FH Vorarlberg
University of Applied Sciences

Entwicklung einer Aktor-Bibliothek in Rust

Masterarbeit
zur Erlangung des akademischen Grades

Master of Science

Fachhochschule Vorarlberg
Informatik - Software and Information Engineering

Betreut von
Jonathan Thaler, Ph.D

Vorgelegt von
André Hopfgartner, BSc

Dornbirn, 1. Juli 2022

Kurzreferat

Entwicklung einer Aktor-Bibliothek in Rust

Das Ziel dieser Arbeit ist die Analyse und Entwicklung einer Aktor-Bibliothek in Rust. Existierende Aktor-Bibliotheken in Rust verletzen grundlegende Regeln des Aktor-Modells, wodurch diese in nicht allen Anwendungsfällen verwendet werden können. Des Weiteren existiert keine strukturierte Arbeit zu der Entwicklung einer Aktor-Bibliothek in Rust. Die in dieser Arbeit entwickelte Bibliothek entspricht einerseits den Ideen des Aktor-Modells, andererseits wird die Erarbeitung und die durchlaufene Analyse der Bibliothek durch diese Arbeit dokumentiert.

In einem ersten Schritt wird das Aktor-Modell und grundlegende Spracheigenschaften von Rust eingeführt, bevor das Design der entwickelten Aktor-Bibliothek vorgestellt und getroffene Design-Entscheidungen beschrieben und begründet werden.

Darauffolgend werden die Implementationen der relevantesten Komponenten der Bibliothek beschrieben und anhand von Code-Beispielen erklärt. Des Weiteren wird das Design und die Implementation eines Test-Frameworks für Aktoren präsentiert, das das Testen von Aktoren erleichtern soll.

Anhand der Implementation eines Praxisbeispiels wird die entwickelte Aktor-Bibliothek mit Java Akka durch Code-Vergleiche und Benchmarks verglichen. Hierbei wird gezeigt, dass die Verwendung der entwickelten Bibliothek mit einem vergleichbar gleich großen Code-Aufwand zu denselben und mitunter besseren Ergebnissen führen kann. Abschließend werden vorhandene Einschränkungen der entwickelten Bibliothek und deren Auswirkungen beschrieben.

Abstract

Development of an actor library in Rust

The goal of this thesis is the analysis and development of an actor library in Rust. Existing actor libraries in Rust violate fundamental rules of the actor model which is why they cannot be applied to specific use cases. Further no structured work exists which documents the development process of an actor library in Rust. The developed actor library satisfies all rules of the original actor model and the whole development and analysis process is documented in this thesis.

First, the actor model and basic language features of Rust are introduced before the design and relevant design decisions of the developed actor library are shown and illustrated.

Next the implementations of the most relevant components of the library are described and explained using code examples. Furthermore the design and implementation of a test framework for actors is presented.

By implementing a real world example in Java Akka and the developed Rust library, code and performance comparisons are made. One of the described results of the comparisons is that the usage of the developed Rust library leads to a similiar coding effort, whereas the resulting code shows similar and in some cases even better performance properties than the Java Akka implementation. In the last chapter limitations of the developed library and their consequences are described and discussed.

Inhaltsverzeichnis

Abbildungsverzeichnis 7

Abkürzungsverzeichnis 11

1 Einleitung 12

2 Stand des Wissens 15

2.1 Rust 15
2.1.1 Allgemeines 15
2.1.2 Das Ownership-Modell 16
2.1.3 Das Drop-Trait 18
2.1.4 Besitzwechsel durch Moves 19
2.1.5 Das Copy-Trait als Ausnahme von Moves 19
2.1.6 Mutability 20
2.1.7 Das Borrowing-Modell 21
2.1.8 Lifetimes 23
2.1.9 Structs 25
2.1.10 Enums 26
2.1.11 Traits 26
2.1.12 Smart Pointer 29
2.1.13 Generische Datentypen 31
2.1.14 Trait Objects 33
2.1.15 Modelle der Nebenläufigkeit 34
2.1.16 Closures 36
2.1.17 Das Any-Trait 37
2.2 Das Aktor-Modell 38
2.3 Existierende Aktor-Modell-Implementierungen in Rust 40
2.3.1 Actix 40
2.3.2 Acteur 40

