu verhindern, dass das äußere Abbruchkriterium bereits beim ersten Aufruf die Generierung beendet, gibt es das Flag `triedOneStep`, das dafür sorgt, dass die Schleife wenigstens einmal durchlaufen wird.

Eine spezielle `IObjectives`-Klasse gibt es nicht. Die Default-Implementierungen der Methoden im Interface reichen in diesem Beispiel aus. Länge und Anzahl der Sequenzen werden fast ausschließlich durch die Strategie gesteuert. Nur wenn eine Sequenz auf `Z` endet, bestätigt das Modell mittels `isSink()` das innere Abbruchkriterium. In allen anderen Fällen endet eine Sequenz, wenn die Strategie kein Element mehr liefert. Wegen der Default-Implementierungen in `IObjectives` obliegt es auch allein der Strategie, die Generierung als Ganzes zu beenden. Das ist genau dann der Fall, wenn alle Buchstaben einmal verwendet wurden.

Zu bemerken ist, dass `ExplorativeRandomizedStrategy` in seiner Implementierung auf das hier verwendete Modell abgestimmt ist. Für weniger vorhersehbare Modelle ist sie zu naiv und wird zu unzureichenden Ergebnissen oder zu Endlosschleifen führen.

In Klasse `Beispiel2` wird ein `Generator` mit Objekten der beschriebenen Klassen instanziiert. Als Container dient wie im vorigen Beispiel ein `SetOfLinkedListsContainer`. Nach Ende der Generierung werden Sequenzen mit allen Buchstaben ausgegeben und kein Buchstabe kommt mehrfach vor.

3.8.3 Beispiel 3: parabolische Irrfahrt in $\mathbb{Z}$

Das Modell im letzten Beispiel dieses Kapitels ist eine homogene Markov-Kette, und zwar eine Anlehnung an einen Random Walk durch die ganzen Zahlen $\mathbb{Z}$. Eine Markov-Kette ist ein stochastischer Prozess, also eine Folge von Zufallsvariablen $(X_t)_{t\in\mathbb{N}}$ mit Werten in einer Zustandsmenge Q für die gilt:

$$P(X_t = q_t | X_0 = q_0, \ldots, X_{t-1} = q_{t-1}) = P(X_t = q_t | X_{t-1} = q_{t-1}).$$

Das heißt, die Übergangswahrscheinlichkeit von einem Zustand q_{t-1} in einen anderen Zustand q_t hängt (nur) von q_{t-1} ab. Mehr über Markov-Ketten ist in [8] zu finden.

Die Zustände werden in diesem Beispiel durch die ganzen Zahlen repräsentiert. Startend bei 0 bewegt man sich in jedem Schritt zufällig zur nächstgrößeren oder -kleineren ganzen Zahl. Anders als beim Random Walk ist die Wahrscheinlichkeitsverteilung der Zufallsvariablen $(X_t)_{t\in\mathbb{N}}$ aber nicht für alle t gleich, sondern hängt von der aktuellen Position ab. X_t ist dabei die ganze Zahl, die nach t Schritten erreicht ist und hat folgende Wahrscheinlichkeitsverteilung:

$$P(X_t = m | X_{t-1} = n) = \begin{cases} \frac{|n|+1}{|n|+2}, & \text{falls } m = |n| - \operatorname{sgn} n \\ \frac{1}{|n|+2}, & \text{falls } m = |n| + \operatorname{sgn} n \\ 0, & \text{sonst.} \end{cases}$$

In Abbildung 3.7 spiegelt sich diese Verteilung in den Übergangswahrscheinlichkeiten zwischen den Zuständen wider. Bei 0 sind die Wahrscheinlichkeiten im nächsten Schritt nach links zur -1 oder nach rechts zur 1 zu gehen gleich, wohingegen es in Zustand 1 mit $\frac{2}{3}$ wahrscheinlicher ist, zur 0 zurückzukehren, als mit $\frac{1}{3}$ zur 2 fortzuschreiten.

Um das häufig verwendete Bild des „Weges des Betrunkenen“ zu bemühen, bewegt sich der Betrunkene bei einem Random Walk in einer Ebene. Im symmetrischen Fall ist diese Ebene horizontal. Die hier verwendete Konstruktion versetzt den Akteur aber in eine liegende Parabolschüssel. Je weiter er sich von der Mitte entfernt, desto stärker wird das Gefälle dorthin zurück und desto wahrscheinlicher wird ein Schritt bergab. Reduziert auf den eindimensionalen Fall mit konstant großen Schritten entspricht dieses

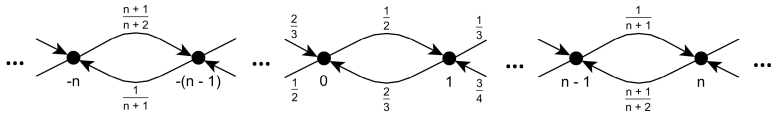

Abbildung 3.7: Markov-Kette „parabolische Irrfahrt in $\mathbb{Z}$“

Bild der beschriebenen Markov-Kette. Der Generator soll dieses Verhalten simulieren und entsprechende Sequenzen aus ganzen Zahlen erzeugen. Insbesondere sind hier nur solche Sequenzen interessant, die auf 0 enden, mindestens die Länge 10 und höchstens die Länge 30 haben.

Im Gegensatz zu den anderen Beispielen, wo sich die Strategie zu Variationszwecken des Zufalls bediente, ist der Zufall hier eine Eigenschaft des Modells. Welches Element als nächstes an die Sequenz angefügt wird, ist damit keine strategische Entscheidung. Um diesen Umstand abzubilden, werden für die Zustände Objekte der `java.util.concurrent.Callable`-Implementierung `Model3.CallableState` verwendet. Sie enthalten neben der ganzen Zahl, die sie repräsentieren, in ihrer `call()`-Methode die zufällige Berechnung des nächsten Schrittes mit der passenden Wahrscheinlichkeitsverteilung. Die Elemente für die Sequenzen sind `Integer`-Objekte.

Es wird ein Modell `Model3` verwendet, das als Startzustand einen `Model3.CallableState` für die Zahl 0 zurückgibt. `getElementsForState()` liefert die zwei `Integer` vor und nach dem vom übergebenen `CallableState` repräsentierten. Der neue Zustand ergibt sich immer aus dem zuletzt verwendeten Element, also ein `CallableState`, der den letzten Integer enthält. Akzeptierende Zustände sind nur solche, die eine 0 enthalten. Der Forderung nach Sequenzen, die auf 0 enden, wird also an dieser Stellen nachgekommen.

Als Strategie kommt folgerichtig eine `CallStateStrategy` zum Einsatz, deren Methode `chooseElement()` lediglich die `call()`-Methode des übergebenen Zustandes aufruft und deren Ergebnis zurückgibt. Ansonsten nimmt diese Strategie keinen Einfluss auf die Generierung.

Für die Zielvorgaben wird die Klasse `MinLengthAndCountObjectives` aus Beispiel 1 in `MinMaxLengthAndMinCountObjectives` erweitert. Es kann nun neben einer minimalen auch eine maximale Länge für die Sequenzen angegeben werden. Wird diese überschritten, liefern `isSequenceLost()` `true` und `isAccepting()` `false`.

Der `Generator` in `Beispiel3` bekommt in seinem Konstruktor ein `Model3`, eine `CallStateStrategy` und ein `MinMaxLengthAndMinCountObjectives` übergeben. Die Zielvorgaben sind auf eine minimale Sequenzlänge von 10, eine maximale Sequenzlänge von 30 und eine Zahl von 10 Sequenzen konfiguriert. Als Container kommt abermals ein `SetOfLinkedListsContainer` zum Einsatz.

Ein zweiter Generator wird mit dem gleichem Modell und den gleichen Zielvorgaben jedoch mit einer anderen Strategie, nämlich der aus Beispiel 1 bekannten `SimpleRandomizedStrategy`, konfiguriert. Dieser Generator simuliert einen symmetrischen Random Walk.

Einmal gestartet, erzeugen die Generatoren Sequenzen mit den beschriebenen Eigenschaften. Wird in einer Sequenz der Zustand 0 vor einer Länge von 10 erreicht, sorgt die Konjunktion von `Model3.isAccepting()` und `MinMaxLengthAndMinCountObjectives.isSequenceComplete()` im inneren Abbruchkriterium dafür, dass die Sequenz fortgesetzt wird. Wird dieser Zustand erst jenseits der Länge 30 erreicht, endet die Sequenz zwar, wird aber in `finalizeSequence()` durch den Einfluss von Methode `acceptSequence()` der `MinMaxLengthAndMinCountObjectives` verworfen. Nur Sequenzen, die bei einer Länge zwischen 10 und 30 im Zustand 0 landen, werden beendet und im Container abgelegt. Die Ausgabe zeigt je zehn Sequenzen, die die Erwartungen erfüllen. Die durch den zweiten Generator erzeugten Sequenzen sind im Schnitt etwas länger, was ebenfalls den Erwartungen an die verwendeten Strategien entspricht.

