Carl von Ossietzky
Universität Oldenburg

Bachelorstudiengang Informatik

BACHELORARBEIT

Regionentreue Algorithmen

vorgelegt von: **Jan Steffen Becker**

Betreuender Gutachter: **Prof. Dr.-Ing. Oliver Theel**

Zweiter Gutachter: **Dipl.-Inform. Eike Möhlmann**

Oldenburg, 12. September 2013

Inhaltsverzeichnis

1 Einleitung **3**

2 Modell, Definitionen und Notationen **4**
2.1 Variablen, Zustände und Konfigurationen 4
2.2 Algorithmen und *Guarded Commands* 5
2.3 Fehlermodell 6
2.4 Ausführungen 6

3 Definition von Regionentreue **7**
3.1 Alternative Definition 8

4 Parallelen zu anderen Fehlertoleranzkonzepten **9**
4.1 Parallelen zur *Graceful Degradation* 10
4.2 Selbststabilisierung als orthogonales Konzept 13

5 Beispiel 1: Arithmetisches Mittel **15**
5.1 Modell und Algorithmus 16
5.2 Fehlerklasse 16
5.3 Beweis der Regionentreue 16

6 Beispiel 2: Bubblesort **17**
6.1 Algorithmus 18
6.2 Fehlermodell 19
6.3 Gütefunktion 20
6.4 Beweis der Regionentreue 21

7 Evaluation des Bubblesort-Algorithmus **22**
7.1 Implementierung des Beispielalgorithmus 22
7.2 Implementierung in PROMELA 23
7.3 Erstellen von Testfällen 25
7.4 Auswertung der Testreihen 26

8 Verbesserung der Regionentreue mittels replizierter Variablen **31**
8.1 Beweis der Notwendigkeit des Reparierens beim Lesen 32
8.2 Beweis der Verbesserung auf $(n+1)f$ 32
8.3 Anwendung und Simulation 34

9 Average Case Betrachtungen **39**
9.1 Mean Time To Failure (MTTF) 39
9.2 Betrachtung mittels Markov-Ketten 40

10 Zusammenfassung und Ausblick **50**

Anhang **54**

A Sourcecode **54**

B CD-ROM **59**

Abbildungsverzeichnis

1 Möglicher Güteverlauf eines regionentreuen Algorithmus 8
2 Vergleich von Fehlertoleranz nach [2] und Regionentreue 11
3 Aufbau des Netzes 18
4 Algorithmus für Knoten i 19
5 Heatmap der Güte vs. Fehlerzahl (Bubblesort) 29
6 Arithmetisches Mittel der Güte vs. Fehlerzahl (Bubblesort) 30
7 Boxplot der Güte vs. Fehlerzahl (Bubblesort) 30
8 Heatmap der Güte vs. Fehlerzahl (Bubblesort mit TMR) 37
9 Arithmetisches Mittel der Güte vs. Fehlerzahl (Bubblesort mit TMR) 38
10 Boxplot der Güte vs. Fehlerzahl (Bubblesort mit TMR) 38

Tabellenverzeichnis

1 Fehlertoleranzklassen 9
2 Absolute Häufigkeiten der Güte geordnet nach Fehlerzahl 29
3 Güte der Zustände 47
4 Mögliche Folgezustände bei Berechnungsschritten 48

Listings

1 PROMELA Code für Knoten P_i, $1 \leq i \leq n-2$ 23
2 Funktion zum Erzeugen gleichverteilter Pseudozufallszahlen 25
3 Java-Code zur Berechnung der Güte aus der initialen Belegung der Register `init` und der abschließenden Belegung `fin`. 26
4 Makrodefinitionen 35
5 Bubblesort mit replizierten Variablen 35

1 Einleitung

Fehlertoleranz kommt dort zum Einsatz, wo verlässliche Computersysteme gefordert werden. Dies ist vor allem dort wichtig, wo Computersysteme lebenswichtige Aufgaben übernehmen, beispielsweise in der Medizin oder der Raumfahrt. Aber auch dort wird Verlässlichkeit benötigt, wo Systemausfälle finanzielle Folgen haben, wie im Bankwesen. Diese Liste lässt sich beliebig fortführen.

Als *Fehler* werden die Ereignisse bezeichnet, die zum Versagen eines Systems führen können [15]. Dies kann z.B. der Ausfall einer Komponente des System sein, oder das „Kippen" eines Speicherbits. Im Allgemeinen nennt man ein System *fehlertolerant*, wenn es seine Funktion auch dann noch erfüllt, wenn Fehler auftreten [3, 15].

Um Fehlertoleranz zu erreichen, finden sich in der Literatur einige grundlegende Strategien.

Als Erstes sei die *Maskierung* von Fehlern genannt. Diese Methode wird vor allem bei fehlertoleranter Hardware eingesetzt, wie z.B. bei der Triple Modular Redundancy (TMR) [3, 15]. Bei TMR sind wichtige Hardwarekomponenten drei mal vorhanden. Alle drei Instanzen sollen parallel Ausgaben zu den selben Eingaben berechnen. Unter der Annahme, dass nur eine Komponente Fehler macht, muss das Ergebnis, das von mindestens zwei Komponenten berechnet wird, korrekt sein. Dieses Ergebnis wird durch einen hinter die Komponenten geschalteten *Voter* ausgewählt und weitergegeben. Ein weiteres Beispiel sind fehlermaskierende Codes, wie sie in der Datenübertragung und -speicherung sowie bei den populären QR-Codes eingesetzt werden [15, 21]. Hier werden Daten so kodiert, dass eine begrenzte Zahl beschädigter Datenbits erkannt und aus den restlichen Bits korrigiert werden kann, so dass trotzdem noch alle Daten wieder hergestellt werden können. Beide Verfahren zielen darauf ab, dass Fehler nach außen nicht sichtbar sind.

Als zweites großes Gebiet sei die *nicht-maskierende* Fehlertoleranz genannt [3]. Im Gegensatz zur maskierenden Fehlertoleranz sind hier Fehler nach außen hin durchaus sichtbar. Bei den meist angewandten Strategien werden diese aber wieder vollständig korrigiert. Bei Datenbanken werden Fehler (z.B. Datenverluste) erkannt und das System in einen früheren fehlerfreien Zustand zurückgesetzt. (Dies wird *Rollback* genannt [15].) Eine andere Klasse nicht-maskierender Systeme sind die *selbststabilisierenden* Systeme [8, 6]. Diese Systeme sind so entworfen, dass sie von jedem beliebigen Zustand irgendwann wieder einen stabilen Zustand erreichen, der in sich konsistent, also frei von Fehlern, ist. Hier sind z.B. mehrere Algorithmen zur Leader Election (z.B. [9, 8]) in verteilten Systemen bekannt.

Bei beiden Arten der Fehlertoleranz werden Fehler nach endlicher Zeit wieder vollständig eleminiert. Bei Fehlermaskierung wird der fehlerfreie Zustand nach außen hin gar nicht erst verlassen, bei nicht-maskierender Fehlertoleranz, wie sie hier definiert ist, ist ein Fehler nur über eine begrenzte Zeitspanne hinweg sichtbar [2]. Es

werden aber keine Angaben darüber gemacht, welchen Nutzen das System erbringt, während es sich in einem fehlerbehafteten Zustand befindet. In dieser Zeit liefert es möglicherweise keinerlei Funktion mehr [10].

T. Warns deutet an,

> *„[...] dass eine andere Variante der nicht-maskierenden Fehlertoleranz definiert werden kann, die es erlaubt, Sicherheit in Raum zu verletzen (anstatt in Zeit).“* [23, S. 31]

In dieser Arbeit wird deshalb eine neue Art der nicht-maskierenden Fehlertoleranz vorgestellt, die wir *Regionentreue* nennen. Wir sprechen hier von einer neuen Art, da es nicht Ziel der Regionentreue ist, einen fehlerfreien Zustand wieder herzustellen, sondern in dem fehlerbehafteten Zustand ein Höchstmaß der Funktionalität des Systems aufrecht zu erhalten.

Diese Arbeit ist wie folgt aufgebaut. Zunächst wird in Abschnitt 2 das verwendete Berechnungsmodell vorgestellt. Darauf aufbauend wird in Abschnitt 3 Regionentreue definiert und in Abschnitt 4 der Definition von Fehlertoleranz von Arora und Gouda [2] gegenübergestellt. Dabei wird Regionentreue mit *Graceful Degradation* [12] und *Selbststabilisierung* [8] verglichen. In den Abschnitten 5 und 6 werden dann zwei regionentreue Algorithmen vorgestellt und bewiesen. Der zweite davon wird in Abschnitt 7 simuliert und die Simulationsergebnisse im Hinblick auf die erreichte Regionentreue ausgewertet. In Abschnitt 8 wird ein Konzept vorgestellt, wie mithilfe replizierter Variablen die Regionentreue eines Algorithmus verbessert werden kann. Dieses Konzept wird nachfolgend anhand des zweiten Beispiels (jenem aus Abschnitt 6) erprobt. Abschnitt 9 befasst sich mit *average case* Betrachtungen zur Regionentreue, insbesondere bei Systemen, die mittels Markov-Ketten [17] modelliert werden können. Abschließend werden die Ergebnisse in Abschnitt 10 zusammengefasst.

2 Modell, Definitionen und Notationen

Im Folgenden werden die in dieser Arbeit angenommenen Modelle sowie die verwendeten Notationen vorgestellt.

2.1 Variablen, Zustände und Konfigurationen

Ein (verteiltes) *System* ist eine Menge an *Prozessen* $P_i \in \mathcal{P}$. Der *Zustand* eines Prozesses ergibt sich aus dem Inhalt aller *Variablen*, die dieser Prozess lesen kann. Die Zustände aller Prozesse eines Systems bilden zusammen die *Konfiguration* des Systems. Wenn ein System beispielsweise aus n Prozessen besteht, die ihren lokalen Zustand in den Variablen $s_1, s_2, \ldots, s_n$ speichern und über globale Register $r_1, r_2, \ldots, r_m$ untereinander Werte austauschen, ist eine Konfiguration von der Form $c = (s_1, s_2, \ldots, s_n, r_1, r_2, \ldots, r_m)$ (vgl. [8]). Die Begriffe Variable und Register werden synonym verwendet. Als lokal bezeichnen wir die Variablen, die nur von einem

Prozess gelesen oder beschrieben werden. Variablen, die von mehr als einem Prozess verwendet werden, als global. Der Inhalt einer Variable r in einer Konfiguration c wird in dieser Arbeit durch $c.r$ beschrieben. Die Menge aller möglichen Konfigurationen eines Systems sei $\mathcal{C}$. $\mathcal{C}$ ist damit eine Teilmenge des kartesischen Produkts der Wertebereiche aller Variablen.

Es wird angenommen, dass sich das System zu Beginn in einer festgelegten Konfiguration befindet. Diese *initiale Konfiguration* eines Systems bezeichnen wir als c_0.

2.2 Algorithmen und *Guarded Commands*

Jeder Prozess führt einen *lokalen Algorithmus* aus. Ein Algorithmus kann in Form von Pseudocode – wir werden insbesondere *Guarded Commands* [7] benutzen – oder, wie im ersten Beispiel in Abschnitt 5, einfach durch eine mathematische Formel beschrieben werden.

Guarded Commands wurden von E. Dijkstra insbesondere zur Beschreibung nichtdeterministischer Algorithmen eingeführt. Obwohl die lokalen Algorithmen der Prozesse in den Beispielen in dieser Arbeit deterministisch gewählt wurden, bieten Guarded Commands sich an, da sich die Beispiele so sehr kompakt ausdrücken lassen.

Ein *Guarded Command* $\mathsf{B} \to st$ besteht aus einer Bedingung B, dem *Guard*, und einer Folge von beliebigen Anweisungen *st*, dem *Command*. Ein Guarded Command heißt *aktiviert*, wenn sein Guard momentan zu **true** ausgewertet werden kann, also unter der momentanen Belegung der Variablen *wahr* ist.

Ein Guarded Command selbst ist kein Statement. Dijkstra führt aber zwei Konstrukte ein, die Guarded Commands zu Statements zusammenfassen. Das erste ist das *alternative construct* **if** ... **fi**, das eine Liste von Guarded Commands umschließt, die durch $\square$ getrennt werden. Die Semantik hiervon ist simpel: Der Prozess wählt ein beliebigen aktivierten Guarded Command aus der Liste aus und führt dessen Command aus. Wenn momentan kein Guarded Commant aus der Liste aktiviert ist, wartet der Prozess, bis dies der Fall ist.

Das zweite Konstrukt ist das *repitative construct* **do** ... **od**. Die Semantik entspricht der des *alternative construct*, nur dass das Auswählen und Ausführen eines Guarded Commands wiederholt wird, bis die Anweisung **break** erreicht wird.

Das System terminiert, wenn alle Prozesse das Ende ihres Algorithmus erreicht haben, oder keiner mehr einen Schritt ausführen kann.

Wir nehmen, wenn nichts anderes gekennzeichnet wird, das Prinzip der *read/write atomicity* [8] an. Dabei wird lediglich ein einzelner Lese- oder Schreibzugriff auf eine Variable als atomar angenommen. Wenn ein Prozess P_1 zum Beispiel eine Programmzeile

$$a := b + c;$$

ausführt, wird er zunächst die Inhalte der Variablen b und c lesen, danach die Summe berechnen und sie in die Variable a speichern. Zwischen dem Lesen der Werte und dem Berechnen und Schreiben des Ergebnisses kann aber ein weiterer Prozess P_2 den Inhalt von b oder c schon wieder verändert haben. Mann kann sich das so vorstellen, dass die gelesenen Werte in lokalen Hilfsvariablen zwischengespeichert werden und dann auf diesen lokalen Variablen weiter gerechnet wird [8].

Wenn eine Anweisung oder eine Folge von Anweisungen entgegen der *read/write atomicity* atomar ausgeführt werden soll, fassen wir diese in spitzen Klammern $\langle \dots \rangle$ ein.

Ein atomarer Schritt eines Algorithmus (im Folgenden *Berechnungsschritt* genannt), den ein Prozess ausführt, äußert sich in einem Zustandsübergang aus der momentanen Konfiguration c des Systems in eine Konfiguration c'. Die Menge aller möglichen (fehlerfreien) Zustandsübergänge eines Systems sei $\mathcal{A}$. Wir schreiben $c \rightarrow_{\mathcal{A}} c'$, wenn das System durch einen Schritt aus $\mathcal{A}$ aus der Konfiguration c in die Konfiguration c' übergehen kann.

2.3 Fehlermodell

Regionentreue ist ein Konzept, mit dem Fehlertoleranz erreicht werden soll, weshalb auch die möglichen Fehler selbst ein wichtiger Bestandteil des Modells sind. Wenn wir Fehlertoleranz nachweisen wollen, reicht es nicht aus, allein die Konfigurationen und Schritte des Algorithmus zu betrachten. Man muss auch festlegen, wie die Fehler aussehen, die das System tolerieren soll. Wir betrachten Regionentreue deshalb immer bezüglich einer zuvor definierten Fehlerklasse (auch Fehlermodell genannt) $\mathcal{F}$.

Bei genauerer Betrachtung entpuppt sich ein Fehler als nichts anderes als ein ungewollter Zustandsübergang. $\mathcal{F}$ ist somit die Menge aller Zustandsübergänge, die aufgrund von Fehlern auftreten. Wir schreiben $c \rightarrow_{\mathcal{F}} c'$ für Übergänge von c nach c' (folgend Fehlerschritt genannt).

Da auch ein Algorithmus im Grunde nichts anderes als eine Menge an Zustandsübergängen ist, können wir die Fehlerklasse ebenfalls quasi als eigenen Prozess mit eigenem Algorithmus beschreiben. Wir können also auch zu dem Zweck Guarded Commands verwenden [2, 10].

2.4 Ausführungen

Eine nicht leere Folge

$$\gamma = c_0 c_1 \dots c_n \in \mathcal{C}^+$$

von Konfigurationen, die mit der initialen Konfiguration c_0 des Systems beginnt und bei der jede Konfiguration c_i, $i > 0$, durch einen Berechnungs- oder Fehlerschritt aus ihrer Vorgängerkonfiguration c_{i-1} entstehen kann, bezeichnen wir als *Ausführung*.

Es ist aber nicht gefordert, dass in der letzten Konfiguration einer Ausführung kein Prozess mehr einen Schritt machen oder kein Fehler mehr auftreten kann, das

System also terminiert. Es ist vielmehr jedes nicht leere Präfix einer Ausführung ebenfalls eine Ausführung.

Wir nennen eine Konfiguration *erreichbar*, wenn es eine endliche Ausführung gibt, die in dieser Konfiguration endet.

Die Fehlerklasse kann Schritte enthalten, die ebenfalls in den Berechnungsschritten des Algorithmus enthalten sind. Nehmen wir z.B. an, dass ein Fehler eine Variable auf einen beliebigen Wert setzt, so ist jede Zuweisung im Algorithmus ebenso im Fehlermodell enthalten. Ein Berechnungsschritt darf allerdings für die Bestimmung von Regionentreue nicht als Fehlerschritt gezählt werden. Es sei deshalb $\#_{\mathcal{F}\setminus\mathcal{A}}(\gamma)$ die Anzahl Fehlerschritte in einer Ausführung γ, die keine Berechnungsschritte sind. Formal:

$$\chi_{\mathcal{F}\setminus\mathcal{A}} : \mathcal{C} \times \mathcal{C} \to \{0, 1\} \subset \mathbb{N}$$

mit

$$\chi_{\mathcal{F}\setminus\mathcal{A}}(c, c') := \begin{cases} 1 & \text{falls } c \to_{\mathcal{F}} c' \text{ und nicht } c \to_{\mathcal{A}} c' \\ 0 & \text{sonst} \end{cases} \tag{2.1}$$

die *charakteristische Funktion* einer Fehlerklasse $\mathcal{F}$ ohne Berechnungsschritte aus $\mathcal{A}$. Dann ist

$$\#_{\mathcal{F}\setminus\mathcal{A}}(c_0 c_1 \dots c_n) := \sum_{i=1}^{n} \chi_{\mathcal{F}\setminus\mathcal{A}}(c_{i-1}, c_i). \tag{2.2}$$

3 Definition von Regionentreue

Wie in der Einleitung bereits angedeutet, verstehen wir unter der Regionentreue ein Konzept, bei dem es darum geht, trotz Fehlern immer noch ein Höchstmaß an Funktionalität aufrecht zu erhalten.

Dafür nehmen wir zunächst einmal an, dass wir in einer beliebigen Konfiguration eines Systems bestimmen können, in welchem Maße das System seine Spezifikation erfüllt. Wir gehen sogar so weit, dass wir einer Konfiguration einen Wert zwischen 0 und 1 zuordnen können, der diesen Anteil der momentan erbrachten Funktionalität in Bezug zur Spezifikation misst. Diesen Wert nennen wir *Güte* einer Konfiguration. Die Funktion, mit der die Güte für einen Algorithmus berechnet wird, heißt *Gütefunktion*.

Solange kein Fehler auftritt, ist die Güte einer Konfiguration 1. Bei einer Güte von 0 ist keine Funktionalität des Systems mehr sichergestellt. Wir gehen davon aus, dass, solange die Güte größer 0 ist, das System zumindest lebendig ist.