2.3.3 Weitere Bibliotheken 41
2.4 Beispiel einer existierenden Aktor-Modell-Implementierung: Erlang . . 42

3 Entwicklung der Bibliothek **44**
3.1 Design 44
3.1.1 Design der Bibliothek 44
3.1.2 Einführungsbeispiel 46
3.2 Design-Entscheidungen 48
3.2.1 Nebenläufigkeit von Aktoren 48
3.2.2 Nachrichtenverarbeitungsmodus 49
3.2.3 Dynamische Typisierung von Nachrichten und Verhalten 50
3.2.4 Warteschlangen-Modell des Postfachs 51
3.2.5 Verarbeitung nicht unterstützter Nachrichten 52
3.3 Implementation 53
3.3.1 Dokumentation 54
3.3.2 Verwendete Rust-Bibliotheken 54
3.3.3 Aktor 55
3.3.4 Aktor-System 66
3.4 Test-Framework 70

4 Praxisbeispiel: Das Schelling-Segregationsmodell mit Aktoren **77**
4.1 Das Schelling-Segregationsmodell 77
4.2 Modell-Design 79
4.3 Implementation 82
4.4 Benchmarks 84
4.5 Zusammenfassung und Diskussion 86

5 Diskussion und Interpretation **87**
5.1 Rust als Sprache 87
5.2 Einschränkungen der Bibliothek 88
5.3 Teilen von Speicher mittels Nachrichten 89
5.4 Teilen von Aktor-Zuständen 90
5.5 Verwendbarkeit der entwickelten Bibliothek 92

6 Zusammenfassung und Ausblick **93**

Literaturverzeichnis **95**

Abbildungsverzeichnis

2.1 Darstellung von Ownership-Beziehungen anhand eines Baums. 17

2.2 Ein `String` wird in einem Scope erzeugt und der Variable hello zugewiesen. Am Ende des Scopes wird der Speicher freigegeben. 18

2.3 Ein neuer `String` wird erzeugt und die Variable owner1 als Owner definiert. Anschließend wird der Besitz an die Variable owner2 übergeben. 19

2.4 Die Werte von a und b werden vor der Übergabe an die Funktion kopiert, weshalb diese nach dem Funktionsaufruf weiterhin verwendet werden können. 20

2.5 Variablen, die nicht explizit als mutable gekennzeichnet sind, können in Rust nicht verändert werden. 21

2.6 Der an die Funktion übergebene Wert `input` muss wieder zurückgegeben werden, da dieser ansonsten am Ende des Scopes der Funktion freigegeben wird. 21

2.7 Die Shared Reference `input` ermöglicht einen lesenden Zugriff auf den referenzierten Wert. 22

2.8 Die Exclusive Reference `input` muss nicht zurückgegeben werden. . . . 23

2.9 Eine Funktion, die eine Shared Reference auf einen Teil eines Strings zurückgibt, ohne den originalen String zu kopieren. Die zurückgegebene Referenz darf maximal so lange verwendet werden, wie der originale String existiert. Dies wird durch die gemeinsame Lifetime `'a` signalisiert. 24

2.10 Jedem Scope wird durch den Compiler eine implizite Lifetime zugewiesen. In diesem Beispiel sind die impliziten Lifetimes durch `'a` und `'b` für das Verständnis gekennzeichnet. In der Praxis sind diese Namen der Lifetimes nicht direkt bekannt, sondern werden durch den Compiler automatisch generiert, damit dieser die Gültigkeit verwendeter Referenzen überprüfen kann. 24

2.11 Beispiel für ein Struct mit verschiedenen assoziierten Funktionen und Methoden. 27

2.12 Enums können für die Modellierung verschiedenster Szenarien verwendet werden. . . . 28
2.13 Das Default-Trait wird verwendet, um für einen Typ einen Standardwert zu definieren. . . . 28
2.14 Durch die Angabe von Supertraits kann ein Trait die Implementation anderer Traits voraussetzen. . . . 29
2.15 Muss ein Typ mehrere Traits implementieren, so kann dies über die Kombination der Traits mittels + ausgedrückt werden. . . . 29
2.16 Smart Pointer können mittels des Dereferenzierungsoperators * wie normale Referenzen dereferenziert werden. . . . 30
2.17 Der Option-Typ ist ein generisches Enum. . . . 31
2.18 Durch die Angabe von Trait Bounds in generischen Datentypen können die Funktionen der angegeben Traits verwendet werden, ohne deren tatsächliche Implementierung kennen zu müssen. . . . 32
2.19 In der Liste werden Trait Objects des Traits SayHello gespeichert. Als Referenztyp wird Box<T> verwendet. . . . 34
2.20 Die Art jeder Closure wird vom Compiler automatisch, basierend auf der Verwendung der Umgebung, bestimmt. . . . 37
2.21 Durch die Verwendung des Any-Traits stehen zur Laufzeit Metainformationen über die Datentypen von Trait Objects zur Verfügung. Trait Objects können dadurch zu ihrem tatsächlichen Typ zurückkonvertiert werden. . . . 38