Kapitel 4

Komplexe, praxisrelevante Beispiele

4.1 Überblick

Die Beispiele in den folgenden beiden Abschnitten sollen zeigen, dass sich mit dem generischen Sequenzgenerator auch komplexe, praxisrelevante Generatoren implementieren lassen. Abschnitt 4.2 präsentiert eine Variante für Online-Testing und in Abschnitt 4.3 wird die Verwendung im Steplistgenerator von TPT besprochen. In beiden Fällen wird nicht soweit ins Detail gegangen, wie in den vorigen Beispielen. Es soll lediglich gezeigt werden, was möglich ist und wie die Umsetzung grundsätzlich aussieht.

4.2 Beispiel 4: Online-Testing-Algorithmus nach T. Kanstrén und M. Chechik

In ihrem Artikel [6] umschreiben die Autoren den Begriff *Online-Testing* folgendermaßen:

> Online testing interleaves test generation with test execution. Once the generator chooses a test step, it immediately executes it against the SUT; once the step has finished executing, it chooses the next step, etc.

Sie schlagen vor, die Zeit, die das Ausführen eines Testschrittes dauert, zu nutzen, um parallel dazu alle Möglichkeiten, die das Modell zur Fortsetzung bietet, zu bewerten und die beste auszuwählen. Dank Mehrkernprozessoren und Multithreading kann beides gleichzeitig auf demselben Rechner geschehen. Ihr Algorithmus kann in [6] nachgelesen werden.

In diesem Abschnitt wird die Implementierung einer an den generischen Sequenzgenerator angepassten Version das Originalalgorithmus vorgestellt.

Bei der Generierung geschieht etwa Folgendes: Sobald eine Sequenz gestartet ist, stößt die Strategie einen parallelen Ausführungsstrang an, der alle Möglichkeiten für die ersten 2 Schritte mit Hilfe des Modells berechnet und bewertet. Jede Möglichkeit erhält eine Punktzahl, die widerspiegelt, wie groß der Beitrag einer solchen Sequenz zum Gesamtergebnis wäre, würde sie zur Menge der generierten Sequenzen hinzugefügt werden. Sind alle Möglichkeiten bewertet, werden die besten ausgewählt und deren Elemente für den ersten Schritt zur Fortsetzung der tatsächlichen Sequenz in Betracht gezogen. Sie werden anschließend dem Generator, genauer der Strategie, zur Verwendung angeboten. Ist die Strategie an der Reihe, ein Element zur Fortsetzung zu bestimmen, wählt sie per Zufall aus den angebotenen Elementen eines aus. Steht noch kein Angebot zur Verfügung, blockiert die Ausführung bis das der Fall ist. Ist dann ein Element ausgewählt, stößt die Strategie die parallele Untersuchung der nächsten zwei Schritte an, bevor sie schließlich das Element dem Generator übergibt.

Bei der Untersuchung der Fortsetzungsmöglichkeiten wird nach dem Master-Worker-Prinzip ebenfalls mit mehreren Threads gearbeitet. Für jede mögliche Fortsetzung wird ein Job erzeugt, dem die aktuelle Sequenz und diese Fortsetzung sowie die Zahl der Schritte, die dann noch zu erforschen sind, übergeben werden. Kommt der Job durch einen Worker zur Ausführung, wird erst eine Kopie der bisherigen Sequenz erzeugt, an diese Kopie das Fortsetzungselement angehängt, diese fortgesetzte Sequenz bewertet und das Ergebnis gespeichert. Falls die Untersuchung noch tiefer geht, werden neue Jobs für alle Möglichkeiten des nächsten Schritts erzeugt und zur Ausführung bereitgestellt. Ist der letzte Job erledigt, werden aus dem Gesamtergebnis die besten Fortsetzungen entnommen und der Strategie angeboten.

Die Bewertung der verschiedenen Möglichkeiten erfolgt anhand der verwendeten Elemente. In diesem Beispiel sollen bei der Generierung möglichst viele Elemente und Paare von Elementen Verwendung finden. Wird eine Sequenz bewertet, gibt es zwei Punkte für jedes Element und drei für jede Elementpaarung, die bisher noch nicht vorkamen. Diese Gewichtung entspricht derjenigen aus dem Original. Dort fließen noch viele andere Faktoren in die Bewertung ein, auf die hier aber der Einfachheit halber verzichtet wird.

Das verwendete Modell repräsentiert alle Wörter aus den Buchstaben A, B, C und D bis zu einer Länge von fünf. Abbildung 4.1 zeigt einen entsprechenden endlichen Automaten.

In der Umsetzung dieses Beispiels enthält die `IModel`-Implementierung den DEA aus Abbildung 4.1. Von `IObjectives` wird `isSequenceComplete()` so überschrieben, dass immer `false` geliefert wird. `acceptSequence()` akzeptiert alle nichtleeren Sequenzen.

Abb. 4.1: DEA mit der Sprache $L = \{w \in \{A, B, C, D\}^* : 0 < |w| <= 5\}$

Da es sich um eine Implementierung für Online-Testing handelt, die ausgewählten Elemente also zur sofortigen Weiterverarbeitung vorgesehen sind, spielt das Brauchbarkeitskriterium eine untergeordnete Rolle. Es aber sinnvoll, die generierten Sequenzen dennoch zu Dokumentations- oder Wiederverwendungszwecken zu erhalten. Modell und Zielvorgaben sorgen dafür, dass in `Generator.finalizeSequence()` alle nichtleeren Sequenzen im Container landen.

Als Container wird übrigens ein `ObservableContainer` verwendet, der `SetOfLinkedListsContainer` erweitert und der eine Listener-Schnittstelle `ContainerListener` anbietet, die es ermöglicht, den Generierungsfortschritt zu beobachten. Die Schnittstelle enthält die folgenden drei Methoden: `sequenceStarted(List<E> sequence)` wird in der `startSequence()`-Methode von `ObservableContainer` aufgerufen; das Hinzufügen einer Sequenz zum Containerinhalt wird durch `sequenceAdded(List<E> sequence)` verkündet; `elementAdded(E element, List<E> sequence)` informiert einen Listener darüber, dass gerade ein Element an eine Sequenz angehängt wurde.

Der interessante Teil dieses Generators liegt in der verwendeten `IStrategy`-Implementierung `Strategy4`. `Strategy4` hat eine innere Klasse `Explorer`, eine `java.lang.Runnable`-Implementierung, in deren Methode `run()` die Suche nach den besten Fortsetzungen für eine Sequenz implementiert ist. Des Weiteren implementiert die Klasse `Strategy4` das Interface `ObservableContainer.ContainerListener`, um mitzubekommen, wann der Generator eine neue Sequenz beginnt. Dann nämlich startet sie in `sequenceStarted()` einen neuen `Explorer` in einem eigenen Thread. Dieser fängt sofort mit seiner Suche an und legt das Ergebnis, eine Liste der besten Elemente, in einem eigenen `BarrierContainer` ab. Ein `BarrierContainer` ist im Prinzip ein Stack, mit genau einem Platz, der gleichzeitig `java.util.concurrent.CyclicBarrier`[9] erweitert und damit der Synchronisierung des `Generator`-Threads mit dem `Explorer`-Thread dient. Eine `CyclicBarrier` ist „A synchronization aid that allows a set of threads to all wait for each other to reach a common barrier point.“[9]. Im Konstruktor initialisiert man so eine `CyclicBarrier` mit der Zahl der Threads, die an einem Punkt im Programmablauf synchronisiert werden sollen.

Im vorliegenden Fall sind das zwei: der Thread, in dem der Generator läuft und derjenige des Explorers. Das Sequenzdiagramm in Abbildung 4.2 ver-

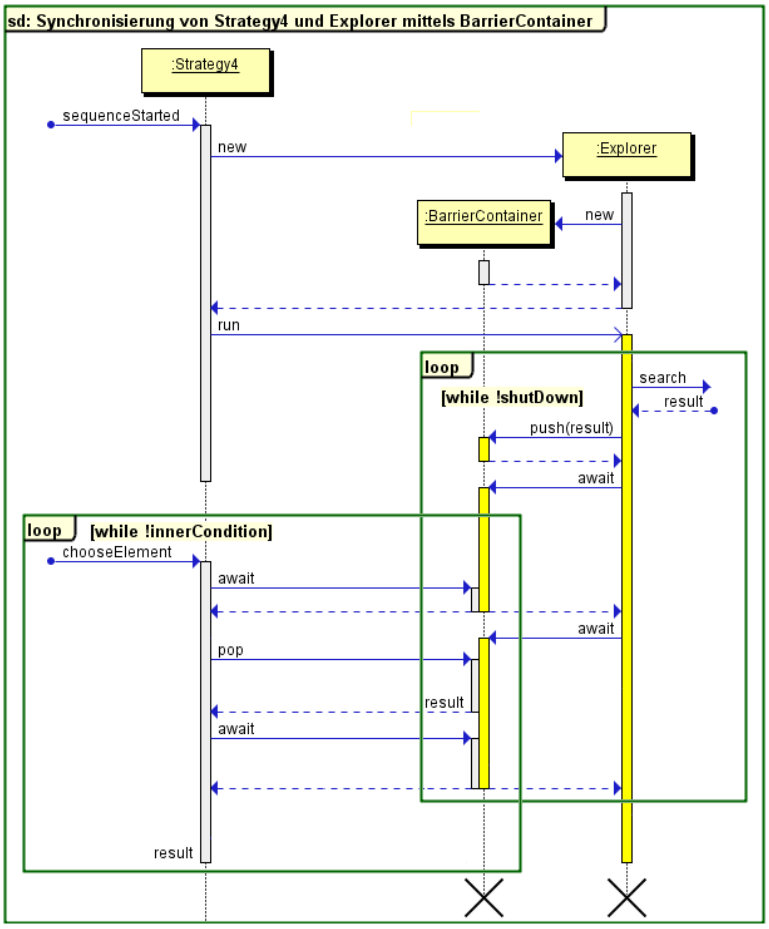

Abbildung 4.2: Sequenzdiagramm zur Synchronisierung von `Strategy4` und `Explorer` mit einem `BarrierContainer`