Wir nennen ein System *regionentreu*, wenn nach k Fehlern die Güte immer noch größer oder gleich $1 - k\Delta$ ist. Dabei ist Δ ebenfalls ein Wert zwischen 0 und 1, der den Güteverlust begrenzt. Um mindestens f Fehler tolerieren zu können, fordern wir $\Delta < \frac{1}{f}$. Formal:

Definition 3.1 (Regionentreue)**.** *Gegeben sei ein System mit den Konfigurationen* $\mathcal{C}$ *und dem Algorithmus* $\mathcal{A}$*. Ferner seien* $g : \mathcal{C} \to [0, 1]$ *eine Gütefunktion und* $\mathcal{F}$ *eine Fehlerklasse des Systems. Es sei* $f \in \mathbb{N}$*.*

$\mathcal{A}$ *ist* f-regionentreu bezüglich g und $\mathcal{F}$*, g.d.w. für alle erreichbaren Konfigurationen* $c \in \mathcal{C}$ *und alle Ausführungen* $\gamma = c_0 \dots c$*, die in* c *enden,*

$$g(c) \geq 1 - \#_{\mathcal{F} \setminus \mathcal{A}}(\gamma)\Delta \tag{3.1}$$

für ein $\Delta \in \mathbb{R}$ *mit*

$$0 \leq \Delta < \frac{1}{f} \tag{3.2}$$

gilt.

Wir fordern damit also eine untere Grenze für die Qualität des Ergebnisses, das der Algorithmus erbringt. Bei jedem Fehler setzen wir diese Grenze um einen festen Betrag herab. Diese Differenz darf allerdings nicht so groß sein, dass eine Güte von 0 erreicht werden kann, bevor mehr als f Fehler aufgetreten sind. Damit wird zumindest ein Teil der Funktionalität über eine begrenzte Zahl an Fehlern hinweg aufrecht erhalten.

Wie am Beispiel 2 (Abschnitt 6) ersichtlich ist, kann sich ein Algorithmus aber im Durchschnitt wesentlich robuster gegen Fehler erweisen, als durch die Regionentreue gefordert.

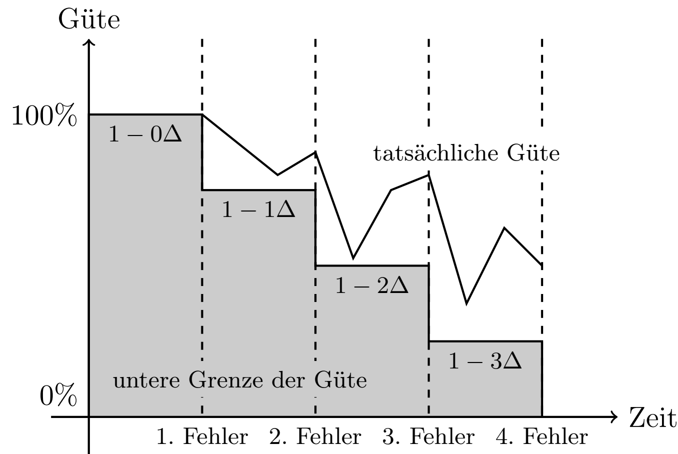


Abbildung 1: Möglicher Güteverlauf eines regionentreuen Algorithmus. Die untere Grenze für die Güte verschiebt sich bei jedem Fehler um Δ nach unten.

3.1 Alternative Definition

Bei der hier vorgeschlagenen Definition von Regionentreue wird die untere Grenze der Güte bei jedem Fehler um einen *absoluten* Betrag Δ – also ein fester Teil der maximalen Güte – herabgesetzt. Alternativ könnte man die Grenze bei jedem Fehler

um einen *relativen* Betrag – also einen festen Teil der Grenze nach dem letzten Fehler – heruntersetzen. Man würde Gleichung (3.1) aus Definition 3.1 durch

$$g(c) \geq \Lambda^{\#_{\mathcal{F}\setminus\mathcal{A}}(\gamma)} \tag{3.3}$$

ersetzen und z.B.

$$2^{-\frac{1}{f}} < \Lambda \leq 1 \tag{3.4}$$

fordern. Damit dürfte die Güte bei f Fehlern jedes Mal um weniger als die Hälfte absinken.

Bei dieser Definition wird ein regionentreues System zu jedem Zeitpunkt noch einen Rest an Funktion liefern, egal wie viele Fehler und in welchem Abstand diese aufgetreten sind. Solche Systeme sind aber wahrscheinlich sehr schwer und je nach Fehlermodell sogar unmöglich zu finden. Das in Abschnitt 6.2 und 8 angenommene Fehlermodell wäre nicht möglich. Dort werden wir annehmen, dass ein Fehler immer eine Variable bzw. eine Instanz einer replizierten Variable beliebig verändert. Bei ausreichend vielen hintereinander auftretenden Fehlern könnten alle Variablen verändert werden und der Algorithmus damit in einen beliebigen Zustand versetzt werden, in dem der Algorithmus u.U. nicht mehr arbeiten kann. Dies hätte die unerlaubte Güte von 0 zur Folge. Aus diesem Grund wird diese Variante der Regionentreue in dieser Arbeit nicht weiter vertieft.

4 Parallelen zu anderen Fehlertoleranzkonzepten

Die Anforderungen, die an ein Fehlertoleranzkonzept gestellt werden, werden meist in zwei Teile aufgeteilt. Zum einen soll ein System *sicher (safe)*, zum anderen *lebendig (live)* sein. Wenn man als Beispiel eine Verkehrsampel nimmt, ist eine zentrale Anforderung an die Sicherheit, dass keine Verkehrsteilnehmer in unterschiedlichen Richtungen die Kreuzung zugleich überqueren dürfen. Es könnte sonst zu Unfällen kommen, die durch die Anlage ja gerade vermieden werden sollen. Umgekehrt wird man aber auch fordern, dass kein Verkehrsteilnehmer unendlich lange an einer roten Ampel warten muss. Dies sichert die Lebendigkeit – der Verkehrsfluss wird aufrecht erhalten.

Im Allgemeinen werden Fehlertoleranzkonzepte in vier Klassen eingeteilt [10]. Diese werden dadurch unterschieden, welche der beiden Eigenschaften *Sicherheit* und *Lebendigkeit* zugesichert werden. Die vier Klassen sind in Tabelle 1 wiedergegeben.

	lebendig	nicht lebendig
sicher	maskierend fehlertolerant	fail-stop
nicht sicher	nicht-maskierend fehlertolerant	nicht fehlertolerant

Tabelle 1: Fehlertoleranzklassen

Die *maskierend fehlertoleranten* Systeme sind jene, bei denen ein Fehler gar nicht erst nach außen sichtbar wird. Wie in der Einleitung bereits erwähnt, gehören korrigierende Codes in diese Klasse. In der Klasse der *nicht-maskierend fehlertoleranten* Systeme sind die selbststabilisierenden Algorithmen ein populärer Vertreter. Im Gegensatz zu den maskierenden Systemen sind Fehler nach außen hin sichtbar. Die Lebendigkeit des Systems bleibt aber immer gewahrt. *Fail-stop*-Systeme sind hingegen so konstruiert, dass das System anhält, sobald ein Fehler auftritt. So wird die Sicherheit des Systems garantiert und dabei ein Verlust der Lebendigkeit in Kauf genommen. Die vierte Klasse bilden die *nicht-fehlertoleranten* Systeme. Im Fehlerfall kann weder Lebendigkeit noch Sicherheit garantiert werden.

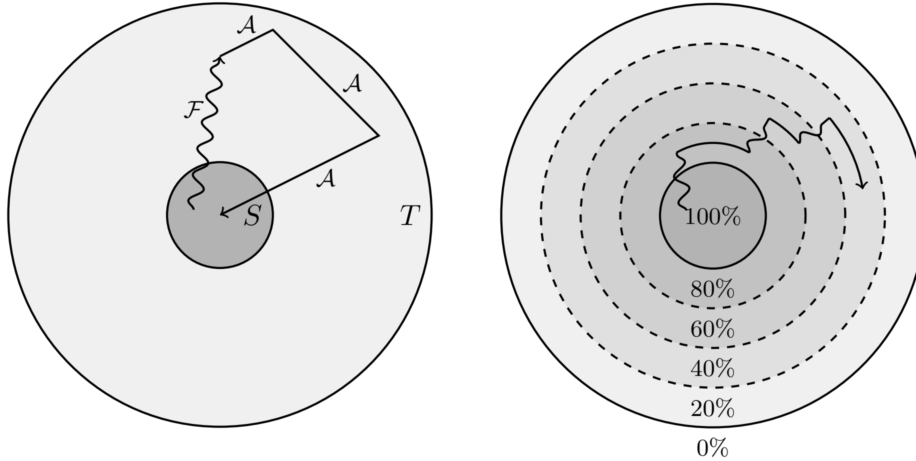
Eine gebräuchliche Definition von Fehlertoleranz findet sich in [2]. Die Spezifikation eines Systems wird in Form eines Prädikats S auf der Menge der Konfigurationen ausgedrückt. Die *stabilen* Konfigurationen, in denen die Spezifikation gilt, sind die, in denen S zu **true** ausgewertet wird. Weiterhin wird eine Fehlerklasse $\mathcal{F}$ angenommen. Ein System heißt dann $\mathcal{F}$-Fehlertolerant bezüglich S g.d.w. ein weiteres Prädikat T mit $S \Rightarrow T$ existiert, so dass folgende Bedingungen gelten [2]:

- T ist sowohl unter Berechnungs- als auch Fehlerschritten abgeschlossen. Wenn in einer Konfiguration T gilt, so gilt T auch in allen Folgekonfigurationen.
- S ist unter Berechnungsschritten abgeschlossen. Ist eine Konfiguration *stabil* (d.h. es gilt S), so ist es auch jede durch Berechnungsschritte erreichbare Folgekonfiguration.
- T konvergiert unter Berechnungsschritten zu S. Gilt in einer Konfiguration T, so wird nach endlich vielen Berechnungsschritten immer eine stabile Konfiguration erreicht. (Solange dazwischen keine Fehler auftreten.)

Wegen der Abgeschlossenheit werden S und T *Invarianten* genannt. T wird auch *fault-span* genannt. Es gibt i.A. mehrere *fault-spans* T. Wenn das schwächste zu findende Prädikat T_w alle Konfigurationen umfasst – also $T_w = \textbf{true}$ gilt – ist das System *selbststabilisierend.* Wenn für das stärkste Prädikat $T_s = S$ gilt, ist das System *maskierend fehlertolerant.*

4.1 Parallelen zur *Graceful Degradation*

Da unser Konzept der Regionentreue keine Konvergenz in die stabilen Konfigurationen (das wären jene mit einer Güte von 100%) fordert, sind regionentreue Systeme nach obiger Definition nicht unbedingt fehlertolerant. Abbildung 2 verdeutlicht den Unterschied. Ein Fehler kann ein solches System zunächst „weit“ von den stabilen Zuständen entfernen. Das System kehrt aber immer in einen stabilen Zustand zurück. Bei einem regionentreuen System werden die stabilen Zustände hingegen nur schrittweise verlassen. Man würde ein regionentreues System daher vielmehr als *fail-soft* [3] oder *gracefully degrading* [3, 12] bezeichnen. Dies sind Systeme, bei denen



(a) Ein fehlertolerantes System nach [2]

(b) Ein regionentreues System. Die Prozentangaben sind Beispiele für die Güte.

Abbildung 2: Die Kreise stellen den Zustandsraum dar. Berechnungsschritte sind durch $\longrightarrow$ und Fehlerschritte durch $\rightsquigarrow$ angedeutet. Während in einem fehlertoleranten System nach [2] (links) ein Fehler in einen beliebigen Zustand innerhalb von T führt, werden in einem regionentreuen System (rechts) die stabilen Zustände nur schrittweise verlassen.

ein Fehler sich durch einen Verlust an Servicequalität (beispielsweise Performance) bemerkbar macht, aber nicht zum totalen Versagen des Systems führt [3].

Herlihy und Wink modellieren in [12] ein System als Menge an Objekten. Ein Objekt hat eine Menge an Zuständen $STATES$, die es annehmen kann, und eine Menge Operationen OP. Ein Objekt wird als endlicher Automat $A = (STATES, s_0, OP, \delta) \in \mathcal{A}$ modelliert. Dabei ist s_0 der initiale Zustand des Objekts und $\delta : STATES \times OP \rightarrow \mathfrak{P}(STATES)$ eine Funktion, die angibt, welche Zustandswechsel des Objekts bei einer Operation möglich sind[1]. (Aus einem Zustand s kann das Objekt bei der Operation o in einen Zustand $s' \in \delta(s, o)$ wechseln.) Die von diesem Automaten akzeptierte Sprache $\mathcal{L}(A) \subseteq OP^*$ gibt also alle zulässigen Operationsfolgen des Objekts an.

Zusätzlich nehmen sie eine Menge an Bedingungen *(constraints)* $\mathcal{B}$ an die Umgebung des Systems an.[2]

Die Autoren stellen die sogenannte *Relaxation Method* vor. Dabei sei $\mathcal{A}$ eine Menge an Automaten mit den selben Zuständen und Operationen, aber unterschiedlichen Zustandsübergängen (sie repräsentieren also das gleiche Objekt), deren Sprachen einen Verband[4] unter *inverser Mengeninklusion* ($\supseteq$) bilden, d.h. für alle $A_1, A_2 \in \mathcal{A}$ existieren $A_3, A_4 \in \mathcal{A}$ mit

$$\mathcal{L}(A_3) = \mathcal{L}(A_1) \cup \mathcal{L}(A_2) \tag{4.1}$$

[1] $\mathfrak{P}$ bezeichnet die Potenzmenge.

[2] In der Originalliteratur wird der Buchstabe $\mathcal{C}$ verwendet. Um Verwechslungen mit den Konfigurationen $\mathcal{C}$ in unserem Berechnungsmodell zu vermeiden, wurde hier auf den Buchstaben $\mathcal{B}$ ausgewichen.

und

$$\mathcal{L}(A_4) = \mathcal{L}(A_1) \cap \mathcal{L}(A_2) \tag{4.2}$$

Dazu wird ein Verbandshomomorphismus $\phi : \mathfrak{P}(\mathcal{B}) \to \mathcal{A}$ definiert, der einer Teilmenge der Bedingungen einen Automaten zuordnet. Dieser muss für beliebige $B_1, B_2 \subseteq \mathcal{B}$

$$\mathcal{L}(\phi(B_1 \cup B_2)) = \mathcal{L}(\phi(B_1)) \cap \mathcal{L}(\phi(B_2)) \tag{4.3}$$

und

$$\mathcal{L}(\phi(B_1 \cap B_2)) = \mathcal{L}(\phi(B_1)) \cup \mathcal{L}(\phi(B_2)) \tag{4.4}$$

erfüllen [4].

Dieser Homomorphismus ordnet damit der größten Menge an Bedingungen die kleinste Sprache und damit das „strengste“ Verhalten von $\mathcal{A}$ zu. Wenn alle Bedingungen aus $\mathcal{B}$ erfüllt sind, wird damit nur ein Verhalten akzeptiert, das die Spezifikation des Systems erfüllt. Je weniger dieser Bedingungen erfüllt sind, desto größer ist die Menge zulässiger Operationsfolgen. Wenn also wichtige Anforderungen an die Umgebung des Systems nicht mehr erfüllt sind, wird ein Verhalten zugelassen, das nicht mehr vollständig der Spezifikation entspricht, aber immer noch ein Mindestmaß an Funktionalität aufrecht erhält.

Bei regionentreuen Systemen hingegen gehen wir statt von einer Menge an Anforderungen an die Umgebung, die nicht mehr erfüllt werden können, einfach von einer Zahl Fehler aus, die aufgetreten sind. Ein Fehler kann sich durchaus in dem Verlust einer Anforderung – beispeilsweise dem Ausfall einer Komponente – äußern. Nach einem Fehler wird die Menge der akzeptablen Zustände vergrößert. Dadurch vergrößert sich zugleich auch die Menge der zulässigen Zustandsfolgen. Die Anforderungen an das System werden damit bei beiden Konzepten in ähnlicher Weise aufgeweicht. Ein wesentlicher Unterschied ist aber, dass es bei der *Graceful Degradation* vollständig dem Entwickler überlassen bleibt, wie ein abgeschwächtes Verhalten des Systems aussehen darf. Die Regionentreue setzt hier durch die Güte engere Grenzen.

Ein Beispiel für ein *gracefully degrading system* findet sich in [20]. Die Autoren präsentieren ein Fahrstuhlkontrollsystem, das funktionstüchtig ist, solange ein Minimum an erforderlichen Komponenten korrekt arbeitet. Auch wenn verschiedene Bedienelemente und Sensoren des Fahrstuhls ausfallen, können noch alle Stockwerke erreicht werden. Es befinden sich z.B. an jedem Stockwerk Sensoren, die ein Signal senden, wenn der Fahrstuhl am entsprechenden Stockwerk ankommt. Wenn diese ausfallen, können sie durch die Messwerte der Positions- und Geschwindigkeitssensoren emuliert werden. Ebenso sind Stockwerkanzeigen für die Fahrgäste nicht für den Betrieb erforderlich. Ein Ausfall von Komponenten hat aber u.U. eine längere Zeit

zur Folge, die Passagiere von einem Stockwerk zum anderen benötigen. Wenn man die Ausfälle als Fehler betrachtet und die mittlere Fahrzeit als Güte nimmt, könnte man so ein regionentreues System erhalten.

4.2 Selbststabilisierung als orthogonales Konzept

Das Konzept der Regionentreue lässt sich gut mit anderen Konzepten zur Fehlertoleranz verbinden. In Abschnitt 6 werden wir einen Algorithmus vorstellen, der in Teilen selbststabilisierend ist. In Abschnitt 8 werden wir eine Methode vorstellen, mit der regionentreue Algorithmen um maskierende Eigenschaften erweitert werden können und damit die Regionentreue verbessert wird. Hier werden wir zeigen, mit welchen Eigenschaften ein regionentreuer Algorithmus zugleich selbststabilisierend ist.

Ein Algorithmus heißt selbststabilisierend, wenn aus jeder beliebigen Konfiguration nach endlich vielen Berechnungsschritten eine stabile Konfiguration erreicht wird [6, 8]. Bei einem regionentreuen Algorithmus sind dies die Konfigurationen, in denen die Gütefunktion ihr Maximum von 100% erreicht. Hier wird die Spezifikation des Algorithmus voll erfüllt.

Die Regionentreue kann damit gut als ein orthogonales Konzept zur Selbststabilisierung, bzw. der Definition nicht-maskierender Fehlertoleranz nach [2], bezeichnet werden. Selbststabilisierende Systeme begrenzen den Verlust der Funktionalität des Systems *zeitlich*, während regionentreue Systeme ihn *räumlich* – auf den Zustandsraum bezogen, damit meinen wir also den Güteverlust – begrenzen. In der Tat reicht eine Verschärfung der Anforderungen an ein regionentreues System aus, um Selbststabilisierung zu erzwingen.

Wir führen dazu zunächst eine verschärfte Form der Regionentreue ein, in der wir nicht mehr die Gesamtzahl der zu einem Zeitpunkt geschehenen Fehler betrachten, sondern nur jeweils Anforderungen an die Güte in einem einzigen Fehler- oder Berechnungsschritt stellen.

Lemma 4.1. *Ein System ist insbesondere dann regionentreu nach Definition 3.1, wenn es folgende Eigenschaften erfüllt:*

1. *Für die initiale Konfiguration c_0 gilt*

$$g(c_0) = 1 \tag{RT1}$$

2. *Für beliebige Konfigurationen $c, c' \in C$ gelten*

$$c \rightarrow_{\mathcal{A}} c' \Rightarrow g(c') \geq g(c) \tag{RT2}$$

und

$$c \rightarrow_{\mathcal{F}} c' \Rightarrow g(c') \geq g(c) - \Delta \tag{RT3}$$

für ein festes $\Delta < \frac{1}{f}$

Beweis von Lemma 4.1. Wir zeigen, dass aus Gleichungen (RT1) bis (RT3) Regionentreue folgt, also

$$g(c) \geq 1 - \#_{\mathcal{F}\setminus\mathcal{A}}(\gamma)\Delta$$

Gegeben sei ein System, das die Bedingungen (RT1) bis (RT3) aus Lemma 4.1 erfüllt.