3.1 Beispiel zweier Aktoren, die abwechselnd eine Ping- und Pong-Nachricht aneinander schicken. . . . 47
3.2 Durch die Verwendung der Thiserror-Attribute wird Error, Display und Debug automatisch implementiert. . . . 55
3.3 Manuelle Implementation des in Abb. 3.2 dargestellten Fehlertyps. . . . 56
3.4 Der Datentyp ExampleState repräsentiert den Zustand des erzeugten Aktors, wobei der Wert init_state dem initialen Zustand des Aktors entspricht. . . . 57
3.5 Der Typ Message besitzt die zu übertragende Nachricht in inner und eine optionale Antwortadresse. pub(crate) bedeutet hierbei, dass auf dieses Feld nur von innerhalb der Bibliothek zugegriffen werden kann. . . . 60

3.6 Über die Adresse eines Aktors können Tell- und Ask-Nachrichten an diesen gesendet werden. Bei einer Ask-Nachricht muss zusätzlich eine Antwort-Adresse übergeben werden. 62
3.7 Während der Verarbeitung einer Ask-Nachricht hat ein Aktor Zugriff auf die empfangene Nachricht, seinen Zustand, die Absenderadresse und seinen Aktor-Kontext. 63
3.8 Das Verhalten eines Aktors kann sich am Ende jeder Nachrichtenverarbeitung ändern. 64
3.9 Durch den Aktor-Kontext kann ein Aktor während der Verarbeitung einer Nachricht neue Aktoren mit dem Aktor-System assoziieren. . . . 64
3.10 Ein Verhalten kann nur über den entsprechenden Builder erzeugt werden. 65
3.11 Durch die Erzeugung eines Aktor-Systems können Aktoren ausgeführt werden. Weil die Ausführung von Aktoren intern auf der Verwendung von Futures basiert, muss der Aufruf der `start()`-Methode des Aktor-Systems mit dem `await`-Schlüsselwort ausgeführt werden. 67
3.12 Aktoren können ihren Aktor-Kontext verwenden, um Zugriff auf Funktionen des Aktor-Systems zu erlangen. In diesem Beispiel wird die Adresse eines Aktors über dessen Name bestimmt. 68
3.13 Ein Aktor kann über seinen Kontext Nachrichten an alle sich in einem Aktor-System befindlichen Aktoren senden. 69
3.14 Broadcast-Nachrichten können direkt über das Aktor-System versendet werden. 69
3.15 Durch die Implementation des `SupervisionStrategy<S>`-Traits können neue Überwachungsstrategien definiert werden. 70
3.16 Jeder Aktor kann mit einer Überwachungsstrategie im Aktor-System erfasst werden. 71
3.17 Ein Beispiel für das Testen eines einfachen Aktors mittels der integrierten Test-Funktionalität. 72
3.18 Durch den Aufruf von `enable_state_checks()` werden die für das Testen benötigten Protokolle aktiviert und automatisch implementiert. 73
3.19 Mittels der `check`-Methode können beliebige lesende Abfragen auf den Zustand eines zu testenden Aktors durchgeführt werden. 73
3.20 Ein Beispiel für das Testen eines einfachen Aktors mit erwarteten Antwortnachrichten und Zustandsüberprüfungen. 74
3.21 Die definierten Testaktionen werden strikt sequenziell abgearbeitet und überprüft. 75