anschaulicht, wie sich `Strategy4` und ihr `Explorer` über diese Barrier synchronisieren. Sobald der `Explorer` mit der Untersuchung fertig ist und das Ergebnis im `BarrierContainer` abgelegt hat, ruft er die Methode `await()` auf dem `BarrierContainer` auf. Diese Methode blockiert solange die Ausführung, bis insgesamt zwei Threads an dieser Barrier warten. Der andere Thread, der an diese Barrier kommt, ist, wie gesagt, der, in dem der Generator läuft. Beim Aufruf von `Strategy4.chooseElement()` wird eingangs ebenfalls `await()` an dieser Barrier aufgerufen. Welcher von beiden Threads zuerst an die Barrier kommt, spielt keine Rolle. Nachdem beide angekommen sind, können sie ihre Ausführungen fortsetzen. Der Explorer wartet gleich wieder an der Barrier. Aber aus Sicht der Strategie ist nun sicher, dass im `BarrierContainer` das Ergebnis der Suche des Explorers, die Liste der besten Elemente zur Fortsetzung, zu finden ist. Sie holt diese Liste ab und wählt

ein Element zur Rückgabe an den Generator aus. Anschließend initialisiert sie den Explorer für die nächste Suche und ruft ihrerseits `await()` auf dem `BarrierContainer` auf. Wird die Barrier freigegeben, gibt sie das Element an den Generator und der Explorer startet die nächste Untersuchung des Models.

Die Jobs zur Untersuchung der nächsten Schritte sind in der inneren Klasse `Strategy4.Explorer.SearchJob` umgesetzt. Als `Runnable`-Implementierung werden sie einem `java.util.concurrent.ExecutorService`[10] als Task übergeben. Dieser `ExecutorService` hält mehrere Worker-Threads, die diese Tasks der Reihe nach ausführen. Wie ein solcher Job aussieht, wurde bereits erläutert. Jobs und Explorer synchronisieren sich über `wait()` und `notify()`, damit der Explorer erst mit der Auswertung der Ergebnisse beginnt, wenn alle Jobs beendet sind.

Die Umsetzung der Bewertung der Sequenzen und die Auswertung der Untersuchungsergebnisse findet hier keine weitere Betrachtung. Die Verwendung der entsprechenden Klassen erschließt sich aus dem Code von `Strategy4` in Abschnitt E.

4.3 Beispiel 5: Steplist-Generator in TPT

„TPT ist ein modellbasiertes Testwerkzeug für den Test eingebetteter Systeme, insbesondere für den Test von Steuerungs- und Regelungssystemen.“[3] Als Werkzeuge zur Testmodellierung stehen Zustandsautomaten und Schrittlisten (Steplists) zur Verfügung. Die Zustandsautomaten in TPT sind Super-Automaten in dem Sinne, dass sie für ihre Zustände und Zustandsübergänge mehrere Varianten unterstützen. So ein Super-Automat vereinigt also mit Super-Zuständen und Super-Transitionen mehrere strukturgleiche Zustandsautomaten in sich. Zur Verwaltung dieser Varianten werden *Testlets* verwendet, ein weiteres zentrales Konzept in TPT. Ein Testlet enthält entweder einen TPT-Automaten oder ein oder mehrere Schrittlisten. Im Fall von Automaten werden die Super-Zustände ebenfalls durch Testlets repräsentiert und auch in Schrittlisten können untergeordnete Testlets definiert werden, deren Varianten in Testlet-Steps Verwendung finden. In Abbildung 4.3 ist ein Beispiel für eine solche Verschachtelung von Testlets zu sehen. Mehr zu TPT ist in der Internetpräsenz des Herstellers[3] zu finden. Über Testlets und andere Konzepte von TPT kann man sich in [11] informieren.

Schrittlisten in TPT enthalten eine Reihe von Steps verschiedener Typen, die bei einer Testausführung nacheinander oder parallel ausgeführt werden. Der Steplist-Generator in seiner aktuellen Form erzeugt Schrittlisten aus Testlet- und Wait-Steps. In einem Testlet-Step ist die Variante eines Testlets, das in der Schrittliste definiert sein muss, angegeben. Ein Wait-Step[1]

[1] Es gibt drei Arten von Wait-Steps in TPT. Hier ist die „Wait ...“-Variante gemeint.

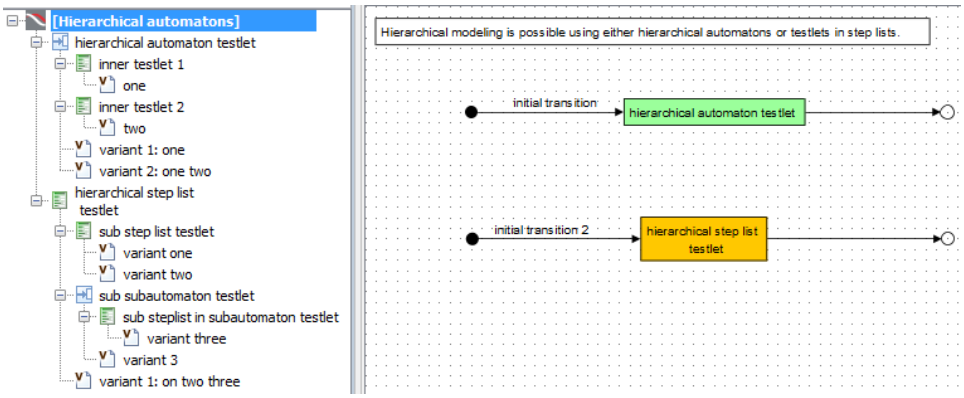

Abbildung 4.3: „Hierarchical automatons“-Example aus TPT 8 (Pre-Release)

enthält eine Transitionsbedingung. Die Ausführung wird an den folgenden Step übergeben, sobald sie erfüllt ist. Hinter den Testlet-Varianten verbergen sich konkrete Varianten von TPT-Automaten oder andere Schrittlisten. Im nächsten Abschnitt zeigt Abbildung 4.8 ein Beispiel einer solchen Schrittliste.

Der Sequenzgenerator in TPT verwendet erweiterte endliche Zustandsautomaten als Modell. Darin gibt es einen Startzustand, eine Reihe von benutzerdefinierten Zuständen und eine Menge von Zeichenkettenvariablen. Die benutzerdefinierten Zustände enthalten jeweils ein Paar aus einer Testlet-Variante und einer Wait-Bedingung und es lassen sich Guards, Actions und Folgezustände angeben. Guards sind Aussagen über die aktuelle Belegung der Zeichenkettenvariablen. Durch Actions kann diese Belegung geändert werden. Des Weiteren gibt es eine globale Abbruchbedingung, die, wenn sie erfüllt ist, jeden Zustandsübergang unterbindet. Der Übergang von einem Zustand q_1 in einen anderen q_2 ist genau dann möglich, wenn q_2 als Folgezustand in q_1 angegeben ist, die Guard von q_2 erfüllt und die globale Abbruchbedingung nicht erfüllt ist. Erfolgt dieser Übergang bei der Generierung, wird die Action von q_2 ausgeführt und q_2 selbst als Sequenzelement ausgegeben. Für den Startzustand lassen sich nur eine Action und Folgezustände angeben. Eine Zugangsbeschränkung durch eine Guard ist unnötig, da es keinen Zustandsübergang dorthin gibt. Das Modell selbst ist einem Testlet für Schrittlisten untergeordnet. Ausschließlich Testlets, die dort definiert sind, beziehungsweise deren Varianten können in seinen Zuständen verwendet werden.

Abbildung 4.4 zeigt beispielhaft einen einfachen und zum Teil schematischen erweiterten Zustandsautomaten, wie er im Steplist-Generator Verwendung findet. Die Zustände enthalten eine Testlet-Variante und eine Wait-

Bedingung. An den Übergängen stehen in eckigen Klammern jeweils eine Konjunktion aus globaler Abbruchbedingung und zustandsspezifischer Guard sowie hinter dem Schrägstrich die zustandsspezifische Action. Alle eingehenden Übergänge eines Zustandes haben die gleiche Guard und die gleiche Action.