Es sei $c \in \mathcal{C}$ eine beliebige Konfiguration des Systems und $\gamma = c_0 \ldots c_n$ eine Ausführung. Gleichungen (RT2) und (RT3) lassen sich mittels der Funktion $\chi_{\mathcal{F}\setminus\mathcal{A}}$ aus Gleichung (2.1) zu

$$g(c_{i-1}) - g(c_i) \leq \Delta\chi_{\mathcal{F}\setminus\mathcal{A}}(c_{i-1}, c_i) \tag{4.5}$$

für beliebige c_i, $1 \leq i \leq n$, in γ zusammenfassen. Es folgt dann

$$g(c_n) = g(c_0) - g(c_0) + g(c_1) - g(c_1) + \cdots + g(c_{n-1}) - g(c_{n-1}) + g(c_n) \tag{4.6}$$

$$= g(c_0) - \sum_{i=1}^{n}(g(c_{i-1}) - g(c_i)) \tag{4.7}$$

mit Gleichungen (RT1) und (4.5)

$$\geq 1 - \Delta\sum_{i=1}^{n}\chi_{\mathcal{F}\setminus\mathcal{A}}(c_{i-1}, c_i) \tag{4.8}$$

$$= 1 - \Delta\#_{\mathcal{F}\setminus\mathcal{A}}(\gamma) \tag{4.9}$$

□

In einem solchen System sinkt die Güte nur in einem Fehlerschritt ab. Dabei ist der Güteverlust durch Gleichung (RT3) begrenzt. Es werden aber noch keinerlei reparierende Eigenschaften gefordert, auch wenn diese nicht ausgeschlossen sind. Wenn wir allerdings Gleichung (RT2) durch

$$c \rightarrow_{\mathcal{A}} c' \Rightarrow (g(c') = 1 \vee g(c') \geq g(c) + \Delta_2 \tag{4.10}$$

mit $\Delta_2 > 0$ ersetzen, ist ein System mit diesen Eigenschaften *selbststabilisierend.* Da Gleichung (4.10) Gleichung (RT2) impliziert, bleibt die Regionentreue erhalten.

Beweis. Es ist g eine Variantfunktion [16, 8]. g hat per Definition ein Maximum von 1 in den stabilen Zuständen. Nach Gleichung (4.10) (mit $\Delta_2 > 0$) ist g bei

Berechnungsschritten streng monoton steigend. Die Abgeschlossenheit der stabilen Zustände unter Berechnungsschritten folgt ebenfalls aus Gleichung (4.10). □

Nach k Fehlern stabilisiert sich ein solches System in maximal

$$\ell = \left\lceil \frac{k\Delta}{\Delta_2} \right\rceil \tag{4.11}$$

Schritten.

Beweis. Sei $\gamma = c_m c_{m+1} \dots c_{m+\ell}$ ein Suffix einer Ausführung. In c_m (am Anfang von γ) seien bereits k Fehler aufgetreten und γ bestehe nur aus

$$\ell \geq \frac{k\Delta}{\Delta_2} \tag{4.12}$$

Berechnungsschritten. Wir nehmen an, dass am Ende von γ $g(c_{m+\ell}) < 1$ gilt. $c_{m+\ell}$ ist also nicht stabil.

Aus der Regionentreue folgt nach Definition

$$g(c_m) > 1 - k\Delta \tag{4.13}$$

und mit Gleichungen (4.10), (4.12) und der Annahme

$$1 > g(c_{m+\ell}) \geq 1 - k\Delta + \frac{k\Delta}{\Delta_2}\Delta_2 = 1 \tag{4.14}$$

Dies ist ein Widerspruch. □

5 Beispiel 1: Arithmetisches Mittel

Als erstes Beispiel für einen einfachen regionentreuen Algorithmus werden wir im Folgenden das *arithmetische Mittel* von Messwerten vorstellen.

Anwendungsfall könnte hier z.B. ein Sensornetzwerk sein, das Temperaturwerte misst. Wir gehen hier davon aus, dass dabei alle Sensoren den *gleichen* Wert messen sollen. Die Messwerte sind somit *redundant*. Wenn ein Sensor immer korrekt arbeiten würde, bräuchte man nur einen einzigen Sensor zum Ermitteln der Daten. Wir nehmen aber an, dass

1. die Sensoren alle unabhängig arbeiten,
2. die Daten einen festen *Wertebereich* haben,
3. auch bei fehlerfreiem Betrieb die Daten einem *Messfehler* unterworfen sind und
4. die Sensorknoten *unabhängig voneinander ausfallen* können. In dem Fall liefert ein Sensor *willkürliche* Werte, die aber immer noch *innerhalb des Wertebereichs* liegen.

In diesem Fall bringt das *arithmetische Mittel* statistisch einen besseren Messwert. Im Folgenden werden wir dieses System genauer betrachten und zeigen, dass die Abweichung des so ermittelten Gesamtmesswerts auch bei ausgefallenen Sensoren nur begrenzt vom Sollwert abweicht.

5.1 Modell und Algorithmus

Gegeben sei ein einfaches Sensornetzwerk bestehend aus n Sensorknoten, die Messwerte v_i, $i = 0, 1, \ldots, n-1$, für ein (quasi-)stetiges Datum v liefern. Die möglichen Daten und Messwerte sind auf einen festen Bereich $v, v_i \in [\ell, u]$ beschränkt. Aus den Messwerten der einzelnen Knoten wird dann der Wert von v durch

$$\tilde{v} := \frac{1}{n} \sum_{i=0}^{n-1} v_i \tag{5.1}$$

geschätzt.

5.2 Fehlerklasse

Wir nehmen eine einfache Fehlerklasse an, in der ein einzelner Sensor entweder korrekt arbeitet – dann weicht v_i höchstens um ε, $\varepsilon \geq 0$, von v ab – oder ausgefallen ist. In diesem Fall liefert er willkürliche Werte. Wir führen dafür eine konzeptionelle Error-Variable $failed_i$ für jeden Sensorknoten ein, die **true** ist, g.d.w. Sensor i ausgefallen ist. Für den gelieferten Wert v_i ergibt sich damit

$$v_i \in \begin{cases} [\ell, u] & \text{falls } failed_i \\ [\max\{v - \varepsilon, \ell\}, \min\{v + \varepsilon, u\}] & \text{sonst} \end{cases}. \tag{5.2}$$

Die Fehlerklasse kann als folgende Menge *Guarded Commands* beschrieben werden:

$$\{(\neg failed_i \rightarrow failed_i := \mathbf{true}) | i = 0, 1, \ldots, n-1\} \tag{5.3}$$

5.3 Beweis der Regionentreue

Im Folgenden werden wir zeigen, dass $\tilde{v}$ regionentreu mit $\Delta = n^{-1}$ bezüglich der gegebenen Fehlerklasse und der Gütefunktion

$$g(\tilde{v}) := \min\left\{1 - \frac{|\tilde{v} - v| - \varepsilon}{u - \ell}, 1\right\} \tag{5.4}$$

ist, in dem Sinne, dass $g(\tilde{v}) \geq 1 - k\Delta$ ist. Dabei ist k die Anzahl der aufgetretenen Fehler – hier die Anzahl der ausgefallenen Sensorknoten.

Zunächst zeigen wir, dass

$$|\tilde{v} - v| \leq \frac{k}{n}(u - \ell) + \frac{n-k}{n}\varepsilon \tag{5.5}$$

gilt.

Beweis. Seien o.B.d.A. die ersten k Sensorknoten ausgefallen, also $\forall i = 0, 1, \ldots, n-1 : failed_i \Leftrightarrow i < k$. Dann ist

$$\tilde{v} - v = \frac{1}{n} \sum_{i=0}^{n-1} v_i - v \tag{5.6}$$

$$= \frac{1}{n} \left(\sum_{i=0}^{k-1} \underbrace{(v_i - v)}_{\substack{\leq u-\ell \\ \geq \ell-u}} + \sum_{i=k}^{n-1} \underbrace{(v_i - v)}_{\substack{\leq \varepsilon \\ \geq -\varepsilon}} \right) \tag{5.7}$$

Es folgt direkt die obige Behauptung. □

Aus Gleichung (5.5) folgt die Regionentreue.

Beweis.

$$|\tilde{v} - v| \leq \frac{k}{n}(u - \ell) + \frac{n-k}{n}\varepsilon \tag{5.8}$$

$$|\tilde{v} - v| - \varepsilon \leq \frac{k}{n}(u - \ell) \tag{5.9}$$

$$\frac{|\tilde{v} - v| - \varepsilon}{u - \ell} \leq \frac{k}{n} \tag{5.10}$$

$$1 - \frac{|\tilde{v} - v| - \varepsilon}{u - \ell} \geq 1 - \frac{k}{n} \tag{5.11}$$

Wegen

$$0 \leq 1 - \frac{k}{n} = 1 - k\Delta \leq 1, \quad 0 \leq k \leq n \tag{5.12}$$

gilt dann auch

$$g(\tilde{v}) \geq 1 - k\Delta. \tag{5.13}$$

Damit ist der Algorithmus regionentreu. □

6 Beispiel 2: Bubblesort

Bubblesort ist ein einfacher vergleichsbasierter Sortieralgorithmus mit quadratischer Laufzeit. Er sortiert eine gegebene Liste $(v_0, v_1, \ldots, v_n)$ an Werten nach der einfachen Regel *„Wenn für ein Paar benachbarter Werte $v_i > v_{i+1}$ gilt, tausche v_i und v_{i+1}.“*

Beweis. Wir zeigen: Die Anwendung dieser Regel (in beliebiger Reihenfolge der v_i) sortiert eine Liste $(v_0, v_1, \ldots, v_n)$ in endlich vielen Schritten.

Sei $v = (v_0, v_1, \ldots, v_n)$ und $f(v) := |\{(i,j) | i < j \wedge v_i > v_j\}|$. Falls $f(v) = 0$, ist v sortiert und wir sind fertig.

Sonst finde ein benachbartes Paar mit $v_k > v_{k+1}$ und erhalte die Liste v' aus v durch Tauschen von v_k und v_{k+1}. Dann gilt $v'_k < v'_{k+1}$. Ansonsten werden aber

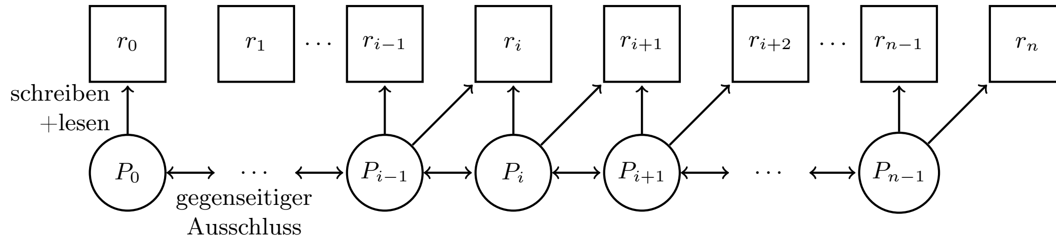


Abbildung 3: Aufbau des Netzes

keine Paare (v_i, v_j) verändert, so dass $f(v') = |\{(i,j)|i < j \wedge v_i > v_j\} \setminus \{(k, k+1)\}| = f(v) - 1$ gilt. Wegen $f(v) \leq \frac{n(n-1)}{2}$ ist die Folge in maximal $\frac{n(n-1)}{2}$ Schritten sortiert. □

6.1 Algorithmus

Wir verwenden ein verteiltes System mit n Knoten, die von 0 bis $n-1$ nummeriert seien. O.B.d.A. sollen $n+1$ Werte aus den Natürlichen Zahlen sortiert werden. Jeder Knoten mit Nummer i hält einen Wert der zu sortierenden Liste in einem öffentlichen Register r_i. Die r_i können beliebige Werte annehmen. Es werden $n+1$ Werte sortiert, dem Register r_n ist also kein Knoten zugeordnet. Jeder Knoten speichert seinen aktuellen Zustand in einem globalen Register s_i, das die Werte 0, 1, 2 und 3 annehmen kann.

Jeder Knoten kann die Zustände seiner Nachbarn s_{i-1} und s_{i+1} lesen und das Register r_{i+1} seines Nachfolgers zusätzlich zu seinem eigenen sowohl lesen als auch beschreiben. Es wird $s_{-1} = s_n =$ konstant 0 angenommen.

Zu Beginn befinden sich alle Knoten im Zustand $s_i = 0$ und die zu sortierenden Werte sind auf die Register r_0 bis r_n verteilt.

Jeder Knoten i führt den folgenden lokalen Algorithmus aus, der in Abbildung 4 als Zustandsautomat veranschaulicht ist:

- $s_i = 0$: Wenn $r_i > r_{i+1}$ ist und $s_{i-1} = 0$, wechsle in den Zustand $s_i = 1$.
- $s_i = 1$: Wenn immer noch $s_{i-1} = 0$ ist, wechsle in den Zustand $s_i = 2$. Ansonsten lasse deinem Vorgänger den Vortritt und wechsle zurück nach $s_i = 0$.
- $s_i = 2$: Sobald $s_{i+1} = 0$ gilt, wechsle in den Zustand $s_i = 3$.
- $s_i = 3$: Wenn immer noch $r_i > r_{i+1}$ gilt (Knoten $i-1$ oder Knoten $i+1$ könnten Werte verändert haben), dann tausche den Inhalt von r_i und r_{i+1} und wechsle wieder nach $s_i = 0$. Ansonsten gehe nach $s_i = 0$ ohne die Werte zu tauschen.

Wie oben gezeigt, werden so die Werte in den Registern in endlicher Zeit aufsteigend sortiert, wenn der letzte Schritt des Algorithmus (aus $s_i = 3$) unter *gegenseitigem Ausschluss* mit den beiden Nachbarn des Knotens stattfindet. Zusätzlich

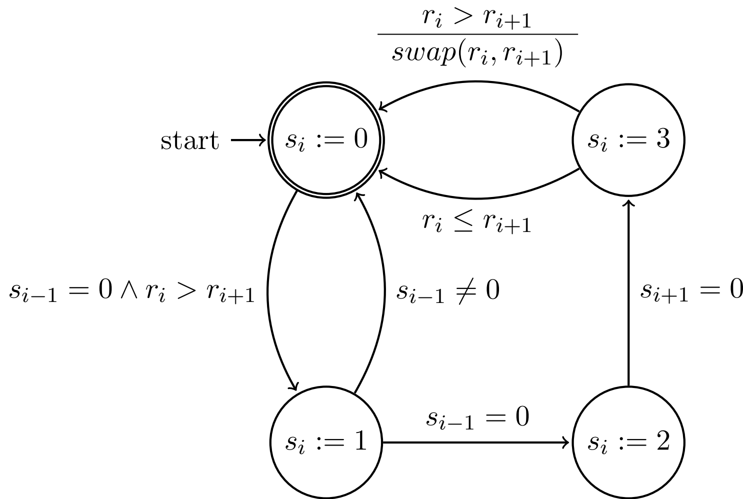


Abbildung 4: Algorithmus für Knoten i

muss *Deadlock-* und *Verhungerungsfreiheit* garantiert werden. Der Beweis dieser Eigenschaften ist im Folgenden skizziert.

Beweis(skizze).

- $\forall\ i \in \{0, 1, \ldots, n-1\}(s_i = 3 \Rightarrow (s_{i-1} \neq 3 \wedge s_{i+1} \neq 3))$: Jeder Knoten verlässt $s_i = 0$ *bevor* der Zustand der Nachbarn überprüft wird und wechselt zu $s_i = 3$ erst *nach* dieser Überprüfung.

- *Verhungerungsfreiheit*: Es kann induktiv leicht gezeigt werden, dass nach endlicher Zeit die Werte in den Registern r_j für alle $j < i \vee j > i$ sortiert sind und Knoten i bei Bedarf Werte tauschen kann. (Es ist dann kein Guard eines anderen Prozesses mehr aktiviert, so dass dieser Prozess mit einem Berechnungsschritt an der Reihe ist.)

- *Verklemmungsfreiheit*: Da der erste Knoten keinen Vorgänger und der letzte keinen Nachfolger hat, ist zyklisches Warten ausgeschlossen. Ansonsten sind Knoten mit niedrigerer Nummer priorisiert.

□

6.2 Fehlermodell

Wir lassen alle Fehler zu, die den Wert eines (Zustands-)registers verändern. Dies schließt also sowohl die r_i als auch die s_i ein. Dadurch können nacheinander alle Variablen verändert werden, allerdings höchstens eine pro aufgetretenem Fehler. Die Zustandsregister s_{-1} und s_n sind ausgenommen. Sie existieren nur konzeptionell und bleiben zu jeder Zeit 0.

6.3 Gütefunktion

Wir werden die Güte dieses Sortieralgorithmus dadurch bestimmen, wie viele der ursprünglichen Werte, die sich zu Anfang in den Registern r_i befunden haben, noch in einem Register r_j enthalten sind, beziehungsweise nicht in Folge von Fehlern verfälscht worden sind. Wir nehmen also eine Güte von 100% an, wenn kein Wert verloren gegangen ist, und 0%, wenn alle Werte korrumpiert worden sind.

Dazu werden die Endzustände des Systems zunächst formal definiert:

Definition 6.1 (Endzustände)**.** *Eine Konfiguration $\bar{c}$ heißt* Endzustand *aus c, wenn sie durch eine endliche Anzahl an Schritten aus $\mathcal{A}$ erreichbar ist und aus $\bar{c}$ keine von $\bar{c}$ unterschiedliche Konfiguration mit einem Schritt aus $\mathcal{A}$ mehr erreicht werden kann. Die Menge der Endzustände aus c bezeichnen wir also als*

$$Fin(c) := \big\{\bar{c} \mid \exists\, c', c'', \cdots : c \to_{\mathcal{A}} c' \to_{\mathcal{A}} c'' \to_{\mathcal{A}} \cdots \to_{\mathcal{A}} \bar{c} \\ \wedge \forall \tilde{c} : \bar{c} \to_{\mathcal{A}} \tilde{c} \Rightarrow \tilde{c} = \bar{c}\big\} \quad (6.1)$$

Falls $|Fin(c)| = 1$, sei

$$fin(c) := \bar{c} \quad \textit{mit } Fin(c) = \{\bar{c}\} \quad (6.2)$$

Da der Bubblesort-Algorithmus aus jeder Konfiguration c in genau eine Konfiguration terminiert, können wir folgende Gütefunktion wählen:

$$g(c) := \frac{1}{n+1} \max_{\pi \in \Pi} \big|\{i \mid fin(c).r_i = c_0.r_{\pi(i)}\}\big| \quad (6.3)$$

Dabei sei Π die Menge aller Permutationen[3] auf der Menge $\{0, 1, \ldots, n\}$ und c_0 die Startkonfiguration. Ein Algorithmus, der zu einer gegebenen Konfiguration c die Güte $g(c)$ für den *Bubblesort*-Algorithmus berechnet, würde ungefähr so vorgehen:

1. Führe den Algorithmus von c aus ohne Fehler fort, bis keine Schritte aus $\mathcal{A}$ mehr möglich sind. Dadurch erhält man die Endkonfiguration $fin(c)$.

2. Versuche nun, so viele Startwerte (Registerinhalte aus c_0) wie möglich Registerinhalten aus $fin(c)$ zuzuordnen.

3. Gib das Verhältnis der Anzahl der zugeordneten Werte zu der Gesamtzahl der Werte zurück.

Eine Implementation dessen findet sich in Listing 3 (Abschnitt 7).

[3] Eine Permutation ist eine Abbildung, die die Elemente einer Menge gegeneinander vertauscht, z.B. $\pi(0) = 1, \pi(1) = 2, \pi(2) = 0$.

6.4 Beweis der Regionentreue

Im Folgenden werden wir beweisen, dass das System $\lfloor \frac{n}{2} \rfloor$-regionentreu bezüglich der in Abschnitt 6.2 beschriebenen Fehlerklasse $\mathcal{F}$ und der in Abschnitt 6.3 beschriebenen Gütefunktion g ist. Wir zeigen dazu, dass in Folge eines Fehlers maximal zwei zu sortierende Werte verloren gehen.