4.1 50×50-Gitter mit 2.000 zufällig platzierten Personen. Rote und grüne Punkte entsprechen den Personen und ihren Personengruppen, weiße Punkte leeren Feldern. 78
4.2 Nach 100 Simulationsschritten, einem minimalen Gruppenanteil $B_{\text{min}} = 0.4$ und einer Nachbarschaftsgröße von einem Feld sind die Gruppen bereits sichtlich separiert. 79
4.3 Jeder Person-Aktor erhält eine zufällige Startposition von dem Grid-Aktor und informiert anschließend den Simulation-Aktor darüber. . . . 80
4.4 Person-Aktoren, die mit ihrer Umgebung zufrieden sind, wechseln den Standort nicht. 81
4.5 Wenn $B < B_{\text{min}}$, dann retourniert der Person-Aktor seine derzeitige Position und erhält eine neue, zufällige Position. 82
4.6 In Java Akka werden Verarbeitungsaktionen von Aktoren als Methoden der Aktor-Klasse implementiert. 83
4.7 In der entwickelten Rust-Bibliothek werden Verarbeitungsaktionen über das Verhalten definiert. 83
4.8 In Java Akka wird der Zustand eines Person-Aktors durch die Attribute der entsprechenden Aktor-Klasse repräsentiert. 84
4.9 In der entwickelten Rust-Bibliothek wird der Zustand von Person-Aktoren durch ein Struct abgebildet. 84

5.1 Über Smart Pointer kann Speicher durch Nachrichten geteilt werden. . . 90
5.2 Über Smart Pointer kann der Zustand von Aktoren geteilt werden. . . 91

Abkürzungsverzeichnis

RAII Resource Acquisition Is Initialization

ADT Algebraic Data Type

DST Dynamically Sized Type

MPSC Multi-Producer, Single-Consumer

1 Einleitung

Das Aktor-Modell ist ein erstmals in Hewitt, Bishop und Steiger (1973) beschriebenes Berechnungsmodell, dessen grundlegende Berechnungseinheiten sogenannte Aktoren darstellen. Jeder Aktor besitzt einen Zustand, ein Postfach für eingehende Nachrichten und ein Verhalten, das die Verarbeitung verschiedener Nachrichtenarten definiert. Aktoren kommunizieren untereinander ausschließlich über Nachrichten und teilen sich keinen gemeinsamen Speicher. Das Postfach jedes Aktors entspricht einer FIFO-Datenstruktur, von der jeweils genau eine Nachricht entfernt und verarbeitet wird. Sinngemäß kann jeder Aktor zu einem bestimmten Zeitpunkt höchstens eine Nachricht verarbeiten, mehrere Aktoren können aber parallel ihre jeweiligen Nachrichten abarbeiten.

Diese grundlegende Idee des Aktor-Modells, welches in Abschnitt 2.2 detailliert beschrieben wird, ermöglicht die Entwicklung sehr gut skalierender, höchst paralleler Systeme, was auch der Grund für die derzeit stark wachsende Beliebtheit dieses Modells ist.

Als ein beispielhaftes Erfolgsprojekt des Aktor-Modells, konkret der Programmiersprache Erlang, deren Implementation grundsätzlich dem Aktor-Modell entspricht, ist der Ericsson AXD301, ein High Performance ATM Switch, zu nennen. In Armstrong (2003, S. 191) wird beschrieben, dass ein Kunde von Ericsson mit einem Cluster von 14-Nodes in einem Zeitraum von acht Monaten eine Verfügbarkeit von 99.999999999% des Systems erreicht haben soll. Diese beschriebene Verfügbarkeit ist mit hoher Wahrscheinlichkeit nicht ausschließlich auf das Aktor-Modell zurückzuführen, sondern könnte auch das Ergebnis einer allgemein guten Software-Entwicklung sein. Was dieses Beispiel allerdings zeigt ist, dass sich dieses Modell in hoch skalierten, parallelen und nebenläufigen Architekturen auch bereits in der Praxis bewähren konnte.

Die Implementation von Aktor-Modellen ist grundsätzlich in jeder Programmiersprache möglich, allerdings eignen sich manche Sprachen aufgrund ihrer Eigenheiten besser

als andere. Eine Systemprogrammiersprache, die in der letzten Zeit immer mehr Aufmerksamkeit bekommen hat, ist Rust. Rust ist eine von Mozilla Research entwickelte Programmiersprache, die erstmals 2010 mit dem Ziel veröffentlicht wurde, möglichst performanten aber auch sicheren Code zu unterstützen. Die Sprache gilt als direkte Konkurrenz zu C++ und wird bereits in verschiedensten Bereichen wie beispielsweise der Embedded-Entwicklung und für Server-Backends verwendet. In Benchmark-Tests erreichen Rust und C++ vergleichbare Ergebnisse mit leichter Variation, je nach Benchmark-Typ.