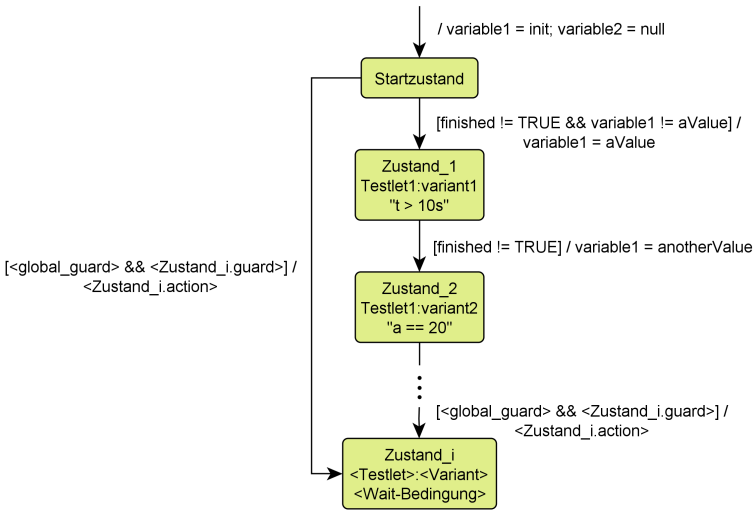

Abbildung 4.4: Prinzip der verwendeten erweiterten Zustandsautomaten

Der Benutzer gibt für die Generierung eine maximale Sequenzlänge und eine Kombinationslänge vor. Die Kombinationslänge k sagt aus, dass der Generator möglichst viele Variationen (mit Wiederholung) von k Elementen aus der Menge aller Zustände durch die generierten Sequenzen abdecken soll. Um den diesbezüglichen Fortschritt zu messen, wird ein Monitor verwendet, der die Generierung überwacht. Er gibt auch Auskunft darüber, ob und in welchem Maß eine Sequenz diese Abdeckung verbessert. Strategie und Zielvorgaben greifen auf diesen Monitor zu.

Die verwendete Strategie kann in einem ähnlichen Sinne wie jene in Beispiel 2 als explorativ bezeichnen werden. Sie erforscht sukzessive das Modell und optimiert mit den gewonnenen Informationen und mit Hilfe des Monitors die Wahl der jeweils nächsten Elemente.

Die Zielvorgaben kümmern sich vorrangig darum, dass die Sequenzlänge nicht die Benutzerangabe übersteigt. Des Weiteren werden Sequenzen, die laut Monitor keinen Beitrag zur Variationsabdeckung leisten, abgelehnt.

Da die generierten Sequenzen Folgen von Zuständen sind, bedarf es einer Nachbereitung des Generierungsergebnisses. Jeder Zustand enthält, wie bereits beschrieben, eine Testlet-Variante und eine Wait-Bedingung. Für jede Sequenz werden nun der Reihe nach von jedem Zustand die Testlet-Variante

in einen Testlet-Step und die Wait-Bedingung in einen Wait ...-Step geschrieben und diese Steps in einer Schrittliste gespeichert. Diese Schrittliste bildet dann eine neue Variante des dem Modell übergeordneten Testlets.

Die Implementierung dieses Generators sieht folgendermaßen aus:

Die beiden generischen Typparameter `S` und `E` sind gleich. Sie repräsentieren beide eine Implementierung der Zustände des erweiterten Zustandsautomaten im Modell. Die generierten Sequenzen sind also Folgen dieser Zustände.

Die Implementierung des Modells als `IModel` liefert mit `getInitialState()` den Startzustand und mit `getEle- mentsForState()` die gerade erlaubten Folgezustände. Dazu wird als erstes die globale Abbruchbedingung gegen die aktuelle Belegung der Zeichenkettenvariablen geprüft. Ist sie erfüllt, gibt die Methode eine leere Liste zurück. Falls nicht, werden alle Zustände, die als Folgezustand des übergebenen Zustandes gelten, in Betracht gezogen. Für sie alle wird die jeweilige Guard ebenfalls gegen die vorliegende Variablenbelegung geprüft und Zustände, deren Guard nicht erfüllt ist, aussortiert. `getNewState()` setzt das übergebene Element als neuen Zustand und verwendet dessen Action zur Aktualisierung der Zeichenkettenvariablen. `isSink()` und `isAccepting()` verbleiben in der Default-Implementierung, nehmen also keinen Einfluss auf die Generierung.

Die implementierte Strategie trifft in `chooseElement()` ihre Entscheidung auf Basis eigener Erkenntnisse über das Modell und der Informationen des Monitors über die gegenwärtige Abdeckung. Die übergebenen Parameter werden für die weitere Erforschung des Modells verwendet. Hält die Strategie das Modell für komplett erforscht und ist keine Verbesserung der Abdeckung mehr möglich, bricht sie die Generierung mit `stopSession()` ab.

In der `IObjectives`-Implementierung der Zielvorgaben gelten für `isSequenceComplete()` Sequenzen dann als fertig, wenn sie die maximale Länge erreicht haben. Liegen sie darüber, liefert `isSequenceLost()` `true`. Mit `acceptSequence()` werden Sequenzen nur akzeptiert, wenn sie die durch den Monitor überwachte Abdeckung verbessern. `isSessionComplete()` gibt `true` zurück, wenn der Monitor „volle Abdeckung“ vermeldet.

Der Monitor selbst ist im Prinzip einfach eine k-dimensionale Matrix mit identischen Dimensionslängen d, also ein k-Würfel, mit Einträgen aus $\{0, 1\}$. Dabei ist k die benutzerdefinierte Kombinationslänge und d die Anzahl der Zustände, exklusive des Startzustandes. Jeder Eintrag repräsentiert eine mögliche Variation von k Zuständen. Ist der Eintrag 0, kommt diese Variation noch in keiner Sequenz vor, ansonsten ist er 1. Als `java.util.Bitset` umgesetzt, lassen sich auf einfache Weise die nötigen Informationen gewinnen.

4.4 Anwendungsbeispiel „Fensterputzen“ in TPT

Um die Funktionsweise des Steplist-Generators zu demonstrieren, wird das folgende anschauliche Beispiel verwendet: Gegeben sei ein Fenster und ein Akteur. Das Fenster lässt sich öffnen und schließen. Der Akteur kann das Fenster innen und außen putzen sowie zum Fenster hinausschauen. Dabei müssen Regeln beachtet werden. Das Fenster kann innen nur geputzt werden, wenn es geschlossen und von innen verschmutzt ist. Analog geht das außen nur, wenn es offen und außen verschmutzt ist. Der Akteur kann nur hinausschauen, wenn das Fenster offen oder innen und außen sauber ist. Nachdem er das Fenster außen geputzt hat, schaut der Akteur niemals hinaus. Wenn der Akteur einmal hinausgeschaut hat, besteht keine Notwendigkeit mehr das Fenster zu putzen oder zu öffnen. Das Fenster ist anfangs geschlossen.

Abbildung 4.5 zeigt einen erweiterten endlichen Zustandsautomaten, der dieses Verhalten abbildet. Die Guards und die Actions wurden zur besseren Übersicht mit in die Zustände geschrieben. Das Funktionsprinzip ist aber dasselbe, wie in Abschnitt 4.3 beschrieben. Hervorzuheben ist die Schleife an Zustand 5. Dieser Zustandsübergang ist wegen der globalen Abbruchbedingung unmöglich.

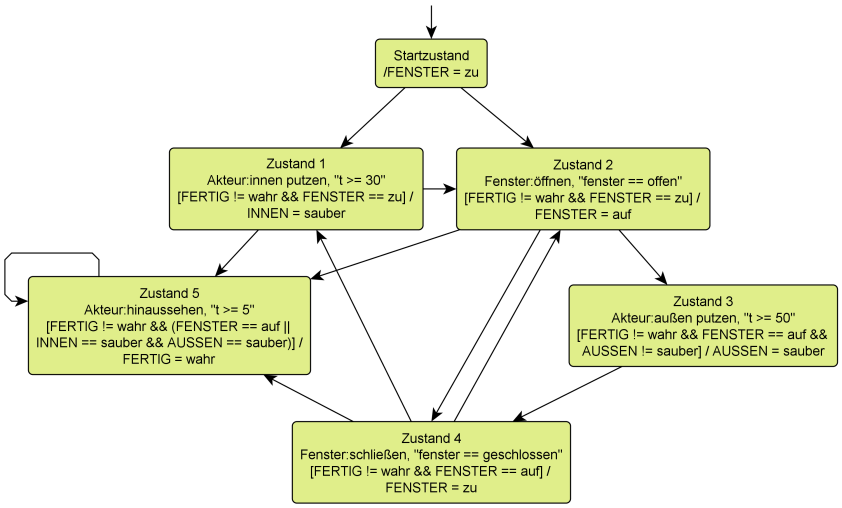

Abb. 4.5: Erweiterter endlicher Zustandsautomat „Fensterputzen“ als Graph

In TPT sind das Fenster und der Akteur sowie ihre Fähigkeiten in zwei Testlets und deren Varianten umgesetzt (siehe Abbildung 4.7). Die Konfiguration des Zustandsautomaten aus Abbildung 4.5 sieht in TPT wie in Abbildung 4.6 aus. Es gibt die vier Zeichenkettenvariablen `FENSTER`, `INNEN`, `AUSSEN` und `FERTIG`, die den erweiterten Zustand des Systems beschrei-

ben. Für `FENSTER` gibt es die Werte `"auf"` und `"zu"`, `INNEN` und `AUSSEN` sind entweder nicht gesetzt oder `"sauber"`. `FERTIG` kann nicht gesetzt oder `"wahr"` sein. Die Tabelle unter *Adjacency* im unteren Teil zeigt die Adjazenzmatrix des Graphen des Zustandsautomaten ohne Berücksichtigung des Startzustandes. Die Variable `FENSTER` wird initial auf `"zu"` gesetzt. Wenn `FERTIG` gleich `"wahr"` ist, endet die Generierung einer Sequenz. Beides ist im oberen Teil der Abbildung als *Initial Action* beziehungsweise *Termination Condition* zu finden. Die Tabelle in der Mitte der Abbildung zeigt fünf benutzerdefinierte Zustände, für jede Testlet-Variante einen. Nur die ersten beiden dürfen auf den impliziten Startzustand folgen. Die Angaben in den Spalten *Guard* und *Action* entsprechen denen aus Abbildung 4.5. Unter *Wait Condition* sind die für die Schrittlisten relevanten Wait-Bedingungen zu finden. t und $fenster$ sind TPT-Signale. Sie haben auf die Generierung keinen Einfluss.