Um zu zeigen, dass ein Algorithmus (oder ein System) regionentreu ist, wird hier die in Lemma 4.1 verschärfte Form von Regionentreue angenommen. Ausgehend von diesem Lemma kann nun einfach bewiesen werden, dass der vorgestellte *Bubblesort*-Algorithmus regionentreu ist. Der Vorteil von Lemma 4.1 ist, dass nur noch ein Berechnungs- und ein Fehlerschritt allgemein betrachtet werden müssen, aber keine gesamten Ausführungen.

Beweis. Das System erfüllt alle in Lemma 4.1 genannten Eigenschaften.

1. Da der Algorithmus korrekt ist (es gehen insbesondere ohne Fehler keine Werte verloren), kann die sortierte Folge durch eine Permutation wieder auf die initiale Konfiguration abgebildet werden und es gilt

$$g(c_0) = \frac{1}{n+1} \max_{\pi \in \Pi} \left| \left\{ i \mid fin(c_0).r_i = c_0.r_{\pi(i)} \right\} \right| \tag{6.4}$$

$$= \frac{1}{n+1}(n+1) = 1 \tag{6.5}$$

2. Wenn aus einer beliebigen Konfiguration c ein Schritt aus dem Algorithmus in die Konfiguration c' führt ($c \rightarrow_{\mathcal{A}} c'$), gilt nach Definition 6.1 $fin(c') = fin(c)$ und somit

$$g(c') = \frac{1}{n+1} \max_{\pi \in \Pi} \left| \left\{ i \mid fin(c').r_i = c_0.r_{\pi(i)} \right\} \right| \tag{6.6}$$

$$= \frac{1}{n+1} \max_{\pi \in \Pi} \left| \left\{ i \mid fin(c).r_i = c_0.r_{\pi(i)} \right\} \right| \tag{6.7}$$

$$= g(c) \tag{6.8}$$

3. Wenn aus einer beliebigen Konfiguration c ein Fehler in die Konfiguration c' führt ($c \rightarrow_{\mathcal{F}} c'$), können wir Folgendes annehmen:

   - Falls ein Register r_i verändert wurde, ist der Wert, der darin stand, verloren. Die anderen Register bleiben auch in der Folge unberührt.
   - Falls ein Zustandsregister s_i verändert wurde, ist Knoten i so lange fehlerhaft, bis er sich wieder in einem Zustand mit $s_i = 0$ befindet. Bis dahin kann er beim Tauschen zweier Registerinhalte aber höchstens zwei Werte verändern. (Wir gehen davon aus, dass zum Tauschen r_i und r_{i+1} jeweils einmal beschrieben werden und der Tauschvorgang nicht atomar ist. An-

dere Vertauschungsalgorithmen, für die dies nicht gilt, können zu einer anderen Güte führen.)

In beiden Fällen gilt

$$g(c') = \frac{1}{n+1} \max_{\pi \in \Pi} \left|\{i \mid fin(c').r_i = c_0.r_{\pi(i)}\}\right| \tag{6.9}$$

$$\geq \frac{1}{n+1} \left(\max_{\pi \in \Pi} \left|\{i \mid fin(c).r_i = c_0.r_{\pi(i)}\}\right| - 2 \right) \tag{6.10}$$

$$= g(c) - \underbrace{\frac{2}{n+1}}_{=:\Delta} \tag{6.11}$$

Wegen $f = \left\lfloor \frac{n}{2} \right\rfloor < \frac{n+1}{2}$ gilt $\frac{1}{f} > \Delta = \frac{2}{n+1}$.

Damit ist der Algorithmus regionentreu für maximal $\left\lfloor \frac{n}{2} \right\rfloor$ Fehler bezüglich der Gütefunktion g und Fehlerklasse $\mathcal{F}$. □

7 Evaluation des Bubblesort-Algorithmus

Im Folgenden werden wir nun den *Bubblesort*-Algorithmus mithilfe des Modelcheckers und Simulators SPIN [14] simulieren. Dazu werden mit einem speziell in Java entwickelten Tool Testfälle generiert, diese mit SPIN ausgeführt und die Ausgabe des Simulators automatisch verarbeitet. Die Ergebnisse werden dann mit der Statistiksoftware R [19] ausgewertet und visualisiert.

7.1 Implementierung des Beispielalgorithmus

Wir übersetzen den Zustandsautomaten aus Abbildung 4 in *Guarded Commands*. Die einzelnen Zustände werden dabei direkt durch die Variable s_i, $i = 0, 1, \ldots, n-1$, kodiert. Es ergibt sich so für P_i mit $1 \leq i \leq n-2$:

do

$s_i = 0 \wedge s_{i-1} = 0 \wedge r_i > r_{i+1} \rightarrow s_i := 1$

$\square$

$s_i = 1 \wedge s_{i-1} \neq 0 \rightarrow s_i := 0$

$\square$

$s_i = 1 \wedge s_{i-1} = 0 \rightarrow s_i := 2$

$\square$

$s_i = 2 \wedge s_{i+1} = 0 \rightarrow s_i := 3$

$\square$

$s_i = 3 \wedge r_i > r_{i+1} \rightarrow \textsc{swap}(r_i, r_{i+1});\ s_i := 0$

$\square$

$s_i = 3 \wedge r_i \leq r_{i+1} \rightarrow s_i := 0$

od

Zum Tauschen von r_i und r_{i+1} benutzen wir folgenden Algorithmus. Es sind tmp_a und tmp_b Lokale Hilfsvariablen des Knotens. Es werden zwei temporäre Variablen benutzt, um *read/write atomicity* zu simulieren. Nach dem verwendeten Berechnungsmodell gehen wir davon aus, dass bei einer Zuweisung $a := b$ zunächst b gelesen wird und in einem weiteren Berechnungsschritt der gelesene Wert in a gespeichert wird. Dazwischen kann allerdings ein anderer Prozess den Wert in b verändern. Der Simulator von SPIN behandelt eine einzelne Anweisung allerdings atomar, weshalb dieser Fall in der Simulation ohne die Verwendung einer Hilfsvariable nicht auftreten kann. Da wir ihn aber simulieren wollen, benutzen wir eine zusätzliche Hilfsvariable tmp_b.

procedure SWAP(a, b: Variable)
  $tmp_a := a$;
  $tmp_b := b$;
  $a := tmp_b$;
  $b := tmp_a$;
end procedure

7.2 Implementierung in PROMELA

Die oben in Form von Guarded Commands angegebene Übersetzung des Zustandsautomaten kann eins zu eins nach PROMELA übernommen werden. Er ist nachfolgend angegeben.

```
1  proctype Node (int me)
   {
3      do
       :: (state[me] == 0) && (r[me] > r[me+1]) -> state[me] = 1
5      :: (state[me] == 1) && (state[me-1] != 0) -> state[me] = 0
       :: (state[me] == 1) && (state[me-1] == 0) -> state[me] = 2
7      :: (state[me] == 2) && (state[me+1] == 0) -> state[me] = 3
       :: (state[me] == 3) && (r[me] > r[me+1]) -> {
9                  int tmp, tmp2;
                   tmp = r[me];
11                 tmp2 = r[me+1];
                   r[me] = tmp2;
13                 r[me+1] = tmp;
                   state[me] = 0
15             }
       :: (state[me] == 3) && (r[me] <= r[me+1]) -> state[me] = 0
17     od
   }
```

Listing 1: PROMELA Code für Knoten P_i, $1 \leq i \leq n-2$

Für die Randknoten sieht die Übersetzung ähnlich aus. Da s_{-1} und s_n als konstant angenommen werden, fallen für P_0 die Zeilen 5 und 6 weg, für P_{n-1} Zeile 7. Der gesamte PROMELA-Code ist im Anhang aufgeführt.

7.2.1 Fault Injection

Um auftretende Fehler zu simulieren, betreiben wir *Fault Injection* [5]. Wir erstellen einen weiteren Prozess, der unsere Fehlerklasse simuliert. Unter der Annahme, dass der Scheduler im Simulator fair ist[4], wird dieser Prozess in regelmäßigen Abständen Variableninhalte verändern.

Die maximale Länge der Simulation setzen wir immer auf 10 000 Schritte, was auf jeden Fall ausreicht, einen Testfall mit bis zu zehn Fehlern durchzuführen. Da der Prozess, der die Fehler simuliert, immer noch einen neuen Fehler injizieren kann, sobald der Sortieralgorithmus terminiert, begrenzen wir in jedem Test die Fehlerzahl. Die Simulation wird dann zusammen mit dem Sortieralgorithmus terminieren, sobald diese Fehlerzahl erreicht worden ist.

Der Algorithmus zur Fault-Injection geht folgendermaßen vor:

Solange die gewählte Fehlerzahl nicht erreicht ist,

1. wähle einen zufälligen Index $i \in \{0, 1, \ldots, N\}$,
2. wähle einen zufälligen Wert $v \in \{0, 1, \ldots, \texttt{MAX_VALUE} - 1\}$ und
3. setze $r_i := v$

oder

1. wähle einen zufälligen Index $i \in \{0, 1, \ldots, N - 1\}$,
2. wähle einen zufälligen Wert $v \in \{0, 1, 2, 3\}$ und
3. Setze $s_i := v$.

Um „zufällige“ Werte dafür zu erzeugen, wird in der Implementierung ein Algorithmus benutzt, der gleichverteilte Zahlen aus $\{0, 1, \ldots, 511\}$ zurück gibt. Im Sprachumfang der verwendeten PROMELA-Version ist zwar ein `select`-Statement enthalten, das eine Variable nichtdeterministisch mit einem Wert aus einem beliebigen Bereich belegt, diese liefert aber scheinbar keine gleichverteilten Werte. Der stattdessen verwendete PROMELA-Code ist nachfolgend angegeben. Die Funktionsweise ist die, dass jede Zahl $x \in \{0, 1, \ldots, 511\}$ eindeutig als Summe

$$x = \sum_{k=0}^{8} a_k 2^k, \quad \text{mit } a_k \in \{0, 1\} \tag{7.1}$$

dargestellt werden kann. Im verwendeten **do** . . . **od**-Konstrukt gibt es je einen Fall für $a_k = 0$ und $a_k = 1$, die beide den gleichen Guard haben. Unter der Annahme, dass beide mit gleicher Wahrscheinlichkeit ausgewählt werden, sind die Rückgabewerte folglich gleichverteilt.

[4] Im Referenzmanual zu SPIN [14] findet sich dazu keine Angabe. Die Verwendung des Pseudo-Zufallsgenerators legt aber eine statistische Gleichverteilung bei der Auswahl aktivierter Anweisungen nahe.

```
inline random (x)
{
    int rand = 0;
    x = 256;
    do
    :: x > 0 -> rand = rand + x; x = x / 2
    :: x > 0 -> x = x / 2
    :: x == 0 -> x = rand; break
    od
}
```

Listing 2: Funktion zum Erzeugen gleichverteilter Pseudozufallszahlen

Da der Algorithmus mit $N = 5$ simuliert wird, werden (wegen $512 \mod 5 = 2$ und $512 \mod 6 = 2$) die Knoten mit Indices 0 und 1 geringfügig öfter ausgewählt. Dies sollte die Ergebnisse aber nicht verfälschen.

Eine alternative Möglichkeit wäre auch hier die Verwendung eines (in PROMELA implementierten) linearen Kongruenzgenerators mit im Testfall festgelegtem Seed oder die verwendeten Zufallszahlen direkt im Testfall anzugeben. In [14] wird aber die hier gewählte Möglichkeit empfohlen, eine nichtdeterministische Auswahl zu benutzen. So kann das PROMELA-Modell auch zum Model-Checking benutzt werden. Eine nichtdeterministische Auswahl der Form

if
  true $\rightarrow x := 0$
  $\square$
  true $\rightarrow x := 1$
  $\square$
  $\vdots$
  $\square$
  true $\rightarrow x := 511$
fi

die für jeden möglichen Wert einen *Guarded Command* besitzt, ist in der Ausführung zwar noch schneller, wurde aber wegen der Länge des Quellcodes nicht in Betracht gezogen.

7.3 Erstellen von Testfällen

Die Testfälle werden zufällig von einem speziell in Java entwickelten Tool erstellt. Ein Teil des Initialisierungscodes – die Initialisierung der r_i – wird vom Tool als Inline-Funktion generiert und in den PROMELA-Code des Modells importiert.

7.3.1 Wahl der Startwerte

Initialisiert werden die r_i mit gleichverteilten Zahlen aus $\{0, 1, \ldots, \texttt{MAX_VALUE} - 1\}$, die durch die Standardfunktion `Math.random()` erzeugt werden. Für die vorliegenden Tests wurde $\texttt{MAX_VALUE} = 32$ gewählt. Da `Math.random()` Fließkommazahlen (`double`) liefert, werden Ganzzahlige Werte durch

```
(int) (Math.random()*MAX_VALUE)
```

erzeugt.

Das Tool legt für jeden Testfall eine Datei `insert1.pml` an, in der eine Inline-Funktion `setup()` deklariert wird. Diese belegt das Array `r` mit den Startwerten r_i und wird zu Beginn der Simulation aufgerufen. Zusätzlich werden in der Datei die Konstanten für die Fehlerzahl und den Wertebereich der r_i vereinbart.

Um die Testreihen bei Bedarf reproduzieren zu können, wird der *Seed* des Pseudozufallsgenerators, der von SPIN während der Simulation zum Auflösen nichtdeterministischer Auswahlen genutzt wird, für jeden Testlauf durch den Komandozeilenparameter `-n`*seed* neu gesetzt. Dieser initiale Seed ist Bestandteil der Testdaten und wird zusammen mit ihnen gespeichert. Auch hier wird `Math.random()` verwendet. Es wird ein Seed zwischen 1 und 10000 erzeugt. Da SPIN eine Version des *Minimal Standard Generator* benutzt, sind alle Werte zwischen 1 und $2^{31} - 2$ gleichermaßen geeignet [18, 1]. (Dies ist nämlich ein linearer Kongruenzgenerator mit maximaler Periode.)

7.4 Auswertung der Testreihen

Jeder der so generierten Tests wird mit SPIN ausgeführt. Die Ausgabe des Simulators beinhaltet unter anderem die Belegung der Variablen am Ende der Simulation. Die Ausgabe wird mittels Regulärer Ausdrücke geparst. Aus dem Vergleich mit den Startwerten kann die Güte des Algorithmus nach dem letzten Fehler berechnet werden.

7.4.1 Berechnung der Güte

Wie bereits oben beschrieben, wird bei jedem Testfall festgelegt, wie viele Fehler auftreten.

Da die Gütefunktion hier so definiert ist, dass die Güte nach dem letzten Fehler aus der Belegung der Variablen am Ende des Sortieralgorithmus berechnet werden kann, können wir also aus der Variablenbelegung nach der Simulation jeweils die Güte nach der gewählten Anzahl Fehlern bestimmen. Die Variablenbelegung wird vom Simulator zum Schluss ausgegeben und die initiale Belegung ist durch den Testfall bekannt. Die Funktion, mit der daraus die Güte berechnet werden kann, ist in Java implementiert und folgend angegeben.

```
1 private static double quality(int[] init, int[] fin) {
```

```
    fin = Arrays.copyOf(fin, fin.length);
    int matched = 0;
    for (int v: init) {
        for (int j = 0; j < fin.length - matched; j++) {
            if (fin[j] == v) {
                matched++;
                fin[j] = fin[fin.length - matched];
                break;
            }
        }
    }
    return ((double) matched) / ((double) fin.length);
}
```

Listing 3: Java-Code zur Berechnung der Güte aus der initialen Belegung der Register `init` und der abschließenden Belegung `fin`.

7.4.2 Statistische Analyse mit R

Die Testergebnisse werden vom Java-Tool in tabellarischer Form gespeichert. Pro Testfall werden folgende Daten ausgegeben:

- `nProc` – Anzahl der Simulierten Prozesse N
- `nFault` – Anzahl der injizierten Fehler
- `seed` – Seed, der an Spin zur Simulation übergeben wurde
- `quality` – Ermittelte Güte
- `input` – Initiale Belegung der r_i
- `output` – Belegung der r_i, nachdem der Algorithmus terminiert ist

Bei der Auswertung werden nur die Fehlerzahl und die Güte berücksichtigt. `input` und `seed` werden angegeben, um den Test bei Bedarf wiederholen zu können. `output` dient nur der Kontrolle – Hieran kann z.B. die Berechnung der Güte überprüft werden oder die erwartete Eigenschaft, dass die Werte zum Schluss auf jeden Fall geordnet sind. Während der Entwicklung half die Ausgabe außerdem, einen Fehler im Promela-Modell aufzudecken, dass die Funktion `random` immer 0 zurückgegeben hatte. Im abschließend verwendeten Code wurde der Fehler korrigiert.

Die Angabe der Prozesszahl ist hier zwar unnötig, da nur Tests mit $N + 1 = 6$ zu sortierenden Werten durchgeführt wurden. Wenn allerdings Ergebnisse verschiedener Tests gemeinsam in einer Datei abgelegt werden, können die Ergebnisse daran gefiltert werden.

Bei der Auswertung interessiert uns hier vor allem das *average case*-Verhalten des Systems. Das *worst case*-Verhalten – also der theoretisch schlimmste Fall – ist bereits

theoretisch betrachtet worden. Interessant ist noch, ob und mit welcher Wahrscheinlichkeit er eingetreten ist.

Bei diesem Algorithmus fällt auf, dass die Gütefunktion nur Werte der Menge

$$G = \left\{ \frac{k}{N+1} \;\middle|\; k = 0, 1, \dots N+1 \right\} \tag{7.2}$$

annehmen kann. (Dies geht aus der Gütefunktion selbst hervor.) Wir können daher die Fehlerzahl gegenüber der Güte in einer *Häufigkeitstabelle* (Tabelle 2) abtragen. In Abbildung 5 ist Tabelle 2 noch einmal in Form einer *Heatmap* dargestellt. Hier ist bereits zu erahnen, dass der *worst case* nicht eingetreten ist. Da der Algorithmus für $N = 5$ regionentreu mit

$$\Delta = \frac{2}{N+1} = \frac{1}{3} \tag{7.3}$$

ist, ist der *worst case* eine Güte von

- $1 = 100\%$ ohne Fehler
- $\frac{2}{3} \approx 66{,}7\%$ bei einem Fehler
- $\frac{1}{3} \approx 33{,}3\%$ bei zwei Fehlern
- 0 bei drei oder mehr Fehlern

Diese unteren Schranken werden bei Weitem nicht erreicht, wie sich schon aus Tabelle 2 ablesen lässt. Noch besser lassen sich die Daten in einem *erweiterten Boxplot* (Abbildung 7) zusammenfassen. Hier ist besonders der *Median*, der durch die Linie in der Mitte einer Box markiert ist, interessant. Er legt die Vermutung nahe, dass die Güte im Mittel ungefähr doppelt so hoch wie die untere Grenze der in den Tests erreichten Güte ist. Die mittlere Güte scheint vier mal so hoch wie der *worst case*.

Ebenfalls werden hier die Extremwerte (bzw. Ausreißer) durch Kreise über und unter der Box Markiert. Man sieht noch einmal, dass der *worst case* nicht erreicht wurde. Im *best case* ist die Güte auch bei wachsender Fehlerzahl 100%.