Das Ziel dieser Arbeit ist die Entwicklung und Analyse einer Aktor-Modell-Bibliothek in Rust, die auf einem lokalen System ausgeführt werden kann. Besonderes Augenmerk wird hierbei auf die für Rust spezifischen Entwicklungsdetails gelegt, die im Rahmen dieser Arbeit näher beschrieben werden. Rust wurde hierfür gewählt, da zwar bereits Bibliotheken mit diesem Ziel existieren, bisher aber keine der existierenden Lösungen alle Grundprinzipien des Aktor-Modells unterstützen beziehungsweise erfüllen, es noch keine strukturierte Arbeit und Analyse zur Entwicklung einer solchen Bibliothek gibt und die existierenden Lösungen für die Verwendung der Bibliothek sehr spezielles, vertieftes Wissen über Rust voraussetzen. Des Weiteren bietet Rust einzigartige Sprach-Features, die in dieser Arbeit vorgestellt und deren Implikationen im Rahmen der Entwicklung der Bibliothek genauer beschrieben und analysiert werden. Zusätzlich zu der entwickelten Aktor-Bibliothek wird ein Test-Framework vorgestellt, das das Testen von Aktoren drastisch erleichtert und sich in das in Rust existierende Test-Framework direkt integriert.

Als Arbeitshypothese wird angenommen, dass sich Rust als Programmiersprache für die Entwicklung einer Aktor-Bibliothek sehr gut eignet und die speziellen Eigenschaften der Sprache einen positiven Beitrag zu der Entwicklung und Funktionsweise einer Aktor-Bibliothek beitragen. Diese Hypothese wird im Rahmen der in Abschnitt 3.2 geführten Diskussionen über mögliche Implementationen der Komponenten einer Aktor-Bibliothek und der Implementation der Bibliothek selbst verifiziert. Es wird gezeigt, dass Rust als Programmiersprache alle für eine Aktor-Bibliothek benötigten Schnittstellen und Funktionen aufweist und die speziellen Sprachfeatures, wie beispielsweise das Ownership-Modell, das in Abschnitt 2.1.2 beschrieben wird, die Entwicklung einer Aktor-Bibliothek ermöglichen respektive erleichtern.

Des Weiteren wird anhand der Implementation des Schelling Segregationsmodells, das in Abschnitt 4.1 beschrieben wird, exemplarisch gezeigt, dass die entwickelte Bibliothek

für die praktische Anwendung geeignet ist. Es wird der Implementationsaufwand des Schelling Segregationsmodells in Java Akka und Rust verglichen, wobei gezeigt wird, dass in beiden Sprachen respektive Bibliotheken ein ähnlicher Implementationsaufwand entsteht. Anhand von Benchmarks der konzeptionell identischen Implementationen wird ermittelt, dass die entwickelte Rust-Bibliothek für kleinere Simulationen dieses Modells eine signifikant bessere Performance als die Java-Implementation aufweist.

Die Ergebnisse der Arbeit werden anschließend diskutiert und vorhandene Einschränkungen und deren potenzielle Ursachen erläutert. Abschließend werden die Ergebnisse der Arbeit zusammengefasst und mögliche Erweiterungen der Bibliothek präsentiert und diskutiert.

2 Stand des Wissens

In diesem Kapitel wird die Programmiersprache Rust und ausgewählte Eigenschaften derer beschrieben. Weiters werden die Grundprinzipien des Aktor-Modells erläutert und anschließend zwei existierende Implementationen dieses Modells in Rust betrachtet. Abschließend wird die Programmiersprache Erlang als Beispiel für eine lang erprobte Implementation des Aktor-Modells vorgestellt.

2.1 Rust

In diesem Kapitel werden ausgewählte Konzepte der Programmiersprache Rust beschrieben. Die Auswahl der beschriebenen Konzepte wurde so getroffen, dass das Wissen für das Verständnis der in späteren Kapiteln beschriebenen Designentscheidungen aufgebaut wird und Grundideen der Programmiersprache Rust vermittelt werden. Alle in den folgenden Kapiteln beschriebenen Konzepte und Ideen entsprechen den in *The Rust Reference* (2022) dokumentierten, zum Zeitpunkt der Erstellung dieser Arbeit gültigen, Konzepten und Paradigmen und können sich in zukünftigen Versionen der Sprache ändern.