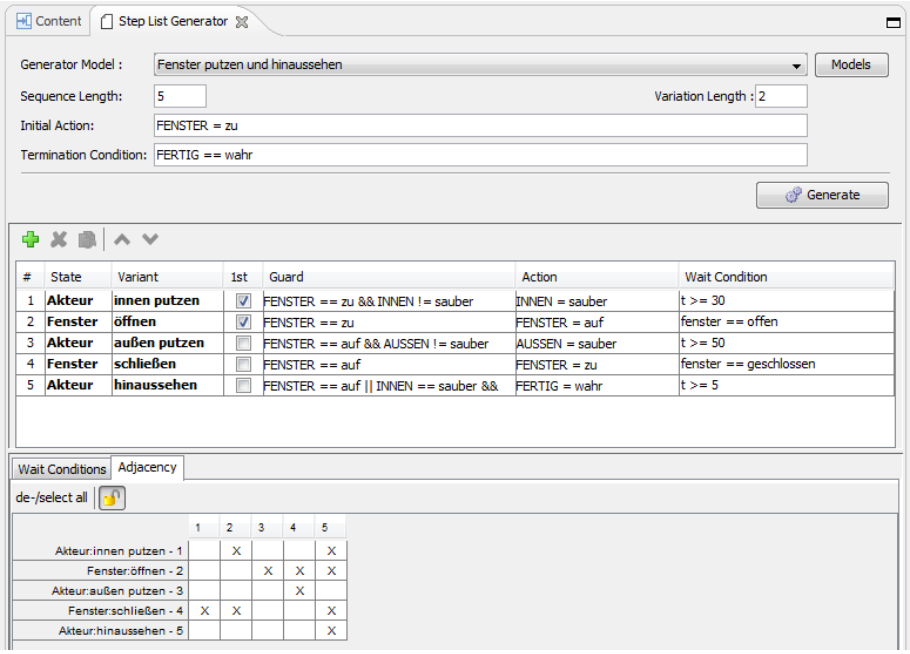

Abbildung 4.6: Modellierung des Automaten „Fensterputzen“ in TPT

Für die Generierung ist eine maximale Sequenzlänge von fünf vorgegeben und es sollen alle möglichen Variationen von zwei Zuständen vorkommen. Dadurch wird gewährleistet, dass jeder erlaubte Zustandsübergang während der Generierung berücksichtigt wird, vorausgesetzt, er ist in fünf Schritten erreichbar. Mit dem *Generate*-Button startet die Generierung und erzeugt die in Abbildung 4.7 unter *Test Cases* aufgeführten Schrittlisten. Ihre Namen spiegeln die verwendeten Testlet-Varianten in der Reihenfolge ihres Auf-

tretens in der jeweiligen Schrittliste wider. Es lässt sich leicht nachprüfen, dass, von der Schleife an Zustand 5 abgesehen, alle Übergänge der Adjazenzmatrix verwendet wurden. Verbotene Übergänge kommen nicht vor. Keine Sequenz ist länger als fünf Schritte und wenn *hinaussehen* vorkommt, endet die Schrittliste anschließend. Begonnen wird erwartungsgemäß immer mit *„öffnen“* oder *„innen putzen“*.

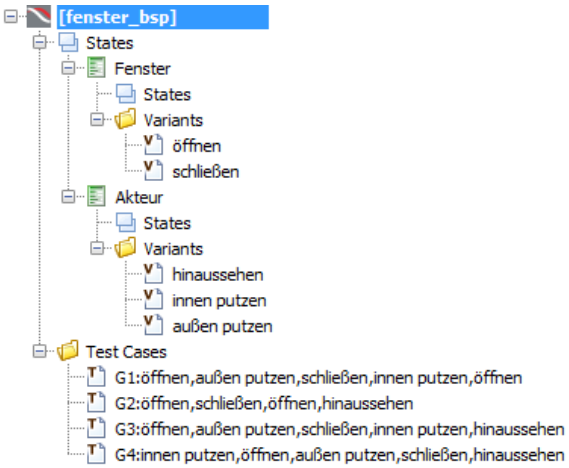

Abbildung 4.7: Testlets, Varianten und generierte Testfälle in TPT

In Abbildung 4.8 ist exemplarisch die Schrittliste mit dem Namenspräfix *G4* dargestellt. Sie lässt sich folgendermaßen lesen:

„Putze die Scheiben 30 Sekunden lang von innen und öffne dann das Fenster (solange bis es offen ist)! Putze anschließend 50 Sekunden lang die Scheiben von außen und schließe danach das Fenster wieder (solange bis es zu ist)! Schaue schließlich fünf Sekunden lang hinaus!“

1	Testlet	Akteur	: innen putzen
2	Wait ...	t >= 30	
3	Testlet	Fenster	: öffnen
4	Wait ...	fenster == offen	
5	Testlet	Akteur	: außen putzen
6	Wait ...	t >= 50	
7	Testlet	Fenster	: schließen
8	Wait ...	fenster == geschlossen	
9	Testlet	Akteur	: hinaussehen
10	Wait ...	t >= 5	

Abbildung 4.8: Eine generierte Schrittliste

Kapitel 5

Fazit

In den vorangegangenen Beispielen konnte gezeigt werden, dass sich der vorgestellte intuitive Ansatz des generischen Sequenzgenerators als Grundlage für konkrete Sequenzgeneratoren eignet. Es lassen sich verschiedene Modelltypen implementieren, existierende Modelle mit Adapter-Implementierungen von `IModel` anbinden und nach unterschiedlichen Strategien durchmustern. Komplexe Sachverhalte lassen sich hinter den angebotenen schlanken Schnittstellen verbergen. Die Verwendung von Interfaces und die Generizität sorgen für Flexibilität und Erweiterbarkeit. Die zweckmäßigen Rollen *Generator*, *Modell*, *Strategie* und *Zielvorgaben* erleichtern die Planung und die Umsetzung konkreter Generatoren. Außerdem lässt sich der generische Sequenzgenerator problemlos als „Standalone“ zum Experimentieren verwenden.

Zurückblickend auf die in Abschnitt 1.4 zitierten Aussagen aus [5] kann der Versuch, verschiedene Modelle und Strategien mit einem einfachen und intuitiven Algorithmus abzudecken, als Beitrag zum besseren Verständnis dieser Ansätze betrachtet werden. Die Idee hinter dem generischen Sequenzgenerator ist es, den Fokus auf die Sequenzgenerierung und weg von der Komplexität der Peripherie zu lenken. Diese Herangehensweise trägt auch zu einer besseren Integrationsfähigkeit von MBT-Methoden bei.

Kapitel 6

Ausblick

Es gibt immer etwas zu verbessern; und so ist es auch beim generischen Sequenzgenerator. Nicht alle Ideen wurden in die vorliegende Version eingearbeitet.

Zum Beispiel gibt der Generator kein Feedback darüber, ob eine Generierungssitzung erfolgreich abgschlossen wurde. In Analogie zu `IObjectives.acceptSequence()` könnte eine weitere Methode `IObjectives.acceptSession()` am Ende einer Generierung Auskunft geben, ob die Zielvorgaben erreicht wurden.

Das Umgehen des inneren Abbruchkriteriums, wenn die Strategie kein Folgeelement liefert, sollte durch geeignete Maßnahmen vermieden werden. Es reicht vermutlich aus, den Ablauf der inneren Schleife so zu modifizieren, dass das Votum der Strategie bereits vor Prüfung des Kriteriums feststeht.

Der Steplist-Generator in TPT ist momentan ein sogenanntes Incubation-Feature. Das heißt, dass er zwar zum Installationsumfang gehört, die Benutzung aber wegen seiner experimentellen Natur auf eigene Gefahr erfolgt.

Entwicklungspotenzial gibt es dort bei der Modellierung der Zustandsautomaten. Hier ist eine grafische Repräsentation als Graph sowie Im- und Exportmöglichkeiten in Dateiformate, wie beispielsweise `GML` oder `GraphML` denkbar.