Um das Verhältnis aus mittlerer Güte und Fehlerzahl quantifizieren zu können, modellieren wir dieses grob als linearen Zusammenhang. Es scheint zu gelten

$$\mathbb{E}(g(c) | \#_{\mathcal{F}}(c_0, \dots, c)) \approx a \#_{\mathcal{F}}(c_0, \dots, c) + b \tag{7.4}$$

Eine lineare Regression mit R ergibt

$$a \approx -0.05 \tag{7.5}$$

$$b \approx 0.96 \tag{7.6}$$

Häufigkeit	Güte→							mittlere
↓Fehler	0,0	0,17	0,33	0,5	0,67	0,83	1,0	Güte
0	0	0	0	0	0	0	2000	1,000
1	0	0	0	0	0	920	1080	0,923
2	0	0	0	0	378	1086	536	0,846
3	0	0	0	97	683	917	303	0,786
4	0	0	19	248	830	725	178	0,733
5	0	3	65	414	885	548	85	0,680
6	0	17	160	543	838	401	41	0,631
7	3	25	224	699	736	290	23	0,592
8	1	26	307	758	705	191	12	0,563
9	10	106	408	771	567	132	6	0,517
10	13	157	458	819	460	90	3	0,487

Tabelle 2: Absolute Häufigkeiten der Güte geordnet nach Fehlerzahl

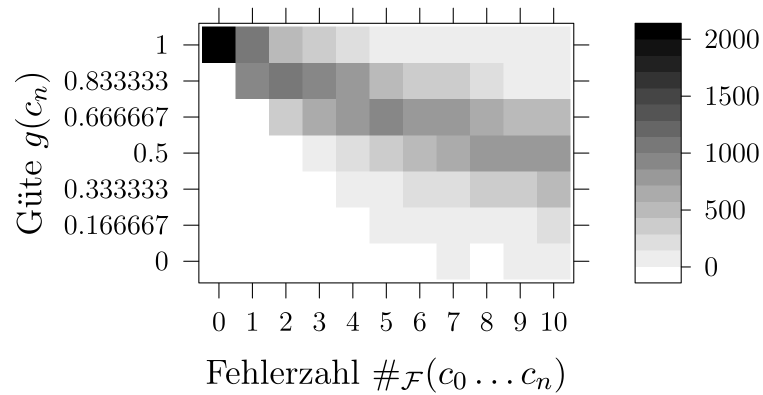


Abbildung 5: Heatmap der Güte vs. Fehlerzahl

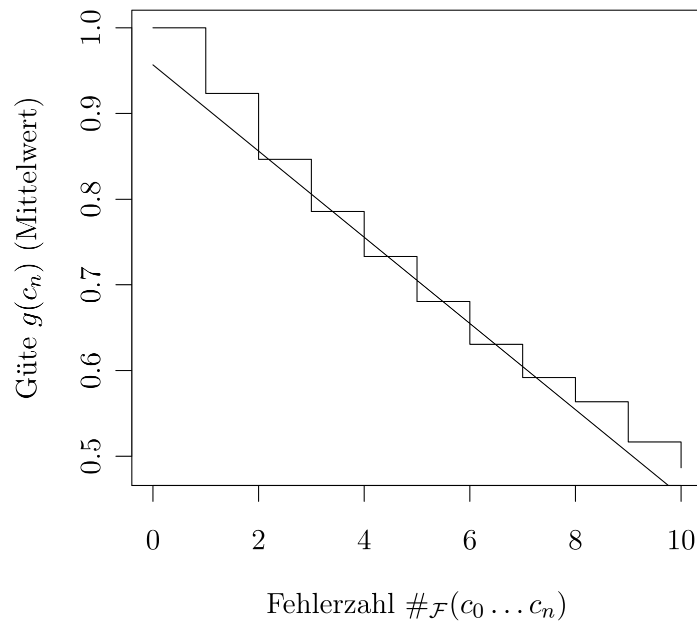


Abbildung 6: Arithmetisches Mittel der Güte vs. Fehlerzahl. Zusätzlich ist die Regressionsgerade eingezeichnet

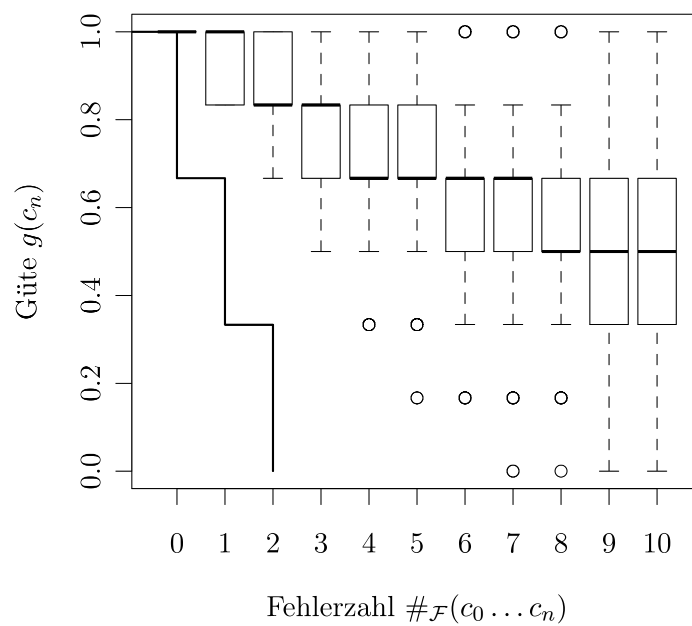


Abbildung 7: Boxplot der Güte vs. Fehlerzahl. Der theoretisch bestimmte *worst case* ist als Treppenfunktion eingezeichnet.

8 Verbesserung der Regionentreue mittels replizierter Variablen

Eine gängige Methode in der maskierenden Fehlertoleranz ist die Strategie, Variablen, die durch Fehler korrumpiert werden können, zu vervielfachen. Dies wird auch *n-Modular-Redundancy* genannt, eine Verallgemeinerung der bereits erwähnten *Triple-Modular-Redundancy* (TMR). Dazu wird dann ein weiteres Modul, ein *Voter*, eingebaut, der fehlerfrei und atomar zum Lesen einer Variable alle Instanzen dieser Variable ließt und den Wert zurück gibt, der mehrheitlich vorkommt. Beim Schreiben wird der Wert atomar in alle Instanzen der Variable geschrieben.

Wenn nun eine (oder eine Minderheit) der Instanzen einer Variable durch Fehler korrumpiert wird, wird durch den Voter immer noch der richtige Wert gelesen.

Diese Strategie lässt sich – mit einer kleinen Verschärfung des Voter-Verhaltens – einsetzen, um einen f-regionentreuen Algorithmus in einen $(n+1)f$-regionentreuen Algorithmus zu verwandeln. Dies geschieht nach folgender Strategie.

Replizierung der Variablen. Für jede Variable r_i des Algorithmus benutze statt einer $2n+1$ Versionen[5], $r_i^0, r_i^1, \dots r_i^{2n}$. Der Wertebereich der Variablen bleibt gleich.

Modifikation des Algorithmus. Beim Lesen und Schreiben wird das oben beschriebene Verhalten des Voters verwendet. Allerdings ist es im Allgemeinen nötig, dass beim Lesen zugleich alle beschädigten Instanzen der Variable „repariert“ werden, sprich auf den gevoteten Wert gesetzt werden. Dazu definieren wir zwei Funktionen READ und WRITE auf einer beliebigen Variable. Es sei $maj(r_i)$ der Wert, der in der Mehrheit der $r_i^0, r_i^1, \dots r_i^{2n}$ steht. Wenn mehrere Werte am häufigsten vorkommen, wird ein beliebiger ausgewählt.

function READ(r: Variable)
  $\langle r^0 := r^1 := \dots := r^{2n} := maj(r);$
  return $r^0;\rangle$
end function
procedure WRITE(r: Variable, x: Value)
  $\langle r^0 := r^1 := \dots := r^{2n} := x;\rangle$
end procedure

Der Algorithmus wird nun so verändert, dass jeder Lesezugriff auf eine Variable r_i durch READ(r_i) und jeder Schreibzugriff $r_i := x$ durch WRITE(r_i, x) ersetzt wird.

Initialisierung. Die initiale Konfiguration des neuen Algorithmus entspricht der des alten; alle Instanzen einer Variable enthalten den gleichen Wert.

[5] Im Folgenden werden die Indices der Versionen immer hochgestellt.

Fehlerklasse. Die Fehlerklasse, die wir annehmen, verändert sich eigentlich nicht. Bisher nehmen wir an, dass ein Fehler eine beliebige Variable korrumpiert. Nun wird angenommen, dass eine beliebige Instanz einer Variable korrumpiert wird.

8.1 Beweis der Notwendigkeit des Reparierens beim Lesen

Es ist tatsächlich nötig, dass beim Lesen alle Instanzen einer Variable angeglichen werden.

Beweis. Man nehme ein System mit einer bool'schen Variable b und einer bool'schen Variable a an, von der $2n+1$ Instanzen $a^0, a^1, \ldots a^{2n}$ vorhanden sind. a und b sind initial **false**. Zusätzlich gibt es eine Variable f, die ganzzahlige Werte aufnimmt. Es wird folgender Algorithmus ausgeführt:

$f := 0;$
do
  $\langle b \neq maj(a) \rightarrow b := maj(a);\ f := f+1 \rangle$
od

Fehler betreffen nur a. Es wird zufällig eine Instanz von a negiert. Wir haben die Fehlerklasse

do
  true $\rightarrow a^i := \neg a^i$ für ein beliebiges $0 \leq i \leq 2n$
od

In f werden also Fehler gezählt. Nehmen wir den Fall an, dass durch Fehler zunächst $a^0 = a^1 = \cdots = a^{n-1} =$ **true** gesetzt werden, aber $a^n = a^{n+1} = \cdots = a^{2n} =$ **false** bleiben. Es bleibt $maj(a) =$ **false**. Fortlaufend verändern Fehler immer die Instanz a^n, die dann den Ausschlag bei der Auswertung von $maj(a)$ gibt. Nach dem nächsten Fehler ist $maj(a) =$ **true**, dann wieder $maj(a) =$ **false**, dann $maj(a) =$ **true**, und so weiter. Nach jedem Fehler mache nun das System einen Berechnungsschritt. Der Wert von f wird also hochgezählt. Ein Zustand mit $f = k$, $k > 0$, kann also bereits schon nach $n + k$ Fehlern erreicht werden. Wir streben aber allgemein eine Verbesserung zu mindestens $(n + 1)k$ Fehlern an. □

8.2 Beweis der Verbesserung auf $(n+1)f$

Nachfolgend zeigen wir, dass ein nach obiger Strategie modifizierter Algorithmus $(n+1)f$-regionentreu ist. Wir zeigen dazu, dass zu einer beliebigen Konfiguration $c \in \mathcal{C}$ aus dem alten Algorithmus, die nicht mit weniger als k Fehlern erreichbar ist, eine analoge Konfiguration $\overline{c} \in \overline{\mathcal{C}}$ mit $maj(\overline{c}) = c$ in dem neuen Algorithmus frühestens mit $(n + 1)k$ Fehlern erreicht werden kann. ($maj(\overline{c})$ bezeichnet die Konfiguration, die durch Voting auf allen Variablen entsteht.)

Es sei $\overline{\mathcal{A}}$ der modifizierte Algorithmus zu $\mathcal{A}$ und $\overline{\mathcal{F}}$ die neue (oben beschriebene) Fehlerklasse. Im Folgenden schreiben wir statt $\#_{\mathcal{F} \setminus \mathcal{A}}$ kurz $\#_{\mathcal{F}}$.

Lemma 8.1. *Es sei $c_m \in \mathcal{C}$ eine beliebige Konfiguration und $\gamma = c_0 \dots c_m$ eine Ausführung, die c mit der minimal möglichen Anzahl $k = \#_{\mathcal{F}}(\gamma)$ Fehlern erreicht. Dann ist $\overline{c}_m \in \overline{\mathcal{C}}$, $maj(\overline{c}_m) = c$, mit $(n+1)k$ Fehlern (oder mehr) erreichbar.*

Beweis. Es ist klar, dass eine solche minimale Ausführung keine Fehlerschritte enthält, die in den Berechnungsschritten liegen. Ansonsten könnte man diese als Berechnungsschritte interpretieren und so eine kleinere Fehlerzahl erreichen. Erzeuge $\overline{\gamma} = \overline{c}_0 \dots \overline{c}_m$ aus γ indem jede Konfiguration c_k, $k > 0$ mit

- $c_{k-1} \rightarrow_{\mathcal{A}} c_k$ durch $\overline{c}_k$ mit $maj(\overline{c}_k) = c_k$ und $\overline{c}_{k-1} \rightarrow_{\overline{\mathcal{A}}} \overline{c}_k$ ersetzt wird.
- $c_{k-1} \rightarrow_{\mathcal{F}} c_k$ durch $\overline{c}^1_{k-1}\overline{c}^2_{k-1} \dots \overline{c}^n_{k-1}\overline{c}_k$ mit $maj(\overline{c}_k) = c_k$ und $\overline{c}_{k-1} \rightarrow_{\overline{\mathcal{F}}} \overline{c}^1_{k-1} \rightarrow_{\overline{\mathcal{F}}} \overline{c}^2_{k-1} \rightarrow_{\overline{\mathcal{F}}} \dots \rightarrow_{\overline{\mathcal{F}}} \overline{c}^n_{k-1} \rightarrow_{\overline{\mathcal{F}}} \overline{c}_k$ ersetzt wird.

Ein einzelner Fehler in dem ursprünglichen System wird so durch $n+1$ Fehler ersetzt, die zusammen mehr als die Hälfte der Instanzen einer Variable korrumpieren. Im ursprünglichen System wird bei $c \rightarrow_{\mathcal{F}} c'$ eine Variable verändert. Im neuen System verändert jeder Fehler eine einzelne Instanz, weshalb nach $n+1$ Fehlern die Variable beim Voting einen anderen Wert annimmt (den nach dem Fehlerschritt im ursprünglichen System) und also $maj(\overline{c}_k) = c_k$ gilt. Es ist so $\#_{\overline{\mathcal{F}}}(\overline{\gamma}) = (n+1)k$. □

Lemma 8.2. *Bei einem durch Replikation verbesserten System kann eine analoge Konfiguration nicht mit weniger als $(n+1)k$ Fehlern erreicht werden.*

Beweis. Es sei $c_m \in \mathcal{C}$ beliebig und k die minimale Fehlerzahl, mit der c_m erreicht werden kann. Ferner sei $\overline{\gamma} = \overline{c}_0 \dots \overline{c}_m$ eine Ausführung, die $\overline{c}_m \in \overline{\mathcal{C}}$ mit $maj(\overline{c}_m) = c_m$ mit der niedrigst möglichen Anzahl Fehlern, $\#_{\overline{\mathcal{F}}}(\overline{\gamma}) = k'$, erreicht. Wir zeigen $k' \geq (n+1)k$.

Es seien $i_1, i_2, \dots i_\ell$ die Indices aus $\overline{\gamma}$ mit $maj(\overline{c}_{i_j-1}) \neq maj(\overline{c}_{i_j})$ und $\overline{c}_{i_j-1} \rightarrow_{\overline{\mathcal{F}}} \overline{c}_{i_j}$. Die Stellen also, bei denen Fehler dann nach außen Auswirkungen zeigen.

Da die Zahl der Fehler in γ minimal ist, muss zwischen zwei Fehlern, die die selbe Variable korrumpieren, diese Variable vom Algorithmus gelesen werden. (Ansonsten wäre der erste Fehler ohne Auswirkungen und kann daher weggelassen werden. Die Zahl der Fehler in γ wäre damit also nicht minimal.) Damit sich $maj(r)$ für eine Variable r bei einem Fehler ändert, müssen demnach jedes Mal mindestens $n+1$ Instanzen von r verändert worden sein. Es ist also $\#_{\overline{\mathcal{F}}}(\overline{\gamma}) = k' \geq (n+1)\ell$.

Erzeuge nun $\gamma = c_0 \dots maj(\overline{c}_m)$ aus $\overline{\gamma}$, indem alle $\overline{c}_i$, $i > 0$ mit $\overline{c}_{i-1} \rightarrow_{\overline{\mathcal{F}}} \overline{c}_i$ und $maj(\overline{c}_{i-1}) = maj(\overline{c}_i)$ weggelassen werden und alle anderen $\overline{c}_i$ durch $maj(\overline{c}_i)$ ersetzt werden.

γ ist dann eine gültige Ausführung für $\mathcal{A}$ mit $\#_{\mathcal{F}}(\gamma) = \ell$. Da c_m nicht mit weniger als k Fehlern erreichbar ist, gilt $\ell \geq k$. Zusammen mit $k' \geq (n+1)\ell$ folgt $k' \geq (n+1)k$. □

Theorem 8.1. *Die Einführung von replizierten Variablen überführt einen f-regionentreuen in einen $(n+1)f$-regionentreuen Algorithmus.*

Beweis. Dies folgt direkt aus den soeben eingeführten Lemmata 8.1 und 8.2.

Es sei

$$\#_{\mathcal{F}}(c) := \min\{\#_{\mathcal{F}}(\gamma) \mid \gamma = c_0 \ldots c \text{ Ausführung}\}. \tag{8.1}$$

Wie oben gezeigt, gilt

$$\#_{\mathcal{F}}(maj(\overline{c})) \leq \frac{\#_{\overline{\mathcal{F}}}(\overline{c})}{n+1}. \tag{8.2}$$

Die Definition von Regionentrue sagt

$$g(c) \geq \#_{\mathcal{F}}(c)\Delta, \tag{8.3}$$

woraus also direkt

$$\overline{g}(\overline{c}) = g(maj(\overline{c})) \geq \#_{\overline{\mathcal{F}}}(\overline{c})\frac{\Delta}{n+1} \tag{8.4}$$

folgt. Wegen $\Delta < \frac{1}{f}$ gilt $\frac{\Delta}{n+1} < \frac{1}{(n+1)f}$, weshalb $\overline{\mathcal{A}}$ $(n+1)f$-regionentreu bezüglich $\overline{g}$ und $\overline{\mathcal{F}}$ ist. □

8.3 Anwendung und Simulation

Diese Technik wollen wir nachfolgend für $n = 1$ auf den bereits bekannten Bubblesort-Algorithmus anwenden und mit dem gleichen Experiment wie in Abschnitt 7 evaluieren.

Bei $n = 1$ benutzen wir statt einer Instanz einer Variable jeweils drei Instanzen, also eine Form der *Triple Modular Redundancy*, kurz *TMR*. Das Majority-Voting wird dabei zu

$$maj(r) := \begin{cases} r^0 & \text{falls } r^0 = r^1 \\ r^2 & \text{sonst} \end{cases} \tag{8.5}$$

Falls $r^0 = r^1$ gilt, ist der Wert aus r^0 folglich mehrheitlich – in mindestens zwei von drei Instanzen – enthalten. Andernfalls ist entweder $r^0 = r^2$ oder $r^1 = r^2$ – dann steht r^2 in mindestens zwei Instanzen –, oder aber alle Instanzen beinhalten unterschiedliche Werte. In diesem Fall wählen wir einfach ebenfalls r^2.

Um dies in Promela zu implementieren, definieren wir zunächst einen neuen Datentyp, der die drei Instanzen eines Registers kapselt.

```
typedef register {
2     int a = 0;
      int b = 0;
4     int c = 0
}
```

Im letzten Abschnitt haben wir die beiden atomaren Funktionen read und write definiert, die dazu verwendet werden, die Regionentreue eines Algorithmus zu verbessern. Die write-Funktion lässt sich als Makro formulieren. Wenn wir die read-Funktion bei der Implementierung eines Algorithmus in Promela direkt einsetzen wollen, stoßen wir allerdings auf zwei Probleme:

- Funktionsaufrufe gehören nicht zum Sprachumfang von PROMELA und lassen sich auch nur begrenzt als Makro ausdrücken.

- Guards bzw. Ausdrücke in PROMELA müssen frei von Seiteneffekten sein.

Um diese zu lösen, führen wir das *Majority-Voting* und das „Reparieren“ der Variablen getrennt aus. Die oben definierte Funktion maj lässt sich in PROMELA durch einen einzigen Ausdruck formulieren. Wir definieren auch dazu ein Makro.