2.1.1 Allgemeines

Rust ist eine erstmals 2010 von Mozilla Research veröffentlichte, stark typisierte Open Source Systemprogrammiersprache ohne Garbage Collection und Runtime. Rust ermöglicht einerseits den für Systemprogrammiersprachen typischen direkten Zugriff auf Speicher, wodurch Rust auch für die Entwicklung von Embedded-Systemen und Betriebssystemen geeignet ist. Andererseits verfügt Rust über Sprach-Features, die das Arbeiten mit Speicher sicher gestalten und zusätzliche Abstraktionen ermöglichen, wodurch in Rust entwickelter Code im Allgemeinen, im Vergleich zu klassischen Systemprogrammiersprachen wie beispielsweise C, sehr gut lesbar ist, da Abstraktionen

wie Iteratoren und Closures verwendet werden können. Weil Rust sowohl viele hilfreiche Abstraktionen, als auch eine sehr gute Performance ermöglicht, ist das Interesse an dieser Programmiersprache in den letzten Jahren stark gewachsen. Repräsentativ hierfür seien die Ergebnisse der Stack Overflow Surveys 2016 bis 2021 zu nennen, in denen Rust in allen Jahren den ersten Platz in der Kategorie „Most loved programming language“ belegen konnte.

2.1.2 Das Ownership-Modell

Eines der markantesten Alleinstellungsmerkmale von Rust ist das sogenannte Ownership-Modell. Jede Programmiersprache verfolgt einen eigenen Ansatz für die Verwaltung von verwendetem Speicher auf dem Heap. Einige Sprachen verwenden Garbage Collection, um nicht mehr benötigten Speicher ohne Zutun des Programmierenden automatisch freizugeben. Ein Beispiel für das andere Extrem, nämlich die manuelle Speicherverwaltung, ist die Programmiersprache C, in der jeder manuell allokierte Speicher auch manuell durch den Programmierenden wieder freizugeben ist. Eine manuelle Speicherverwaltung wird vor allem für Sprachen verwendet, deren Code in ressourcenbeschränkten Umgebungen funktionieren muss oder deren Code sehr performant sein muss, da bei dieser Art der Speicherverwaltung keine zusätzliche Laufzeit für etwaige Garbage-Collection-Algorithmen aufgebracht werden muss.

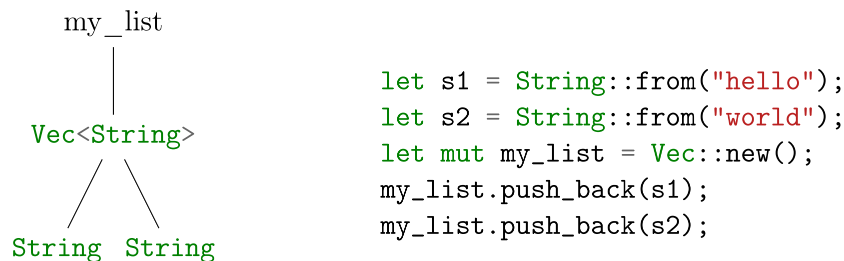
Der manuelle Speicherverwaltungsansatz übergibt dem Entwickelnden die Verantwortung für die korrekte Speicherverwaltung, wobei Compiler einfache Fehler in der Speicherverwendung detektieren können. Oft stellt sich die Frage, an welcher Stelle des Codes und zu welchem Zeitpunkt Speicher zu allokieren und freizugeben ist. Ein in diesem Kontext häufig angewandtes Prinzip ist Resource Acquisition Is Initialization (RAII). Bei RAII obliegt dem Teil des Codes die Speicherverwaltung, der den jeweiligen Speicher allokiert hat. In Rust ist der für die Speicherverwaltung zuständige Teil eines Speichers immer eine Variable, die Owner genannt wird. Wird der Speicher des Owners freigegeben, so muss dieser dafür sorgen, dass der gesamte Speicher in dessen Besitz auch freigegeben wird. In C++ wird RAII seit längerem verwendet, wobei dies nur als Code-Empfehlung gehandhabt und nicht durch den Compiler sichergestellt wird. Eine Verletzung dieser Empfehlung kann zu nicht freigegebenem oder mehrfach freigegebenem Speicher führen, was beides einen Fehler darstellen würde.

In Rust wurde das Prinzip von RAII unter dem Namen Ownership-Modell vollständig

in die Sprache integriert. Jeder Code in Rust muss diesem Prinzip folgen, wobei der Compiler die Einhaltung dessen prüft und Code, der diese Regeln verletzt, ablehnt. In Rust gelten folgende Regeln im Zusammenhang mit dem Ownership-Modell:

- Jeder Wert in Rust gehört einer Variable, die Owner genannt wird.
- Für jeden Wert gibt es zu einem bestimmten Zeitpunkt genau einen Owner.
- Erreicht ein Owner das Ende eines Scopes, so wird dessen Speicher freigegeben.