Des Weiteren sollte es eine detaillierte Rückmeldung über den Generierungserfolg geben: Welche Variationen wurden abgedeckt und wie oft kommen sie vor? Welche Übergänge im Graphen wurden nicht erreicht?

Eine nachträgliche Optimierung der generierten Sequenzen sollte in Betracht gezogen werden. Es kommt vor, dass früh in der Generierung erzeugte Sequenzen im weiteren Verlauf hinfällig werden, zum Beispiel wenn sie Präfix einer später erzeugten, längeren Sequenz sind. Solche Sequenzen könnten am Ende aussortiert werden, um die Menge der Testfälle zu reduzieren.

Danksagung

An dieser Stelle möchte ich mich bei Frau Prof. Dr. Ina Schieferdecker dafür bedanken, dass sie diese externe Arbeit unterstützt und ihre Betreuung übernommen hat.

Dr. Eckard Bringmann und Dr. Jens Lüdemann, meinen Chefs und Betreuern bei PikeTec, danke ich für ihr Vertrauen, für Geduld und Rat und vor allem für die dringend benötigte moralische Unterstützung auf den letzten Metern.

An Prof. Dr. Katinka Wolter geht ein Dank für die sehr kurzfristige Übernahme des Amtes der Zweitgutachterin.

Ich hoffe, mit dieser Arbeit einen langen und interessanten zweiten Bildungsweg zu einem guten Abschluss zu bringen.

Anhang A

Sequenzgenerator - Alle Klassen und Interfaces

Generator

```
5  public class Generator<S, E> {
6
7    private IModel<S, E> model;
8
9    private IStrategy<S, E> strategy;
10
11   private IObjectives<S, E> objectives;
12
13   private IContainer<E> container;
14
15   private List<E> sequence;
16
17   private S state;
18
19   public Generator(IModel<S, E> model, IStrategy<S, E>
         strategy, IObjectives<S, E> objective) {
20     super();
21     this.model = model;
22     this.strategy = strategy;
23     this.objectives = objective;
24   }
25
26   public void generateSequences(IContainer<E>
         container) {
27     this.container = container;
28     while (!outerCondition()) {
29       sequence = container.startNewSequence();
30       state = model.getInitialState();
31       while (!innerCondition()) {
32         List<E> elements = model.getElementsForState(
               state);
33         E element = strategy.chooseElement(sequence,
               state, elements);
34         if (element == null) {
35           break;
36         }
37         sequence.add(element);
38         state = model.getNewState(state, element);
39       }
40       finalizeSequence();
41     }
42   }
43
44   private boolean innerCondition() {
45     return objectives.isSequenceLost(sequence, state)
           || model.isSink(state) || strategy.
           stopSequence(sequence, state) || (model.
           isAccepting(state) && objectives.
           isSequenceComplete(sequence, state) && !
           strategy.continueSequence(sequence, state));
46   }
47
48   private boolean outerCondition() {
49     return strategy.stopSession(container) ||
           objectives.isSessionComplete(container) ||
           objectives.isSessionLost(container);
50   }
51
52   private void finalizeSequence() {
53     if (model.isAccepting(state) && objectives.
           acceptSequence(sequence, state)) {
54       container.add(sequence);
55     }
56   }
57
58 }
```

IModel

```
5  public interface IModel<S, E> {
6
```

```
  S getInitialState();

  List<E> getElementsForState(S state);

  S getNewState(S state, E element);

  default boolean isSink(S state) {
    return false;
  }

  default boolean isAccepting(S state) {
    return true;
  }
}
```

IStrategy

```
public interface IStrategy<S, E> {

  E chooseElement(List<E> sequence, S state, List<E>
      elements);

  default boolean stopSequence(List<E> sequence, S
      state) {
    return false;
  }

  default boolean continueSequence(List<E> sequence, S
      state) {
    return false;
  }

  default boolean stopSession(Collection<List<E>>
      sequences) {
    return false;
  }

}
```

IObjectives

```
public interface IObjectives<S, E> {

  default boolean isSequenceComplete(List<E> sequence,
      S state) {
    return true;
  }

  default boolean isSequenceLost(List<E> sequence, S
      state) {
    return false;
  }

  default boolean acceptSequence(List<E> sequence, S
      state) {
    return true;
  }

  default boolean isSessionComplete(Collection<List<E
      >> sequences) {
    return false;
  }

  default boolean isSessionLost(Collection<List<E>>
      sequences) {
    return false;
  }

}
```

IContainer

```
6  public interface IContainer<E> extends Collection<List
       <E>> {
7
8    List<E> startNewSequence();
9
10 }
```

Anhang B

Beispiel 1 - Alle Klassen

Beispiel1

```
14
15    public static void main(String[] args) {
16      IModel<Object, Character> model = new
            SimpleUnconstrainedModel<>('A', 'B', 'C');
17      IStrategy<Object, Character> strategy = new
            SimpleRandomizedStrategy<>();
18      IObjectives<Object, Character> objective = new
            MinLengthAndCountObjectives<>(4, 10);
19      Generator<Object, Character> generator = new
            Generator<>(model, strategy, objective);
20      IContainer<Character> sequenceContainer = new
            SetOfLinkedListsContainer<>();
21      generator.generateSequences(sequenceContainer);
22      System.out.println(sequenceContainer);
23    }
24
25  }
```

SimpleUnconstrainedModel

```
6   public class SimpleUnconstrainedModel<S, E> implements
        IModel<S, E> {
7
8     private final E[] elements;
9
10    public SimpleUnconstrainedModel(E... elements) {
11      this.elements = elements;
12    }
13
14    @Override
15    public S getInitialState() {
16      return null;
17    }
18
19    @Override
20    public final List<E> getElementsForState(S state) {
21      return Arrays.asList(elements);
22    }
23
24    @Override
25    public S getNewState(S state, E element) {
26      return getInitialState();
27    }
28
29  }
```

SimpleRandomizedStrategy

```
8     private Random rnd = new Random();
9
10    @Override
11    public E chooseElement(List<E> sequence, S state,
          List<E> elements) {
12      if (elements.isEmpty()) {
13        return null;
14      }
15      int index = rnd.nextInt(elements.size());
16      return elements.get(index);
17    }
18
19  }
```

MinLengthAndCountObjectives

```
6   public class MinLengthAndCountObjectives<S, E>
        implements IObjectives<S, E> {
7
8     private int minLength = 0;
```

```java

  private int minCount = 0;

  public MinLengthAndCountObjectives(int minLength,
      int minCount) {
    this.minLength = minLength;
    this.minCount = minCount;
  }

  @Override
  public boolean isSequenceComplete(List<E> sequence,
      S state) {
    return sequence.size() >= minLength;
  }

  @Override
  public boolean isSessionComplete(Collection<List<E>>
      sequences) {
    return sequences.size() >= minCount;
  }

}
```

SetOfLinkedListsContainer

```java
public class SetOfLinkedListsContainer<E> extends
    HashSet<List<E>> implements IContainer<E> {

  @Override
  public List<E> startNewSequence() {
    return new LinkedList<>();
  }

}
```

Anhang C

Beispiel 2 - Alle Klassen

Beispiel2

```
13
14    public static void main(String[] args) {
15      IModel<Character, Character> model = new Model2();
16      IStrategy<Character, Character> strategy = new
            ExplorativeRandomizedStrategy<>();
17      IObjectives<Character, Character> objective = new
            IObjectives<Character, Character>() {
18      };
19      Generator<Character, Character> generator = new
            Generator<>(model, strategy, objective);
20      IContainer<Character> sequenceContainer = new
            SetOfLinkedListsContainer<>();
21      generator.generateSequences(sequenceContainer);
22      System.out.println(sequenceContainer);
23    }
24
25  }
```

Model2

```
6   public class Model2 implements IModel<Character,
        Character> {
7
8     private final Character[] alphabet;
9
10    private final Character initialState = '@';
11
12    public Model2() {
13      alphabet = new Character[26];
14      for (char i = 'A'; i <= 'Z'; ++i) {
15        alphabet[i - 'A'] = i;
16      }
17    }
18
19    @Override
20    public Character getInitialState() {
21      return initialState;
22    }
23
24    @Override
25    public List<Character> getElementsForState(Character
          state) {
26      Character[] segment = Arrays.copyOfRange(alphabet,
            state - 'A' + 1, alphabet.length);
27      return Arrays.asList(segment);
28    }
29
30    @Override
31    public Character getNewState(Character state,
          Character element) {
32      return element;
33    }
34
35  }
```

ExplorativeRandomizedStrategy

```
9   public class ExplorativeRandomizedStrategy<S, E>
        extends SimpleRandomizedStrategy<S, E> {
10
11    private final Set<E> elementsFound = new HashSet<>()
          ;
12
13    private final Set<E> elementsUsed = new HashSet<>();
14
15    private boolean triedOneStep = false;
16
17    @Override
18    public E chooseElement(List<E> sequence, S state,
```

```
        List<E> elements) {
      triedOneStep = true;
      List<E> elementsCopy = new LinkedList<>(elements);
      elementsCopy.removeAll(elementsUsed);
      elementsFound.addAll(elementsCopy);
      E retVal = super.chooseElement(sequence, state,
          elementsCopy);
      if (retVal != null) {
        elementsUsed.add(retVal);
      }
      return retVal;
    }

    @Override
    public boolean continueSequence(List<E> sequence, S
        state) {
      return true;
    }

    @Override
    public boolean stopSession(Collection<List<E>>
        sequences) {
      return triedOneStep && elementsFound.size() ==
          elementsUsed.size();
    }

  }
```