Im vorherigen Abschnitt wird allerdings gefordert, dass bei jedem Lesen einer Variable diese „repariert“ wird. Wir können diese Forderung zumindest für Guards soweit abschwächen, dass wir nur die Variablen „reparieren“ müssen, die in dem ausgewählten Guard gelesen werden, aber nicht die in den anderen Guards eines **if**- oder **do**-Konstrukts. Diese Annahme können wir treffen, da keine Reihenfolge vorgegeben ist, in der aktivierte Guarded Commands, bzw. Prozesse, ausgewählt werden. Ob noch der Guard irgendeines nicht aktivierten Commands ausgewertet wurde, ist deshalb für einen gültigen Programmverlauf irrelevant und wir nehmen an, dass nur der ausgewählte Guard ausgewertet wurde. Eine Ausnahme bilden natürlich **else**-Zweige, da diese genau dann ausgeführt werden, wenn kein anderer Guard aktiviert ist. Dafür müssen dann alle Guards ausgewertet werden.

Das „Reparieren“ der Variablen eines Guards wird zur ersten Anweisung jedes Commands und wird in PROMELA zusammen mit dem Guard in einen `atomic`-Block eingeschlossen.

Wir definieren die Makros

```
1 #define READ(R)      ( R .a == R .b -> R .a : R .c )
  #define WRITE(R,X)   atomic { R .a = (X) ; R .b = (X) ; R .c = (X) }
3 #define REPAIR(R)    WRITE(R, READ(R))
```

Listing 4: Makrodefinitionen

Der PROMELA-Code für Knoten 1 bis $n-1$ wird damit zu

```
1 proctype Node (int me)
  {
3     do
      :: atomic{(READ(state[me]) == 0) && (READ(r[me]) > READ(r[me+1])) ↩
          -> REPAIR(state[me]); REPAIR(r[me]); REPAIR(r[me+1])}; WRITE(↩
          state[me], 1)
5     :: atomic{(READ(state[me]) == 1) && (READ(state[me-1]) != 0) -> ↩
          REPAIR(state[me]); REPAIR(state[me-1]);} WRITE(state[me], 0)
      :: atomic{(READ(state[me]) == 1) && (READ(state[me-1]) == 0) -> ↩
          REPAIR(state[me]); REPAIR(state[me-1]);} WRITE(state[me], 2)
7     :: atomic{(READ(state[me]) == 2) && (READ(state[me+1]) == 0) -> ↩
          REPAIR(state[me]); REPAIR(state[me+1]);} WRITE(state[me], 3)
      :: atomic{(READ(state[me]) == 3) && (READ(r[me]) > READ(r[me+1])) ↩
          ->
9                 REPAIR(state[me]); REPAIR(r[me]); REPAIR(r[me+1])}
                  int tmp, tmp2;
```

```
11                  tmp = READ(r[me]);
                    tmp2 = READ(r[me+1]);
13                  WRITE(r[me], tmp2);
                    WRITE(r[me+1], tmp);
15                  WRITE(state[me], 0)

17      :: atomic{(READ(state[me]) == 3) && (READ(r[me]) <= READ(r[me+1])) ↩
          -> REPAIR(state[me]); REPAIR(r[me]); REPAIR(r[me+1])}; WRITE(↩
          state[me], 0)
        od
19  }
```

Listing 5: Bubblesort mit replizierten Variablen

Die Simulation läuft auf die gleiche Weise wie in Abschnitt 7 ab. Die Unterschiede sind nur minimal:

- Der Fault-Injection-Prozess verändert nun eine zufällige Instanz `r.a`, `r.b` oder `r.c` eines Registers `r`.
- Durch den Testfall, bzw. die `setup`-Inline-Funktion, müssen alle Instanzen einer Variable initialisiert werden.
- Es wird bei der Berechnung der Güte nach jedem Test das Majority-Voting aus den Endbelegungen der Variablen herangezogen. Ansonsten wird die Funktion `quality` (Listing 3) nicht verändert.
- Wir führen Tests mit bis zu 30 statt zuvor 10 Fehlern durch. Durch die Replizierung der Variablen sollten jetzt mindestens doppelt so viele Fehler toleriert werden.

8.3.1 Auswertung der Testreihen des Algorithmus mit TMR

Die Testreihen werden auf die gleiche Art wie in Abschnitt 7 ausgewertet. Die Daten werden in Abbildung 8 bis 10 visualisiert.

Durch die Replikation der Variablen, die wir vorgenommen haben, sollte sich ein mindestens 4-*regionentreuer* Algorithmus ergeben, bzw. der Algorithmus regionentreu mit $\Delta = \frac{1}{6}$ sein. (Zuvor wurde $\Delta = \frac{1}{3}$ bewiesen.) Aus dem Boxplot (Abbildung 10) ist ersichtlich, dass diese Grenze eingehalten wird. Allein schon ein rein visueller Vergleich der Simulationsergebnisse (Abbildung 8 vs. Abbildung 5) zeigt, dass die Güte des verbesserten Algorithmus durchschnittlich der Güte des einfachen Algorithmus bei halber Fehlerzahl entspricht. Eine Regressionsgerade (Abbildung 9) ergibt hier die Relation

$$\mathbb{E}(g(c)|\#_{\mathcal{F}}(c_0,\ldots,c)) \approx a\#_{\mathcal{F}}(c_0,\ldots,c) + b \tag{8.6}$$

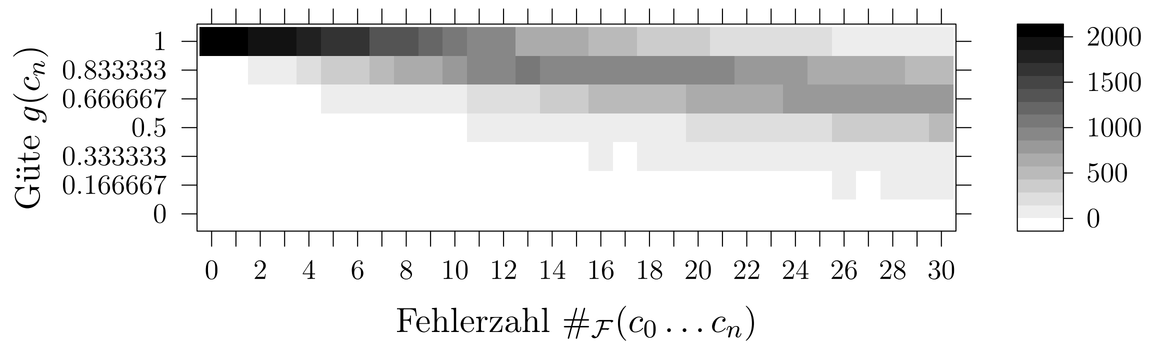


Abbildung 8: Heatmap der Güte vs. Fehlerzahl. Die Färbung der Kästchen gibt an, wie oft die entsprechende Kombination aus Güte und Fehlerzahl aufgetreten ist. Weiß: 0-mal, schwarz: 2000-mal

mit

$$a \approx -0.012 \tag{8.7}$$

$$b \approx 1.02 \tag{8.8}$$

Die Steigung (b) ist weniger als halb so groß.

Durch die Verbesserungen des Algorithmus wurde die Regionentreue wie erwartet verbessert.

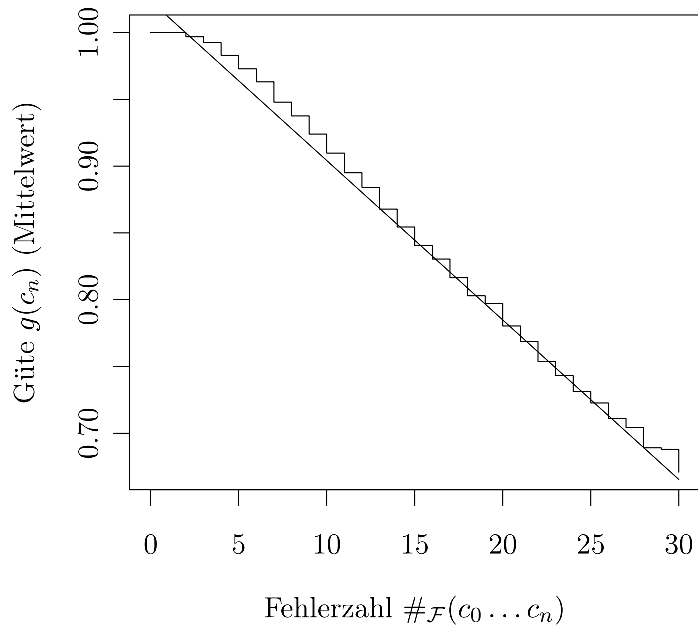


Abbildung 9: Arithmetisches Mittel der Güte vs. Fehlerzahl. Zusätzlich ist die Regressionsgerade eingezeichnet.

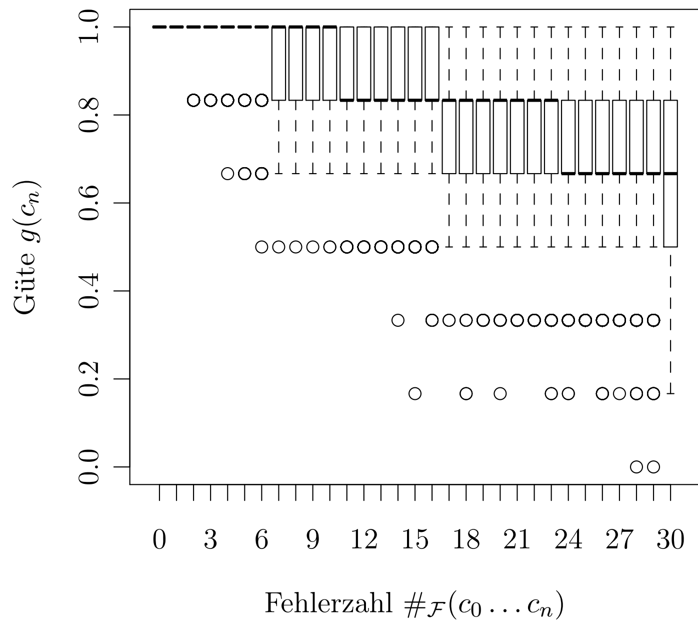


Abbildung 10: Boxplot der Güte vs. Fehlerzahl

9 Average Case Betrachtungen

Bisher haben wir nur den *worst case* bei regionentreuen Algorithmen betrachtet, d.h.

- wie viele Fehler können minimal toleriert werden, bzw. wann wird frühestens eine Güte von 0% erreicht?
- wie hoch ist die Güte nach k Fehlern mindestens?

Im Folgenden werden wir nun zwei *average case* Betrachtungen anstellen, und zwar

- wann wird durchschnittlich die Güte auf 0% sinken?
- wie hoch ist die Güte bei langer Laufzeit des Systems durchschnittlich?

9.1 Mean Time To Failure (MTTF)

Um die erste Frage zu klären, betrachten wir die mittlere Zeit, bis das System versagt. Dies wird durch die *Mean Time To Failure (MTTF)* ermittelt [15]. Die MTTF ist ein gängiges Maß für die Verfügbarkeit eines Systems [15].

Es sei $R(t)$ die Wahrscheinlichkeit, dass das System auf dem gesamten Zeitintervall $[0, t]$ verfügbar ist, sprich t Zeiteinheiten vom Start des Systems. (Dies wird auch *Reliability* genannt.) Dann ist die *Mean Time To Failure* [15, 17]

$$MTTF = \int_0^\infty R(t)\mathrm{d}t \tag{9.1}$$

Ein f-regionentreues System ist verfügbar, solange die Güte noch nicht 0 ist. Dies ist mindestens der Fall, solange die Zahl der aufgetretenen Fehler kleiner oder gleich f ist. Die Zahl der aufgetretenen Fehler ist ein *counting process* [17] und kann als *Poison-Prozess* modelliert werden. Es sei λ die *Fehlerrate*, also die durchschnittliche Anzahl Fehler, die pro Zeiteinheit auftreten. Die Wahrscheinlichkeit, dass nach t Zeiteinheiten k Fehler aufgetreten sind, ist damit

$$P(N(t) = k) = P_k(t) = \frac{(\lambda t)^k}{k!} e^{-\lambda t} \tag{9.2}$$

Daraus folgt für die *Reliability*

$$R(t) \geq \sum_{k=0}^{f} P_k(t) = e^{-\lambda t} \sum_{k=0}^{f} \frac{(\lambda t)^k}{k!} \tag{9.3}$$

Man beachte, dass das System auch nach mehr als f Fehlern noch lebendig sein kann. Wir leiten hier daher nur eine untere Grenze für die MTTF her.

Es folgt für die MTTF

$$MTTF \geq \frac{f+1}{\lambda} \tag{9.4}$$

Beweis. Es gilt

$$MTTF = \int_0^\infty R(t)\mathrm{d}t \geq \int_0^\infty e^{-\lambda t} \sum_{k=0}^{f} \frac{(\lambda t)^k}{k!}\mathrm{d}t = \sum_{k=0}^{f} \int_0^\infty e^{-\lambda t} \frac{(\lambda t)^k}{k!}\mathrm{d}t \tag{9.5}$$

Es sei $u := \lambda t$. Dann gilt durch Substitution

$$\int e^{-\lambda t} \frac{(\lambda t)^k}{k!}\mathrm{d}t = \frac{1}{\lambda} \int e^{-u} \frac{u^k}{k!}\mathrm{d}u \tag{9.6}$$

und durch partielle Integration

$$\int e^{-u} \frac{u^k}{k!}\mathrm{d}u = -e^{-u}\frac{u^k}{k!} - \int -e^{-u} \frac{u^{k-1}}{(k-1)!}\mathrm{d}u \tag{9.7}$$

$$= -e^{-u}\frac{u^k}{k!} + \int e^{-u} \frac{u^{k-1}}{(k-1)!}\mathrm{d}u \tag{9.8}$$

durch wiederholte Anwendung

$$= -e^{-u} \sum_{\ell=0}^{k} \frac{u^\ell}{\ell!} \tag{9.9}$$

Es folgt

$$MTTF \geq \frac{1}{\lambda} \sum_{k=0}^{f} \left(-\lim_{u\to\infty} \left(e^{-u} \sum_{\ell=0}^{k} \frac{u^\ell}{\ell!} \right) + e^0 \sum_{\ell=0}^{k} \frac{0^\ell}{\ell!} \right) \tag{9.10}$$

$$= \frac{1}{\lambda} \sum_{k=0}^{f} \sum_{\ell=0}^{k} \left(-\lim_{u\to\infty} \left(e^{-u} \frac{u^\ell}{\ell!} \right) + e^0 \frac{0^\ell}{\ell!} \right) \tag{9.11}$$

$$= \frac{1}{\lambda} \sum_{k=0}^{f} (f+1-k) \Bigg(\underbrace{-\lim_{u\to\infty} \left(e^{-u} \frac{u^k}{k!} \right)}_{=0} + e^0 \frac{0^k}{k!} \Bigg) \tag{9.12}$$

$$= \frac{1}{\lambda} \Bigg((f+1) \underbrace{e^0 \frac{0^0}{0!}}_{=1} + \sum_{k=1}^{f} (f+1-k) \underbrace{e^0 \frac{0^k}{k!}}_{=0} \Bigg) \tag{9.13}$$

$$= \frac{f+1}{\lambda} \tag{9.14}$$

□

9.2 Betrachtung mittels Markov-Ketten

Im folgenden Abschnitt betrachten wir ein System als *zeitdiskreten stochastischen Prozess*. Wir nehmen an, dass die möglichen Konfigurationen des Systems *abzählbar* sind und nummerieren diese der Einfachheit halber durch. Es existiert also eine

injektive[6] Funktion $\phi : \mathcal{C} \to \mathbb{N}$, die diese Nummerierung vornimmt. Wir betrachten weiter eine (unendliche) Ausführung des Systems als Zufallsvariable X. $X(k) = \phi(c)$ gibt an, dass sich das System nach k Schritten (gemeint sind sowohl Berechnungs- als auch Fehlerschritte) in der Konfiguration $c \in \mathcal{C}$ befindet.

Definition 9.1 (Markov-Kette [17]). *Ein stochastisches System ist eine* zeitdiskrete Markov-Kette*, wenn es* gedächnislos*, d.h. wenn ein Schritt unabhängig von den vorherigen Schritten ist. Es muss für beliebige* $n \in \mathbb{N}$ *und* $i_0, i_1, \dots, i_{n-1}, i, j \in \text{Bild}(\phi)$

$$\begin{aligned} P(X(n+1) = j | X(0) = i_0 \wedge X(1) = i_1 \wedge \dots \wedge X(n-1) = i_{n-1} \wedge X(n) = i) \\ = P(X(n+1) = j | X(n) = i) = p_{ij} \end{aligned} \tag{9.15}$$

gelten.

Die p_{ij} bilden eine *Übergangswahrscheinlichkeitsmatrix.*

$$\mathbf{P} = (p_{ij}) = \begin{pmatrix} p_{00} & p_{01} & p_{02} & \cdots \\ p_{10} & p_{11} & p_{12} & \\ p_{20} & p_{21} & p_{22} & \\ \vdots & & & \ddots \end{pmatrix} \tag{9.16}$$

p_{ij} gibt die Wahrscheinlichkeit an, in einem Schritt von der Konfiguration c mit $\phi(c) = i$ in die Konfiguration c' mit $\phi(c') = j$ zu gelangen. Man bezeichnet mit

$$p_{ij}^n := P(X(n+m) = j | X(m) = i) \tag{9.17}$$

die Wahrscheinlichkeit, mit n Schritten von c nach c' zu gelangen. Da eine Markov-Kette gedächnislos ist – die Wahrscheinlichkeit für einen Schritt ist unabhängig von vorherigen Schritten – ist es egal, nach wie vielen (m beliebigen) Schritten die betrachtete Kette von n Schritten beginnt. Folglich ist die Wahrscheinlichkeit in c' zu enden von m unabhängig. Die p_{ij}^n sind Komponenten der n-ten Potenz von $\mathbf{P}$. Es gilt

$$(p_{ij}^n) = \mathbf{P}^n = \underbrace{\mathbf{P} \cdot \mathbf{P} \cdots \mathbf{P}}_{n\text{-mal}} \tag{9.18}$$

Wir nennen eine Markov-Kette [17]

- *irreduzibel*, wenn

  $$\forall i,j \in \text{Bild}(\phi)\ \exists n \in \mathbb{N} : p_{ij}^n > 0 \tag{9.19}$$

  D.h. alle Konfigurationen sind von einer beliebigen anderen Konfiguration aus erreichbar.

[6] Eine Funktion f ist *injektiv*, wenn $f(x) = f(y) \Rightarrow x = y$ gilt.

- *rekurrent*, wenn

$$\forall i \in \text{Bild}(\phi) : \sum_{n=1}^{\infty} p_{ii}^n = \infty \tag{9.20}$$

D.h., dass in einer unendlichen Ausführung jeder Zustand unendlich oft durchlaufen werden kann.

- *aperiodisch*, wenn

$$\forall i \in \text{Bild}(\phi) : \text{ggT}\{n \in \mathbb{N} | p_{ii}^n > 0\} = 1 \tag{9.21}$$

Wenn eine Markov-Kette diese drei Eigenschaften erfüllt, wird sie *ergodisch* genannt. Dann hat jede Konfiguration eine *Aufenthaltswahrscheinlichkeit* p_i [17]. Dies ist die Wahrscheinlichkeit, dass sich das System nach ausreichend langer Laufzeit in einer Konfiguration c mit $\phi(c) = i$ befindet. Es gilt

$$\lim_{n \to \infty} \mathbf{P}^n = \begin{pmatrix} p_0 & p_1 & p_2 & \cdots \\ p_0 & p_1 & p_2 & \cdots \\ \vdots & \vdots & \vdots & \\ p_0 & p_1 & p_2 & \cdots \end{pmatrix} \tag{9.22}$$

Das Gleichungssystem

$$p \cdot \mathbf{P} = p \tag{9.23}$$

bzw.

$$p(\mathbf{P} - I) = 0 \tag{9.24}$$

mit $p = (p_0, p_1, p_2, \dots)$ hat dann eine eindeutige Lösung mit $p(1 \dots 1)^T = 1$ [17]. Es gelten

$$p_i > 0 \ (\forall i \in \text{Bild}(\phi)) \tag{9.25}$$

9.2.1 Folgerungen für die Güte

Was bedeutet dies für die Güte eines Systems? Bei einem regionentreuen Algorithmus stellt sich insbesondere die Frage, wie die mittlere Güte bei einer ausreichend langen Laufzeit des Systems ausfallen wird. Kann sich das System von Fehlern „erholen“, d.h. steigt die Güte nach einem Fehler wieder an, oder strebt die Güte durch Fehler unaufhaltsam gegen 0? Was ist im ersteren Fall die mittlere Güte des Systems?