Durch diese sehr strikten Einschränkungen kann die Speicherzuständigkeit jedes Programms in Rust durch einen Baum dargestellt werden, in dem jeder Knoten einem Owner und jedes Blatt einem Wert entspricht. In Abb. 2.1 ist ein solcher Baum anhand des Beispiels Vec<String> dargestellt. Dieser Datentyp speichert eine Liste von String's und besitzt die gespeicherten Werte, ist also deren Owner. Die Liste wird wiederum von einer Variable mit dem Namen my_list besessen.



(a) Der Ownership-Baum des in Abb. 2.1b dargestellten Codes.

(b) Erzeugen eines Vec<String> mit zwei String's.

Abbildung 2.1: Darstellung von Ownership-Beziehungen anhand eines Baums.

Die Freigabe des Speichers eines Owners entspricht der Freigabe des Speichers des Owners und sämtlichen Elementen unter diesem Knoten im Ownership-Baum. Details zur Freigabe von Speicher in Rust werden in Abschnitt 2.1.3 beschrieben. In dem in Abb. 2.1b dargestellten Code-Beispiel wurde ein Detail noch nicht geklärt, nämlich wird zum Beispiel in der Zeile `let s1 = String::from("hello");` ein neuer String erzeugt und die Variable s1 als Owner definiert. In Abb. 2.1a ist der Owner beider String's allerdings als Vec<String> deklariert. Da beide Aussagen richtig sind, und, gemäß den zuvor definierten Regeln, zu jedem Zeitpunkt genau ein Owner existieren kann, muss sich der Owner des String's geändert haben. Konkret geschieht dies im

oben genannten Beispiel in der Zeile `my_list.push_back(s1);`. Eine solche Operation, nämlich der Besitzwechsel, wird in Rust Move genannt. Dieses Konzept wird in Abschnitt 2.1.4 näher beschrieben und ist essenziell für das Funktionieren des Ownership-Modells.

2.1.3 Das Drop-Trait

Bevor im nächsten Kapitel Moves beschrieben werden, muss geklärt werden, wie in Rust Speicher freigegeben wird. Allgemein gilt, dass am Ende jedes Scopes sämtlicher Speicher des Scopes freigegeben wird. Ein Scope entspricht hierbei der in der Softwareentwicklung üblichen Definition von Scopes, die in Rust, wie in vielen anderen Sprachen, durch geschwungene Klammern dargestellt werden. In Rust wird an jeder Stelle, an der eine geschwungene Klammer verwendet wird, auch ein neues Scope erstellt. Beispiele für die Erzeugung von Scopes sind if-Statements, match-Statements und Funktionsdefinitionen. Jeder Wert in Rust besitzt automatisch eine `drop()`-Methode, die über das `Drop`-Trait definiert wird und vergleichbar mit Destruktoren in anderen Sprachen ist. Die `drop()`-Methode wird auf jedem Wert, dessen Besitz sich in einer Variable befindet, die sich am Ende eines Scopes in diesem befindet, aufgerufen. Der Speicher wird dadurch freigegeben und kann zu einem späteren Zeitpunkt nicht mehr verwendet werden. Ein Beispiel für dieses Prinzip ist in Abb. 2.2 dargestellt. Der Drop-Mechanismus garantiert, dass jeder allokierte Speicher bis spätestens zur Terminierung des Programms freigegeben wird.

```rust
{
    // Start eines neuen Scopes

    let hello = String::from("hello");
    // Ende des Scopes. hello wird an dieser Stelle freigegeben
}
// hello kann hier nicht mehr verwendet werden
```

Abbildung 2.2: Ein `String` wird in einem Scope erzeugt und der Variable hello zugewiesen. Am Ende des Scopes wird der Speicher freigegeben.