Anhang D

Beispiel 3 - Alle Klassen

Beispiel3

```java

  public static void main(String[] args) {

    IModel<Model3.CallableState, Integer> model = new Model3();
    IStrategy<Model3.CallableState, Integer> strategyA = new CallStateStrategy<>();
    IStrategy<Model3.CallableState, Integer> strategyB = new SimpleRandomizedStrategy<>();
    IObjectives<Model3.CallableState, Integer> objectives = new MinMaxLengthAndMinCountObjectives<>(10, 30, 10);
    Generator<Model3.CallableState, Integer> generator1 = new Generator<>(model, strategyA, objectives);
    Generator<Model3.CallableState, Integer> generator2 = new Generator<>(model, strategyB, objectives);
    IContainer<Integer> sequenceContainer = new SetOfLinkedListsContainer<>();

    generator1.generateSequences(sequenceContainer);
    System.out.println("CallStateStrategy");
    System.out.println(sequenceContainer);
    System.out.println();

    sequenceContainer.clear();
    generator2.generateSequences(sequenceContainer);
    System.out.println("SimpleRandomizedStrategy");
    System.out.println(sequenceContainer);
  }

}
```

Model3

```java
public class Model3 implements IModel<CallableState, Integer> {

  private final CallableState initialState = new CallableState(0);

  @Override
  public CallableState getInitialState() {
    return initialState;
  }

  @Override
  public List<Integer> getElementsForState(CallableState state) {
    Integer stateInt = state.stateInt;
    return Arrays.asList(stateInt - 1, stateInt + 1);
  }

  @Override
  public CallableState getNewState(CallableState state, Integer element) {
    return new CallableState(element);
  }

  @Override
  public boolean isAccepting(CallableState state) {
    return state.stateInt == 0;
  }

  // ----------------------------------------

  public static class CallableState implements Callable<Integer> {

    private final static Random RND = new Random();
```

```java

    private final Integer stateInt;

    private CallableState(Integer stateInt) {
      this.stateInt = stateInt;
    }

    @Override
    public Integer call() throws Exception {
      if (stateInt == null) {
        return 0;
      }
      int rnd = RND.nextInt(Math.abs(stateInt) + 2);
      int step = stateInt < 0 ? 1 : -1;
      return rnd == 0 ? stateInt - step : stateInt +
          step;
    }

  }

}
```

CallStateStrategy

```java
public class CallStateStrategy<S extends Callable<E>,
    E> implements IStrategy<S, E> {

  @Override
  public E chooseElement(List<E> sequence, S state,
      List<E> elements) {
    try {
      return state.call();
    } catch (Exception e) {
      return null;
    }
  }

}
```

MinMaxLengthAndMinCountObjectives

```java
public class MinMaxLengthAndMinCountObjectives<S, E>
    extends MinLengthAndCountObjectives<S, E> {

  private int maxLength = Integer.MAX_VALUE;

  public MinMaxLengthAndMinCountObjectives(int
      minLength, int maxLength, int minCount) {
    super(minLength, minCount);
    this.maxLength = maxLength;
  }

  @Override
  public boolean isSequenceLost(List<E> sequence, S
      state) {
    return sequence.size() > maxLength;
  }

  @Override
  public boolean acceptSequence(List<E> sequence, S
      state) {
    return !isSequenceLost(sequence, state);
  }

}
```

Anhang E

Beispiel 4 - Ausgewählte Klassen

Strategy4

```
19 public class Strategy4 extends SimpleRandomizedStrategy<Integer, Character> implements ObservableContainer.ContainerListener<Character> {
20
21   // das Modell des Generators
22   private final IModel<Integer, Character> model;
23
24   // zur Bewertung der Sequenzen
25   private final SequenceRatingSet<Character> ratingSet;
26
27   // Anzahl der vorauszuberechnenden Schritte
28   private final int sneakDepth;
29
30   // Master-Worker-Einheit
31   private final ExecutorService executorService = Executors.newFixedThreadPool(4);
32
33   // Der aktuelle Explorer untersucht die Moeglichkeiten fuer genau eine Sequenz.
34   private Explorer currentExplorer = null;
35
36   // Flag zeigt an, dass die Generierung beendet werden soll
37   private boolean stopSession = false;
38
39   public Strategy4(IModel<Integer, Character> model, SequenceRatingSet<Character> ratingSet, int sneakDepth) {
40     super();
41     this.model = model;
42     this.ratingSet = ratingSet;
43     this.sneakDepth = sneakDepth;
44   }
45
46   @Override
47   public Character chooseElement(List<Character> sequence, Integer state, List<Character> elements) {
48     // auf den Explorer warten, dass er im BarrierContainer Elemente zur Verfuegung stellt
49     waitForExplorer();
50     Character retVal;
51     // Elemente vom BarrierContainer holen und eines auswaehlen
52     retVal = super.chooseElement(sequence, state, currentExplorer.barrierContainer.pop());
53     // Wenn ein Element ausgewaehlt wurde, ...
54     boolean continueExploring = retVal != null;
55     if (continueExploring) {
56       // ... wird der Explorer fuer den naechsten Schritt initialisiert
57       List<Character> sequenceCopy = new LinkedList<>(sequence);
58       sequenceCopy.add(retVal);
59       Integer newState = model.getNewState(state, retVal);
60       currentExplorer.initExploration(sequenceCopy, newState, model.getElementsForState(newState));
61     }
62     // auf Bereitschaft des Explores zum Weitersuchen warten
63     waitForExplorer();
64     // Wenn eine bisher leere Sequenz nicht fortgesetzt wird, soll die Generierung enden
65     if (sequence.isEmpty() && retVal == null) {
66       stopSession = true;
67     }
68     // ausgewaehltes Element oder null zurueckgeben
```

```
69      return retVal;
70    }
71
72    private void waitForExplorer() {
73      // Zur Synchronisierung mit dem Explorer wird der
          BarrierContainer in seiner Eigenschaft als
          CyclicBarrier verwendet
74      try {
75        currentExplorer.barrierContainer.await();
76      } catch (InterruptedException |
          BrokenBarrierException e) {
77        assert false;
78      }
79    }
80
81    @Override
82    public boolean stopSession(Collection<List<Character
        >> sequences) {
83      return stopSession;
84    }
85
86    public void dispose() {
87      // Nach der Generierung muessen der
          ExecutorService und ein eventuell noch
          laufender Explorer beendet werden.
88      if (currentExplorer != null) {
89        currentExplorer.dispose();
90      }
91      currentExplorer = null;
92      executorService.shutdown();
93      try {
94        executorService.awaitTermination(51, TimeUnit.
            SECONDS);
95      } catch (InterruptedException e) {
96        assert false;
97      }
98    }
99
100   @Override
101   public void sequenceStarted(List<Character> sequence
        ) {
102     // Fuer jede neue Sequenz wird in einem eigenen
          Thread ein neuer Explorer gestartet. Der
          vorherige wird beendet.
103     if (currentExplorer != null) {
104       currentExplorer.dispose();
105     }
106     Integer state = model.getInitialState();
107     currentExplorer = new Explorer(sequence, state,
          model.getElementsForState(state));
108     new Thread(currentExplorer).start();
109   }
110
111   @Override
112   public void sequenceAdded(List<Character> sequence)
        {
113   }
114
115   @Override
116   public void elementAdded(Character element, List<
        Character> sequence) {
117   }
118
119   // Innere Klasse Explorer zur Untersuchung der
        naechsten moeglichen Schritte fuer eine Sequenz
120   private class Explorer implements Runnable {
121
122     // Ablage fuer die Elemente die der Strategie zur
          Auswahl zur Verfuegung gestellt werden und
          CyclicBarrier zur Synchronisierung
123     private final BarrierContainer<List<Character>>
          barrierContainer = new BarrierContainer<>(2);
124
125     // eine Kopie der Sequenz im zu untersuchenden
```

```
Stand
126     private List<Character> currentSeed;
127
128     // der Modell-Zustand der dem Stand der
            untersuchten Sequenz entspricht
129     private Integer state;
130
131     // die fuer den naechsten Schritt verfuegbaren
            Elemente
132     private List<Character> nextElements;
133
134     // Ablage fuer die Bewertungen der naechsten
            Moeglichkeiten
135     private RatingResults ratingResult;
136
137     // Flag, das anzeigt, das der Explorer beendet
            werden kann
138     private boolean shutDown = false;
139
140     // Anzahl der laufenden SearchJobs
141     private final AtomicInteger numberOfRunningJobs =
            new AtomicInteger(0);
142
143     private Explorer(List<Character> sequence, Integer
             state, List<Character> elements) {
144       initExploration(sequence, state, elements);
145     }
146
147     private void initExploration(List<Character>
            sequence, Integer state, List<Character>
            elements) {
148       // Zur Untersuchung des naechsten Schritts muss
             der Generator mit den benoetigten
             Informationen initialisiert werden. Des
             Weiteren wird jedes Mal ein neues
             RatingResults-Objekt verwendet
149       ratingResult = new RatingResults(sequence.size()
            , ratingSet.rate(sequence));
150       this.currentSeed = sequence;
151       this.state = state;
152       this.nextElements = elements;
153     }
154
155     private void dispose() {
156       // dafuer sorgen, dass ein nicht mehr
             benoetigter Explorer beendet wird und den
             Generator-Thread freigibt
157       shutDown = true;
158       waitForStrategy();
159     }
160
161     @Override
162     public void run() {
163       // In dieser Schleife laeuft der Explorer in
             seinem eigenen Thread bis er via shutdown
             beendet wird.
164       while (true) {
165         // Jobs zur Untersuchung der naechsten
               Moeglichkeiten starten und warten, bis
               alle fertig sind
166         if (!nextElements.isEmpty()) {
167           startJobsAndWait();
168         }
169         // das RatingResult-Objekt die besten Elemente
               bestimmen lassen ...
170         List<Character> choice = new ArrayList<>(
               ratingResult.getBestElements());
171         // ... und diese der Strategie im
               BarrierContainer zur Verfuegung stellen
172         barrierContainer.push(choice);
173         // warten, dass die Strategie Interesse am
               Inhalt des BarrierContainers signalisiert
174         waitForStrategy();
175         // wenn das Signal zum Beenden kam, Schleifen
```

```
            verlassen
176         if (shutDown) {
177           break;
178         }
179         // Bereitschaft zum Weitersuchen signalisieren, warten dass die Strategie den BarrierContainer leert
180         waitForStrategy();
181       }
182     }
183
184     // synchronized, damit das wait() auf jeden Fall vor dem notify() des letzten SearchJobs kommt
185     private synchronized void startJobsAndWait() {
186       // Jobs zur Untersuchung der naechsten Moeglichkeiten starten ...
187       startSearchJobs(currentSeed, nextElements, state, sneakDepth - 1);
188       try {
189         // ... und auf den letzten Job warten
190         wait();
191       } catch (InterruptedException e) {
192         assert false;
193       }
194     }
195
196     private void startSearchJobs(List<Character> seed, Collection<Character> nextElements, Integer state, int depth) {
197       // Vor dem Starten der Jobs muss bereits die Zahl der laufenden Jobs erhoeht werden. Es kann sonst passieren, dass sie sieder 0 ist, bevor alle Jobs gestartet wurden.
198       numberOfRunningJobs.addAndGet(nextElements.size());
199       // fuer jedes moegliche Folgeelement einen eigenen Job starten und dem ExecutorService zur Bearbeitung uebergeben
200       for (Character element : nextElements) {
201         Runnable job = new SearchJob(Collections.unmodifiableList(seed), element, state, depth);
202         executorService.execute(job);
203       }
204     }
205
206     private void waitForStrategy() {
207       // Zur Synchronisierung mit der Strategie wird der BarrierContainer als CyclicBarrier verwendet
208       try {
209         barrierContainer.await();
210       } catch (InterruptedException | BrokenBarrierException e) {
211         assert false;
212       }
213     }
214
215     // Innere Klasse SearchJob zur Untersuchen eines moeglichen naechsten Schrittes fuer eine Sequenz
216     private class SearchJob implements Runnable {
217
218       // eine Kopie der Sequenz im zu untersuchenden Stand
219       private final List<Character> seed;
220
221       // der Modell-Zustand der dem Stand der untersuchten Sequenz entspricht
222       private final Integer state;
223
224       // das Element fuer den naechsten Schritt, den es zu untersuchen gilt
225       private final Character nextElement;
```

```java
226
227         // Anzahl der Schritte, die noch untersucht werden muessen, inklusive des aktuellen
228         private final int numberOfStepsToExplore;
229
230         public SearchJob(List<Character> seed, Character nextElement, Integer state, int depth) {
231           super();
232           this.seed = seed;
233           this.nextElement = nextElement;
234           this.state = state;
235           this.numberOfStepsToExplore = depth;
236         }
237
238         // Diese Methode ruft der ExecutorService zur Erledigung des Jobs auf.
239         @Override
240         public void run() {
241           // Kopie der Sequenz erstellen und ...
242           List<Character> seedCopy = new LinkedList<>(seed);
243           // ... das Element fuer den zu untersuchenden Schritt anhaengen
244           seedCopy.add(nextElement);
245           // diese Sequenz bewerten und ...
246           double score = ratingSet.rate(seedCopy);
247           // ... die Bewertung speichern
248           ratingResult.submit(score, numberOfStepsToExplore, seedCopy);
249           // Wenn noch weitere Schritte untersucht werden muessen, werden auf Basis der gerade bewerteten Seqeunz die naechsten Jobs gestartet.
250           if (numberOfStepsToExplore > 0) {
251             Integer newState = model.getNewState(state, nextElement);
252             List<Character> nextElements = model.getElementsForState(newState);
253             startSearchJobs(seedCopy, nextElements, newState, numberOfStepsToExplore - 1);
254           }
255           // Der letzte Job gibt den Explorer wieder frei.
256           if (numberOfRunningJobs.decrementAndGet() == 0) {
257             synchronized (Explorer.this) {
258               Explorer.this.notify();
259             }
260           }
261         }
262
263       }
264
265     }
266
267   }
```