Wenn wir die Konfiguration eines Systems nach n Schritten als Zufallsvariable $X(n) \in \mathbb{N}$ betrachten, können wir folglich auch die Güte als Zufallsvariable $G(n) = g(\phi^{-1}(X(n)))$ betrachten. Im vorherigen Abschnitt sind insbesondere Syste-

me eingeführt worden, zu deren Konfigurationen sich *Aufenthaltswahrscheinlichkeiten* p_i bestimmen lassen. Daraus ergibt sich folgendes Lemma:

Lemma 9.1. *Gegeben sei ein regionentreues System mit Gütefunktion g, das sich als* irreduzible, rekurrente *und* azyklische *Markov-Kette mit Zustandsübergangswahrscheinlichkeitsmatrix* $\mathbf{P}$ *beschreiben lässt. Dann folgt für die Güte $G(n) = g(\phi^{-1}(X(n)))$:*

- *Nach ausreichend langer Laufzeit ist der Erwartungswert der Güte*

$$\mathbb{E}(G) := \lim_{n\to\infty} \mathbb{E}(G(n)) = \sum_i g(\phi^{-1}(i))p_i \tag{9.26}$$

- *und die Varianz*

$$Var(G) := \lim_{n\to\infty} Var(G(n)) = \sum_i (g(\phi^{-1}(i)) - \mathbb{E}(G))^2 p_i \tag{9.27}$$

mit Aufenthaltswahrscheinlichkeiten p_i, also $p = (p_0, p_1, \dots)$ und $p(\mathbf{P} - I) = 0$.

Beweis. Dies folgt direkt aus den zuvor beschriebenen Eigenschaften von Markov-Ketten und der Definition von Erwartungswert bzw. Varianz. □

Insbesondere gilt in diesem Fall

$$\mathbb{E}(G) > 0. \tag{9.28}$$

Beweis. Es gilt für die initiale Konfiguration $g(c_0) = 1$ und, da die Markov-Kette irreduzibel ist, $p_{\phi(c_0)} > 0$. Es folgt

$$\mathbb{E}(G) = \sum_i g(\phi^{-1}(i))p_i \geq g(c_0)p_{\phi(c_0)} > 0 \tag{9.29}$$

□

9.2.2 Mean Time To Failure

Aus der Zustandsübergangswahrscheinlichkeitsmatrix können wir neben der erwarteten Güte und deren Varianz auch die *Mean Time To Failure* berechnen.

Die MTTF berechnet sich aus $\int_0^\infty R(t)\mathrm{d}t$. Da wir hier aber ein diskretes Modell betrachten, wandeln wir dies zu

$$MTTF = \sum_{k=0}^{\infty} R(k) \tag{9.30}$$

Die *Reliability* $R(k)$ ist die Wahrscheinlichkeit, dass das System bis einschließlich Zeitpunkt k nur Zustände mit einer Güte größer 0 durchlaufen hat. Diese fassen wir in der Menge

$$S := \{\phi(c) | c \in \mathcal{C}, g(c) > 0\} \tag{9.31}$$

zusammen. Daraus ergibt sich

$$R(n) := P(X(0), X(1), \dots, X(n) \in S) \tag{9.32}$$

Mit p^n_{Sij} bezeichnen wir die Wahrscheinlichkeit, in n Schritten von Zustand i nach j zu gelangen, und dabei nur Zwischenzustände aus S zu erreichen. Es gilt folglich

$$p^{n+1}_{Sij} = P(X(n+1) = j | X(0) = i \wedge X(1), X(2), \dots, X(n) \in S) \tag{9.33}$$

$$= \sum_{s \in S} p^n_{Sis} p_{sj} \tag{9.34}$$

mit

$$p^0_{Sij} = \begin{cases} 1 & \text{für i = j} \\ 0 & \text{sonst} \end{cases} \tag{9.35}$$

In Matrixform folgt

$$(p^n_{Sij}) = (\mathbf{SP})^n \tag{9.36}$$

wobei

$$\mathbf{S} := \begin{pmatrix} s_0 & 0 & 0 & \dots \\ 0 & s_1 & 0 & \dots \\ 0 & 0 & s_2 & \\ \vdots & \vdots & & \ddots \end{pmatrix} \qquad \text{mit } s_i = \begin{cases} 1 & \text{falls } s_i \in S \\ 0 & \text{sonst} \end{cases} \tag{9.37}$$

Da das System per Definition in c_0 startet, ist letztendlich

$$R(n) = \sum_{j \in S} p^n_{S\phi(c_0)j} = (\pi_0 \ \pi_1 \ \dots)(\mathbf{SP})^n (s_0 \ s_1 \ \dots)^\mathsf{T} \tag{9.38}$$

mit den s_i wie oben und

$$\pi_i = \begin{cases} 1 & \text{für } i = \phi(c_0) \\ 0 & \text{sonst} \end{cases} \tag{9.39}$$

Aus all dem folgt für die Mean Time To Failure

$$MTTF = \sum_{k=0}^{\infty} R(k) \tag{9.40}$$

$$= (\pi_0 \ \pi_1 \ \dots) \left(\sum_{k=0}^{\infty} (\mathbf{SP})^k \right) (s_0 \ s_1 \ \dots)^\mathsf{T} \tag{9.41}$$

$$= \begin{cases} (\pi_0 \ \pi_1 \ \dots)(I - \mathbf{SP})^{-1} (s_0 \ s_1 \ \dots)^\mathsf{T} & \text{falls } (\mathbf{SP})^n \to 0 \\ \infty & \text{sonst} \end{cases} \tag{9.42}$$

Falls $(\mathbf{SP})^n \to 0$, gilt obige Gleichung nach [22]. Dies ist insbesondere dann der Fall, wenn aus jedem Zustand nach endlich vielen Schritten ein Zustand erreicht werden kann, der nicht in S liegt. Intuitiv geht dann die Wahrscheinlichkeit bei wachsender Schrittzahl nur Zustände aus S zu besuchen gegen null. Andernfalls geht die Summe, da alle Potenzen von $\mathbf{SP}$ nur positive Einträge enthalten, gegen unendlich.

Der erste Fall tritt folglich insbesondere bei ergodischen – und damit irreduziblen – Matrizen ein, sofern es erreichbare Zustände $i \notin S$ gibt.

9.2.3 Zusammenfassung des Zustandsraums

Um die Güte eines Systems zu bewerten, sind die konkreten Variableninhalte in einem Zustand meist uninteressant. Die Güte ergibt sich aus den Eigenschaften der Variableninhalte.

Dies lässt sich anhand des Beispiels aus Abschnitt 6 verdeutlichen: Für die Güte des Sortieralgorithmus ist nicht interessant, welche konkreten Werte zu einem Zeitpunkt in den Registern stehen, sondern vielmehr, wie viele davon den ursprünglich zu sortierenden Werten entsprechen. So lässt sich der ursprünglich sehr große Zustandsraum (die Änderung eines Variableninhalts stellt ja in jedem Fall eine neue Konfiguration dar) in $n+2$ verschiedene Gruppen einteilen, die sich jeweils durch die Zahl der zu sortierenden Werte ergeben, die aufgrund von Fehlern „verloren gegangen“ sind. Es ergeben sich die Gruppen von keinem verlorenen Wert mit einer Güte von 1 bis hin dazu, dass alle $n+1$ Werte korrumpiert worden sind. Diese Gruppe hat die Güte 0.

Bisher wurde gefordert, dass die Funktion ϕ, die die Konfigurationen durchnummeriert, injektiv ist. Damit ist sichergestellt, dass jeder Zahl auch wieder eine eindeutige Konfiguration zugeordnet werden kann. Nach obiger Argumentation ist dies aber nicht nötig. Es reicht aus, dass wir einer Zahl – sprich einem Zustand in der betrachteten Markov-Kette – eine Güte zuordnen können. Daher kann die Injektivität bis hin zu der Bedingung

$$\phi(c) = \phi(c') \Rightarrow g(c) = g(c') \tag{9.43}$$

für beliebige Konfigurationen c und c' abgeschwächt werden. Genauer gilt Folgendes:

Wenn ein System mit der Zufallsvariable $X(t) = \phi(c)$, wenn sich das System zum Zeitpunkt t im Zustand c befindet, eine zeitdiskrete Markov-Kette ist, so ist das System mit der Zufallsvariable $\widetilde{X}(t) = \widetilde{\phi}(c)$ (wenn $X(t) = \phi(c)$) und der (nicht-injektiven) Nummerierungsfunktion $\widetilde{\phi} : \mathcal{C} \to \mathbb{N}$ ebenfalls eine zeitdiskrete Markov-

Kette, solange für beliebige $c, c' \in \mathcal{C}$ die Gleichungen

$$\phi(c) = \phi(c') \Rightarrow \widetilde{\phi}(c) = \widetilde{\phi}(c') \tag{9.44}$$

$$\widetilde{\phi}(c) = \widetilde{\phi}(c') \Rightarrow g(c) = g(c') \tag{9.45}$$

$$\begin{aligned} \widetilde{\phi}(c) = \widetilde{\phi}(c') \Rightarrow \forall i \in \text{Bild}(\widetilde{\phi}) : \\ \sum_{\widetilde{\phi}(c'')=i} P(X(t+1) = \phi(c'')|X(t) = \phi(c)) \\ = \sum_{\widetilde{\phi}(c'')=i} P(X(t+1) = \phi(c'')|X(t) = \phi(c')) \end{aligned} \tag{9.46}$$

gelten.

Beweis. Es ist $i \sim j :\Leftrightarrow \widetilde{\phi}(c) = \widetilde{\phi}(c')$ (mit $\phi(c) = i$ und $\phi(c') = j$) eine Äquivalenzrelation. Mit Gleichung (9.46) ist $\sim$ *bisimilar*, weshalb die zu den Äquivalenzklassen von $\sim$ (bzw. mittels $\widetilde{\phi}$) zusammengefassten Zustände wieder eine Markov-Kette bilden [13]. □

9.2.4 Beispiel

Im Folgenden werden wir die Berechnung der mittleren Güte und ihrer Varianz anhand eines einfachen 1-regionentreuen Algorithmus verdeutlichen.

Die Zustände des Systems bestehen aus drei bool'schen Variablen, sind also von der Form

$$c = (x_0, x_1, x_2) \in \mathcal{C} := \{0,1\}^3 \tag{9.47}$$

Ziel des Algorithmus ist, dass die Variablen reihum auf 1 gesetzt werden. Die stabile Zustandsfolge ist

$$\cdots \to (1,0,0) \to (0,1,0) \to (0,0,1) \to (1,0,0) \to (0,1,0) \to (0,0,1) \to (1,0,0) \ldots$$

Um die Güte eines Zustands zu bewerten, wählen wir die kleinste *Hamming-Distanz* [11] des Zustands zu einem der stabilen Zustände (1,0,0), (0,1,0) und (0,0,1). Die *Hamming-Distanz* zweier bool'scher Vektoren $a = a_0, a_1, \ldots, a_n$ und $b = b_0, b_1, \ldots, b_n$ ist definiert als

$$D(a,b) := |\{i | a_i \neq b_i\}| \tag{9.48}$$

Daraus berechnen wir die Güte eines Zustands c mit

$$g(c) := 1 - \frac{1}{2} \min\{D(c,x) | x \in \{(1,0,0), (0,1,0), (0,0,1)\}\} \tag{9.49}$$

$$= 1 - \frac{1}{2}|x_0 + x_1 + x_2 - 1| \tag{9.50}$$

Es ergeben sich die in Tabelle 3 aufgeführten Werte:

Der Algorithmus ist intuitiv und in Guarded Commands wie folgt:

c	(0,0,0)	(1,0,0)	(0,1,0)	(1,1,0)	(0,0,1)	(1,0,1)	(0,1,1)	(1,1,1)
$g(c)$	0,5	1	1	0,5	1	0,5	0,5	0

Tabelle 3: Güte der Zustände

do

$x_0 = 1 \rightarrow x_0 := 0; x_1 = 1$

$\square$

$x_1 = 1 \rightarrow x_1 := 0; x_2 = 1$

$\square$

$x_2 = 1 \rightarrow x_2 := 0; x_0 = 1$

$\square$

$x_0 = x_1 = x_2 = 0 \rightarrow x_0 := 1$

od

Die Startkonfiguration sei

$$c_0 := (1, 0, 0) \tag{9.51}$$

Als Fehlermodell lassen wir „Bitflips“ zu: Wir nehmen an, dass ein Fehler immer genau eine Variable invertiert. Also

do

true $\rightarrow x_0 := 1 - x_0$

$\square$

true $\rightarrow x_1 := 1 - x_1$

$\square$

true $\rightarrow x_2 := 1 - x_2$

od

Der Algorithmus ist in der Tat 1-regionentreu.

Beweis. Seien c und c' zwei aufeinanderfolgende Konfigurationen.

- Aus Tabelle 3 und 4 ergibt sich für Berechnungsschritte $g(c') \geq g(c)$.
- Für Fehlerschritte gilt $D(c, c') = 1$. Mit der Dreiecksungleichung $D(x, z) \leq D(x, y) + D(y, z)$ [11] für beliebige x, y, z gilt

$$g(c) - g(c') = 1 - \frac{1}{2}\min\{D(c,x)|x \in S\} - (1 - \frac{1}{2}\min\{D(c',x)|x \in S\}) \tag{9.52}$$

$$= \frac{1}{2}(\min\{D(c',x)|x \in S\} - \min\{D(c,x)|x \in S\}) \qquad \leq \frac{1}{2}(\min\{D(c,x)|x \in S \tag{9.53}$$

$$\leq \frac{1}{2} \tag{9.54}$$

mit $S := \{(1, 0, 0), (0, 1, 0), (0, 0, 1)\}$

- Es ist $g(c_0) = 1$.

Damit sind die Bedingungen für Lemma 4.1 erfüllt. $\square$

Der Algorithmus ist zugleich randomisiert-selbststabilisierend (Dies sind Algorithmen, die sich unter der Annahme eines fairen Schedulers statistisch nach endlicher Zeit stabilisieren [8]). Den Beweis dieser Eigenschaft werden wir hier nicht führen, die möglichen Zustandsübergänge durch Berechnungsschritte legen dies aber nahe. Es kann mittels zweier Berechnungsschritte immer eine stabile Konfiguration erreicht werden.

Um die Zustände zu nummerieren, interpretieren wir diese einfach als Binärzahl. Es sei also

$$\phi(x_0, x_1, x_2) := x_0 + 2x_1 + 4x_2 \tag{9.55}$$

In Tabelle 4 sind die möglichen Zustände, ihre möglichen Folgezustände und deren Nummern im Falle eines Berechnungsschritts angegeben.

Zustand→mögliche Folgezustände	Nummer des Zustands→Nummer der Folgezustände
(0,0,0)→(1,0,0)	0→1
(1,0,0)→(0,1,0)	1→2
(0,1,0)→(0,0,1)	2→4
(1,1,0)→(0,1,0), (1,0,1)	3→2, 5
(0,0,1)→(1,0,0)	4→1
(1,0,1)→(1,0,0), (0,1,1)	5→1, 6
(0,1,1)→(1,1,0), (0,0,1)	6→3, 4
(1,1,1)→(1,1,0), (1,0,1), (0,1,1)	7→3, 5, 6

Tabelle 4: Mögliche Folgezustände bei Berechnungsschritten

Wir nehmen an, dass jeder aktivierte Guard mit gleicher Wahrscheinlichkeit ausgewählt wird. Wir werden zunächst zwei getrennte Zustandsübergangsmatrizen $\mathbf{P}_{\mathcal{A}}$ für Berechnungsschritte und $\mathbf{P}_{\mathcal{F}}$ für Fehlerschritte aufstellen.

Es ergibt sich die Zustandsübergangsmatrix für Berechnungsschritte

$$\mathbf{P}_{\mathcal{A}} = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & 0 & 0 & \frac{1}{2} & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & \frac{1}{2} & 0 \\ 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{3} & 0 & \frac{1}{3} & \frac{1}{3} & 0 \end{pmatrix} \tag{9.56}$$

Aus dem Fehlermodell ergibt sich analog

$$\mathbf{P}_{\mathcal{F}} = \begin{pmatrix} 0 & \frac{1}{3} & \frac{1}{3} & 0 & \frac{1}{3} & 0 & 0 & 0 \\ \frac{1}{3} & 0 & 0 & \frac{1}{3} & 0 & \frac{1}{3} & 0 & 0 \\ \frac{1}{3} & 0 & 0 & \frac{1}{3} & 0 & 0 & \frac{1}{3} & 0 \\ 0 & \frac{1}{3} & \frac{1}{3} & 0 & 0 & 0 & 0 & \frac{1}{3} \\ \frac{1}{3} & 0 & 0 & 0 & 0 & \frac{1}{3} & \frac{1}{3} & 0 \\ 0 & \frac{1}{3} & 0 & 0 & \frac{1}{3} & 0 & 0 & \frac{1}{3} \\ 0 & 0 & \frac{1}{3} & 0 & \frac{1}{3} & 0 & 0 & \frac{1}{3} \\ 0 & 0 & 0 & \frac{1}{3} & 0 & \frac{1}{3} & \frac{1}{3} & 0 \end{pmatrix} \tag{9.57}$$

Wir nehmen an, dass Fehler mit einer Wahrscheinlichkeit von $\lambda = 0{,}1$ geschehen. Es folgt also für die Zustandsübergangswahrscheinlichkeitsmatrix

$$\mathbf{P} = 0{,}1\mathbf{P}_{\mathcal{F}} + 0{,}9\mathbf{P}_{\mathcal{A}} \approx \begin{pmatrix} 0 & 0{,}933 & 0{,}033 & 0 & 0{,}033 & 0 & 0 & 0 \\ 0{,}033 & 0 & 0{,}9 & 0{,}033 & 0 & 0{,}033 & 0 & 0 \\ 0{,}033 & 0 & 0 & 0{,}033 & 0{,}9 & 0 & 0{,}033 & 0 \\ 0 & 0{,}033 & 0{,}483 & 0 & 0 & 0{,}45 & 0 & 0{,}033 \\ 0{,}033 & 0{,}9 & 0 & 0 & 0 & 0{,}033 & 0{,}033 & 0 \\ 0 & 0{,}483 & 0 & 0 & 0{,}033 & 0 & 0{,}45 & 0{,}033 \\ 0 & 0 & 0{,}033 & 0{,}45 & 0{,}483 & 0 & 0 & 0{,}033 \\ 0 & 0 & 0 & 0{,}333 & 0 & 0{,}333 & 0{,}333 & 0 \end{pmatrix} \tag{9.58}$$

Im Folgenden werden wir den Erwartungswert und die Varianz der Güte bei längerer Laufzeit berechnen. Zunächst einmal ist die mittels $\mathbf{P}$ beschriebene Markov-Kette *ergodisch*.

Beweis. In den Potenzen $\mathbf{P}^3$ und $\mathbf{P}^4$ sind alle Einträge strikt größer 0. Die Markov-Kette ist also

- *irreduzibel*, da

$$p_{ij}^3 > 0 \forall i, j = 0, 1, \ldots, 7 \tag{9.59}$$

- *aperiodisch*, da

$$p_{ii}^3 > 0, p_{ii}^4 > 0 \forall i = 0, 1, \ldots, 7. \tag{9.60}$$

  Es folgt

$$\forall i = 0, 1, \ldots, 7 : \{3, 4\} \subset \{n \in \mathbb{N} | p_{ii}^n > 0\} \tag{9.61}$$

  und mit $\mathrm{ggT}\{3, 4\} = 1$

$$\forall i = 0, 1, \ldots, 7 : \mathrm{ggT}\{n \in \mathbb{N} | p_{ii}^n > 0\} = 1 \tag{9.62}$$

- *rekurrent*, da die Markov-Kette *endlich*, *irreduzibel* und *aperiodisch* ist [17].

□

Gleichung (9.24) ergibt für die *Aufenthaltswahrscheinlichkeiten*

$$p \approx (0{,}0286 \quad 0{,}2949 \quad 0{,}2855 \quad 0{,}0370 \quad 0{,}2768 \quad 0{,}0370 \quad 0{,}0366 \quad 0{,}0037) \tag{9.63}$$

Zusammen mit den Werten aus Tabelle 3 ergibt sich

- der *Erwartungswert* der Güte

$$\mathbb{E}(G) = \sum_{i=0}^{7} g(\phi^{-1}(i))p_i = \\ g(0,0,0)p_0 + g(1,0,0)p_1 + \cdots + g(1,1,1)p_7 \approx 0{,}927 \tag{9.64}$$

- und deren *Varianz*

$$Var(G) = \sum_{i=0}^{7} (g(\phi^{-1}(i)) - \mathbb{E}(G))^2 p_i \approx 0{,}0333 \tag{9.65}$$

Da die Markov-Kette irreduzibel ist, ergibt sich die MTTF nach Gleichung (9.42)

$$MTTF = (0\ 1\ 0\ 0\ 0\ 0\ 0\ 0)(I - \mathbf{SP})^{-1}(1\ 1\ 1\ 1\ 1\ 1\ 1\ 0)^{\mathsf{T}} = 285.75 \tag{9.66}$$

Es ist dafür $S = \{0, 1, \ldots, 6\}$

10 Zusammenfassung und Ausblick

In der vorliegenden Arbeit wurde ein neues Fehlertoleranzkonzept vorgestellt, das wir *Regionentreue* nennen. Ziel dieses Konzepts ist es, Systeme zu entwerfen, die, sobald ein Fehler auftritt, jeweils nur einen Teil ihrer Funktionalität einbüßen. Regionentreue zeigt damit starke Parallelen zur *Graceful Degradation*. Wie in Abschnitt 9.1 gezeigt wurde, haben regionentreue Systeme eine Laufzeit, die mindestens proportional zum erreichten Grad der Regionentreue ist.

Das Konzept der Regionentreue kann mit anderen Fehlertoleranzkonzepten, beispielsweise der *Selbststabilisierung*, kombiniert werden, um die Resistenz gegenüber Fehlern zu erhöhen. Insbesondere durch die Einführung redundanter Variablen kann der Grad der Regionentreue direkt vervielfacht werden.

Anhand mehrerer Beispiele wurde vorgeführt, wie Regionentreue eines Systems nachgewiesen werden kann. Bei der Simulation einer der Beispielalgorithmen hat sich gezeigt, dass sich regionentreue Systeme im Mittel wesentlich robuster gegenüber Fehlern erweisen können, als ihr Grad der Regionentreue vermuten lässt. Deswegen

wurde im letzten Abschnitt die Modellierung regionentreuer Systeme durch Markov-Ketten betrachtet. Besonders bei Systemen, die selbststabilisierende Eigenschaften aufweisen, lassen sich so genauere Aussagen über ihr Langzeitverhalten herleiten, insbesondere über die mittlere Güte und die Mean Time To Failure (MTTF) des Systems.

Was in dieser Arbeit nicht betrachtet werden konnte, ist die Kombination regionentreuer Algorithmen. Möglicherweise lassen sich bestehende regionentreue Systeme zu neuen regionentreuen Systemen kombinieren, wie es auch bei selbststabilisierenden Systemen möglich ist (zur Kombination selbststabilisierender Systeme siehe [8]).

Neben der hier vorgestellten Definition von Regionentreue sind auch noch andere Definitionen möglich, wo die untere Grenze für die Güte nicht in gleich großen Schritten herabgesetzt wird. In Abschnitt 3.1 wurde lediglich ein Beispiel dafür gegeben.

Literatur

[1] *SPIN Sourcecode.* Zum Download bei http://spinroot.com/.

[2] Arora, Anish und Mohamed Gouda: *Closure and Convergence: A Foundation of Fault-Tolerant Computing.* IEEE Transactions on Software Engineering, 19:1015–1027, 1993.

[3] Avižienis, Algirdas: *Fault-Tolerant Systems.* IEEE Transactions on Computers, c-25(12), Dezember 1976.

[4] Berghammer, Rudolf: *Ordnungen und Verbände.* In: *Ordnungen, Verbände und Relationen mit Anwendungen*, Seiten 1–26. Springer Fachmedien Wiesbaden, 2012, ISBN 978-3-658-00618-1. http://dx.doi.org/10.1007/978-3-658-00619-8_1.

[5] Clarc, J. und D. Pradham: *Fault Injection: A method for validating computer-system dependability.* IEEE Computer, Juni 1995.

[6] Dijkstra, Edsger W.: *Self-stabilizing systems in spite of distributed control.* Commun. ACM, 17(11):643–644, November 1974, ISSN 0001-0782. http://doi.acm.org/10.1145/361179.361202.

[7] Dijkstra, Edsger W.: *Guarded commands, nondeterminacy and formal derivation of programs.* Commun. ACM, 18(8):453–457, August 1975, ISSN 0001-0782. http://doi.acm.org/10.1145/360933.360975.

[8] Dolev, Shlomi: *Self-stabilization.* MIT Press, Cambridge, MA, USA, 2000, ISBN 0-262-04178-2.

[9] Gupta, Arobinda: *Fault-Containment in Self-Stabilizing Distributed Systems.* Dissertation, University of Iowa, Mai 1997.

[10] Gärtner, F. C.: *Fundamentals of Fault-Tolerant Distributed Computing.* ACM Computing Surveys, 31(1), März 1999.

[11] Hamming, Richard W.: *Error detecting and error correcting codes.* Bell Systems Technical Journal, 29(2):147–160, 1950.

[12] Herlihy, Maurice P. und Jeanette M. Wing: *Specifying Graceful Degradation in Distributed Systems.* In: *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, Seiten 167–177. ACM, August 1987.

[13] Hermanns, Holger: *Interactive Marcov Chains: and the Quest for Quantified Quality*, Band 2428 der Reihe *Lecture Notes in Computer Science.* Springer, 2002.

[14] Holzmann, Gerard: *Spin model checker, the: primer and reference manual.* Addison-Wesley Professional, erste Auflage, 2003, ISBN 0-321-22862-6.

[15] Jalote, Pankaj: *Fault Tolerance in Distributed Systems.* PTR Pentrice Hall, 2002.

[16] Kessels, J. L. W.: *An exercise in proving self-stabilization with a variant function.* Inf. Process. Lett., 29(1):39–42, September 1988, ISSN 0020-0190. `http://dx.doi.org/10.1016/0020-0190(88)90131-7`.

[17] Osaki, Shunji: *Applied stochastik system modeling.* Springer, 1992.

[18] Park, Stephen K. und Keith W. Miller: *Random Number Generators: Good ones are hard to find.* Communications of the ACM, 31(10), Oktober 1988.

[19] R Development Core Team: *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Österreich, 2011. `http://www.R-project.org/`, ISBN 3-900051-07-0.

[20] Shelton, Charles und Philip Koopman: *Using Architectural Properties to Model and Measure Graceful Degradation.* In: Lemos, Rogério, Cristina Gacek und Alexander Romanovsky (Herausgeber): *Architecting Dependable Systems*, Band 2677 der Reihe *Lecture Notes in Computer Science*, Seiten 267–289. Springer, 2003, ISBN 978-3-540-40727-0. `http://dx.doi.org/10.1007/3-540-45177-3_12`.

[21] Soon, T. J.: *QR-Code.* Synthesis Journal, Seiten 60–78, 2008.

[22] Stewart, G. W.: *Matrix Algorithms*, Band 1. SIAM, 1988.

[23] Warns, Timo: *Structural Failure Models for Fault-Tolerant Distributed Computing.* Dissertation, Universität Oldenburg, 2009.

Anhang

A Sourcecode

Listing 6: Promela-Code für den Bubblesort-Algorithmus ohne Replikation (Abschnitt 7)

```
#include "insert1.pml"

#define
int r[ N + 1 ] = 0;
byte state [ N ] = 0;
bool fin = false;
int n_faults;

inline random (x)
{
    int rand = 0;
    x = 256;
    do
    :: x > 0 -> rand = rand + x; x = x / 2
    :: x > 0 -> x = x / 2
    :: x == 0 -> x = rand; break
    od
}

proctype Node (int me)
{
    do
    :: (state[me] == 0) && (r[me] > r[me+1]) -> state[me] = 1
    :: (state[me] == 1) && (state[me-1] != 0) -> state[me] = 0
    :: (state[me] == 1) && (state[me-1] == 0) -> state[me] = 2
    :: (state[me] == 2) && (state[me+1] == 0) -> state[me] = 3
    :: (state[me] == 3) && (r[me] > r[me+1]) -> {
                int tmp, tmp2;
                tmp = r[me];
                tmp2 = r[me+1];
                r[me] = tmp2;
                r[me+1] = tmp;
                state[me] = 0
            }
    :: (state[me] == 3) && (r[me] <= r[me+1]) -> state[me] = 0
    od
}
proctype FirstNode ()
{
    do
    :: (state[0] == 0) && (r[0] > r[1]) -> state[0] = 1
    :: (state[0] == 1) && (state[1] == 0) -> state[0] = 2
    :: (state[0] >= 2) && (r[0] > r[1]) -> {
```

```
                int tmp, tmp2;
45              tmp = r[0];
                tmp2 = r[1];
47              r[0] = tmp2;
                r[1] = tmp;
49              state[0] = 0
            }
51      :: (state[0] >= 2) && (r[0] <= r[1]) -> state[0] = 0
        od
53  }

55  proctype LastNode (int me)
    {
57      do
        :: (state[me] == 0) && (r[me] > r[me+1]) -> state[me] = 1
59      :: (state[me] == 1) && (state[me-1] != 0) -> state[me] = 0
        :: (state[me] == 1) && (state[me-1] == 0) -> state[me] = 2
61      :: (state[me] >= 2) && (r[me] > r[me+1]) -> {
                int tmp;
63              int tmp2;
                tmp = r[me];
65              tmp2 = r[me+1];
                r[me] = tmp2;
67              r[me+1] = tmp;
                state[me] = 0
69          }
        :: (state[me] >= 2) && (r[me] <= r[me+1]) -> state[me] = 0
71      od
    }
73
    proctype FaultClass (int max_faults)
75  {
        int n = 0;
77      int node;
        int v;
79      do
        :: n < max_faults -> random (node);
81                           random (v);
                             r[node % ( N + 1 )] = v % MAX;
83                           n++
        :: n < max_faults -> random (node);
85                           random (v);
                             state[node %  N ] = v % 4;
87                           n++
        od
89  }

91  init
    {
93      int i;
```

```
    int sel;
95  atomic
    {
97      setup ();

99      run FaultClass (n_faults) ;
        run FirstNode () ;
101     for (i : 1 .. ( N - 2 ))
        {
103         run Node (i)
        } ;
105     run LastNode ( N - 1 ) ;
    }
107 }
```

Listing 7: Beispielhafte Ausgabe des Java-Tools. Jede Zeile enthält die relevanten Daten für einen Testfall. Diese werden mit R ausgewertet. (Es sind nur die ersten Zeilen abgedruckt. Insgesamt sind 20 000 Datensätze enthalten.)

```
1  #TESTS     = 2000
2  #PROCESSES = 5
3  #FAULTS    = 0 TO 30
4  #MAX_VALUE = 32
5  nProc   nFault  seed    quality input   output
6  5       0       5365    1.000000        "[19, 23, 23, 4, 17, 11]"↩
           "[4, 11, 17, 19, 23, 23]"
7  5       0       8784    1.000000        "[5, 22, 25, 21, 5, 20]"↩
           "[5, 5, 20, 21, 22, 25]"
8  5       0       3379    1.000000        "[9, 12, 3, 25, 23, 23]"↩
           "[3, 9, 12, 23, 23, 25]"
9  5       0       7223    1.000000        "[15, 14, 14, 27, 16, 7]"↩
           "[7, 14, 14, 15, 16, 27]"
10 5       0       3433    1.000000        "[11, 14, 21, 9, 0, 17]"↩
           "[0, 9, 11, 14, 17, 21]"
```

Listing 8: PROMELA-Code für den Bubblesort-Algorithmus mit TMR (Abschnitt 8)

```
   #include "insert1.pml"
2
   typedef register {
4      int a = 0;
       int b = 0;
6      int c = 0;
   };
8
   #define READ(R)      ( R .a == R .b -> R .a : R .c )
10 #define WRITE(R,X)   atomic { R .a = (X) ; R .b = (X) ; R .c = (X) }
   #define REPAIR(R)    WRITE(R, READ(R))
12 register r[ N + 1 ];
   register state [ N ];
```

```
14 bool fin = false;
   int n_faults;
16
   inline random (x)
18 {
       int rand = 0;
20     x = 256;
       do
22     :: x > 0 -> rand = rand + x; x = x / 2
       :: x > 0 -> x = x / 2
24     :: x == 0 -> x = rand; break
       od
26 }

28 proctype Node (int me)
   {
30     do
       :: atomic{(READ(state[me]) == 0) && (READ(r[me]) > READ(r[me+1])) ↩
           -> REPAIR(state[me]); REPAIR(r[me]); REPAIR(r[me+1])}; WRITE(↩
           state[me], 1)
32     :: atomic{(READ(state[me]) == 1) && (READ(state[me-1]) != 0) -> ↩
           REPAIR(state[me]); REPAIR(state[me-1]);} WRITE(state[me], 0)
       :: atomic{(READ(state[me]) == 1) && (READ(state[me-1]) == 0) -> ↩
           REPAIR(state[me]); REPAIR(state[me-1]);} WRITE(state[me], 2)
34     :: atomic{(READ(state[me]) == 2) && (READ(state[me+1]) == 0) -> ↩
           REPAIR(state[me]); REPAIR(state[me+1]);} WRITE(state[me], 3)
       :: atomic{(READ(state[me]) == 3) && (READ(r[me]) > READ(r[me+1])) ↩
           ->
36                 REPAIR(state[me]); REPAIR(r[me]); REPAIR(r[me+1])}
                   int tmp, tmp2;
38                 tmp = READ(r[me]);
                   tmp2 = READ(r[me+1]);
40                 WRITE(r[me], tmp2);
                   WRITE(r[me+1], tmp);
42                 WRITE(state[me], 0)

44     :: atomic{(READ(state[me]) == 3) && (READ(r[me]) <= READ(r[me+1])) ↩
           -> REPAIR(state[me]); REPAIR(r[me]); REPAIR(r[me+1])}; WRITE(↩
           state[me], 0)
       od
46 }

48 proctype FirstNode ()
   {
50     int me = 0;
       do
52     :: atomic{(READ(state[me]) == 0) && (READ(r[me]) > READ(r[me+1])) ↩
           -> REPAIR(state[me]); REPAIR(r[me]); REPAIR(r[me+1])}; WRITE(↩
           state[me], 1)
```

```
        :: atomic{(READ(state[me]) == 1) && (READ(state[me+1]) == 0) -> ↩
            REPAIR(state[me]); REPAIR(state[me+1]);} WRITE(state[me], 2)
54      :: atomic{(READ(state[me]) >= 2) && (READ(r[me]) > READ(r[me+1])) ↩
            ->
                    REPAIR(state[me]); REPAIR(r[me]); REPAIR(r[me+1])}
56                  int tmp, tmp2;
                    tmp = READ(r[me]);
58                  tmp2 = READ(r[me+1]);
                    WRITE(r[me], tmp2);
60                  WRITE(r[me+1], tmp);
                    WRITE(state[me], 0)
62
        :: atomic{(READ(state[me]) >= 2) && (READ(r[me]) <= READ(r[me+1])) ↩
            -> REPAIR(state[me]); REPAIR(r[me]); REPAIR(r[me+1])}; WRITE(↩
            state[me], 0)
64      od
    }
66
    proctype LastNode (int me)
68  {
        do
70      :: atomic{(READ(state[me]) == 0) && (READ(r[me]) > READ(r[me+1])) ↩
            -> REPAIR(state[me]); REPAIR(r[me]); REPAIR(r[me+1])}; WRITE(↩
            state[me], 1)
        :: atomic{(READ(state[me]) == 1) && (READ(state[me-1]) != 0) -> ↩
            REPAIR(state[me]); REPAIR(state[me-1]);} WRITE(state[me], 0)
72      :: atomic{(READ(state[me]) == 1) && (READ(state[me-1]) == 0) -> ↩
            REPAIR(state[me]); REPAIR(state[me-1]);} WRITE(state[me], 2)
        :: atomic{(READ(state[me]) >= 2) && (READ(r[me]) > READ(r[me+1])) ↩
            ->
74                  REPAIR(state[me]); REPAIR(r[me]); REPAIR(r[me+1])}
                    int tmp, tmp2;
76                  tmp = READ(r[me]);
                    tmp2 = READ(r[me+1]);
78                  WRITE(r[me], tmp2);
                    WRITE(r[me+1], tmp);
80                  WRITE(state[me], 0)

82      :: atomic{(READ(state[me]) >= 2) && (READ(r[me]) <= READ(r[me+1])) ↩
            -> REPAIR(state[me]); REPAIR(r[me]); REPAIR(r[me+1])}; WRITE(↩
            state[me], 0)
        od
84  }

86  proctype FaultClass (int max_faults)
    {
88      int n = 0;
        int node;
90      int v;
        do
```

```
    :: n < max_faults -> random (node);
                         random (v);
                         if
                         :: true -> r[node % ( N + 1 )].a = v % MAX
                         :: true -> r[node % ( N + 1 )].b = v % MAX
                         :: true -> r[node % ( N + 1 )].c = v % MAX
                         fi;
                         n++
    :: n < max_faults -> random (node);
                         random (v);
                         if
                         :: true -> state[node %  N ].a = v % 4
                         :: true -> state[node %  N ].b = v % 4
                         :: true -> state[node %  N ].c = v % 4
                         fi;
                         n++
    od
}

init
{
    int i;
    int sel;
    atomic
    {
        setup ();

        run FaultClass (n_faults) ;
        run FirstNode () ;
        for (i : 1 .. ( N - 2 ))
        {
            run Node (i)
        } ;
        run LastNode ( N - 1 ) ;
    }
}
```

B CD-ROM

Folgende Daten sind auf der beigefügten CD-ROM enthalten:

- `Bachelorarbeit.pdf`: diese Arbeit als PDF
- `bubblesort03.pml`: PROMELA-Code für den Bubblesort-Algorithmus ohne TMR (Abschnitt 7)
- `bubblesort02.pml.test`: Ausgabe des Testtools für den Bubblesort-Algorithmus ohne TMR

- `auswertung.r`: R-Skript zur Auswertung der Testreihen (ohne TMR)
- `bubblesort04.pml`: PROMELA-Code für den Bubblesort-Algorithmus mit TMR (Abschnitt 8)
- `bubblesort04.pml.test`: Ausgabe des Testtools für den Bubblesort-Algorithmus mit TMR
- `auswertung02.r`: R-Skript zur Auswertung der Testreihen (mit TMR)
- `spin_tools`: Das Tool zur automatischen Generierung der Testfälle (als Eclipse-Projekt)

Versicherung

Hiermit versichere ich, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Außerdem versichere ich, dass ich die allgemeinen Prinzipien wissenschaftlicher Arbeit und Veröffentlichung, wie sie in den Leitlinien guter wissenschaftlicher Praxis der Carl von Ossiezky Universität Oldenburg festgelegt sind, befolgt habe.

Jan Steffen Becker

Oldenburg, 12. September 2013