BarrierContainer

```java
5   // Ein BarrierContainer dient als Ablage fuer ein Element vom Typ E und als CyclicBarrier
6   public class BarrierContainer<E> extends CyclicBarrier {
7
8     // Ablage fuer ein E
9     private E storedValue = null;
10
11    // Flag zeigt an, ob der Container gerade leer ist
12    private boolean isEmpty = true;
13
14    public BarrierContainer(int parties) {
15      this(parties, null);
16    }
```

```java

    public BarrierContainer(int parties, Runnable barrierAction) {
      super(parties, barrierAction);
    }

    public synchronized boolean isEmpty() {
      return isEmpty;
    }

    public synchronized boolean push(E value) {
      // Falls der Container nicht leer ist, kann kein Element abgelegt werden.
      if (!isEmpty) {
        return false;
      }
      storedValue = value;
      isEmpty = false;
      return true;
    }

    public synchronized E peek() {
      // Falls der Container leer ist, darf diese Methode nicht aufgerufen werden. Passiert das doch, deutet das auf Probleme mit der Synchronsierung hin.
      if (isEmpty) {
        throw new IllegalStateException("Container is empty.");
      }
      // abgelegtes Element ausgeben
      return storedValue;
    }

    public synchronized E pop() {
      // Falls der Container leer ist, darf diese Methode nicht aufgerufen werden. Passiert das doch, deutet das auf Probleme mit der Synchronsierung hin.
      if (isEmpty) {
        throw new IllegalStateException("Container is empty.");
      }
      // abgelegtes Element ausgeben und Container leeren
      isEmpty = true;
      return storedValue;
    }

  }
```

Literaturverzeichnis

[1] "IEEE Standard Glossary of Software Engineering Terminology," *IEEE Std 610.12-1990*, pp. 1–84, Dec 1990.

[2] B. Siebenhaar, "Machbarkeitsstudie zur Kombination eines modellbasierten Testwerkzeuges mit einem Testfallgenerierungstool," Master's thesis, Technische Universität Berlin, Aug. 2013.

[3] PikeTec GmbH, "TPT - Test eingebetteter Systeme," http://www.piketec.com/products/tpt.php?lang=de, accessed: 2015-09-19.

[4] Nils Olsson, Kristian Karl, "GraphWalker," http://graphwalker.org/, accessed: 2015-09-21.

[5] A. C. Dias Neto, R. Subramanyan, M. Vieira, and G. H. Travassos, "A Survey on Model-based Testing Approaches: A Systematic Review," in *Proceedings of the 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies: Held in Conjunction with the 22Nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007*, ser. WEASELTech '07. New York, NY, USA: ACM, 2007, pp. 31–36.

[6] T. Kanstrén and M. Chechik, "A comparison of three black-box optimization approaches for model-based testing," in *Proceedings of the 2014 Federated Conference on Computer Science and Information Systems, Warsaw, Poland, September 7-10, 2014.*, M. Ganzha, L. A. Maciaszek, and M. Paprzycki, Eds., 2014, pp. 1591–1598.

[7] J. Hopcroft, R. Motwani, and J. Ullman, *Einführung in die Automatentheorie, formale Sprachen und Komplexitätstheorie*, ser. I theoretische informatik. Pearson Education Deutschland GmbH, 2003.

[8] D. Schulz, *Eine Einführung in zeit-diskrete homogene Markov-Ketten.* GRIN Verlag, 2011.

[9] Oracle Corporation, "CyclicBarrier (Java Platform SE 8)," http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CyclicBarrier.html, accessed: 2015-09-16.

[10] ——, "ExecutorService (Java Platform SE 8)," http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html, accessed: 2015-09-17.

[11] E. Lehmann/Bringmann, "Time Partition Testing : systematischer Test des kontinuierlichen Verhaltens von eingebetteten Systemen," Ph.D. dissertation, 2004.