2.1.4 Besitzwechsel durch Moves

Eine Implikation der in Abschnitt 2.1.2 beschriebenen Regeln des Ownership-Modells ist, dass sich der Owner eines Wertes ändern kann. Diese Operation erfüllt alle zuvor beschriebenen Regeln und ist essenziell für das Funktionieren des Ownership-Modells, da dieses ansonsten nicht flexibel genug wäre, um funktionierenden Code entwickeln zu können. Im Rahmen von Rust wird ein Besitzwechsel Move genannt. Moves haben einen sehr geringen rechnerischen Aufwand, da lediglich die besitzenden Pointer verändert werden. Ein Move entspricht somit dem Kopieren eines Pointers. Bildlich veranschaulicht gleicht ein Move dem Abschneiden eines Knotens inklusive den Elementen unter dieses Knotens im Ownership-Baum und dem Einfügen dieses Teilbaums an einer anderen Stelle im Baum.

```
let owner1 = String::from("hello");
let owner2 = owner1;
// owner1 ist an dieser Stelle nicht mehr gültig, da der Besitz des
// Strings auf owner2 übertragen wurde
```

Abbildung 2.3: Ein neuer `String` wird erzeugt und die Variable owner1 als Owner definiert. Anschließend wird der Besitz an die Variable owner2 übergeben.

Grundsätzlich ist jede Weitergabe eines Wertes in Rust ein Move. Für Funktionen bedeutet dies, dass jeder Parameter, der übergeben wird und zukünftig weiterhin benötigt wird, über den Rückgabewert der Funktion zurückgegeben werden muss, da bei der Übergabe der Besitz der Werte an die Funktion abgegeben wird. Werden die Werte nicht zurückgegeben, so werden diese am Ende des Scopes der Funktion, wie in Abschnitt 2.1.3 beschrieben, freigegeben und können nicht mehr verwendet werden. Da dieser Ansatz für mehrere Parameter nicht praktikabel wäre, wurden in Rust zwei weitere Prinzipien integriert, nämlich Typen, die den Copy-Trait implementieren, und das Borrowing-Modell. Diese zwei Prinzipien werden in den folgenden Kapiteln genauer beschrieben und erleichtern die Arbeit mit dem Ownership-Modell enorm.

2.1.5 Das Copy-Trait als Ausnahme von Moves

Um für speichertechnisch kleine Datentypen wie beispielsweise 32-Bit-Integer den zuvor beschriebenen Aufwand von Moves, nämlich die Übergabe und Rückgabe der

Parameter, zu minimieren, wurde in Rust das sogenannte Copy-Trait eingeführt. Trait's werden in Abschnitt 2.1.11 genauer beschrieben, an dieser Stelle sei aber bereits vorweggenommen, dass Traits verwendet werden, um Eigenschaften von Typen zu beschreiben und vergleichbar mit Typklassen in Haskell und Interfaces in Java sind. Im Fall des Copy-Traits wird beschrieben, dass der entsprechende Wert kopiert werden kann. Alle Werte, deren Typen diesen Trait implementieren, werden in Rust nicht direkt über Moves übergeben, sondern werden zuerst kopiert und die Kopie wird über einen Move übergeben, wobei das Original den Owner nicht wechselt. Diese Operation verletzt keine der zuvor beschriebenen Regeln und vereinfacht die Arbeit mit primitiven Datentypen in Rust, da der Ausgangswert weiterhin verwendet werden kann. Ein Beispiel für dieses Prinzip ist in Abb. 2.4 dargestellt. Um den Kopieraufwand minimal zu halten, wird das Copy-Trait per Konvention ausschließlich für speichertechnisch kleine Datentypen wie beispielsweise 32-Bit Integer und Enums implementiert. Für größere Datentypen sei auf das Borrowing-Modell, das in Abschnitt 2.1.7 beschrieben wird, verwiesen.

```rust
fn add(a: u32, b: u32) -> u32 {
    a + b
}

let a = 3;
let b = 4;
let sum = add(a, b);
// a und b können weiterhin verwendet werden, da nur Kopien
// von a und b übergeben wurden
```

Abbildung 2.4: Die Werte von a und b werden vor der Übergabe an die Funktion kopiert, weshalb diese nach dem Funktionsaufruf weiterhin verwendet werden können.

2.1.6 Mutability

Jede Variable in Rust ist standardmäßig immutable, was bedeutet, dass deren Wert nicht verändert werden kann. Da in der Regel einige Variablen in Programmen verändert werden müssen, wurde in Rust das mut-Schlüsselwort eingeführt. Variablen und Referenzen, die mit einem mut-Schlüsselwort gekennzeichnet sind, können den Wert beziehungsweise im Fall einer Referenz den referenzierten Wert verändern. Die explizite Angabe von Schreib- und Leseberechtigungen ermöglicht die Implementation

eines feingranularen Berechtigungskonzepts, wodurch Schnittstellen klarer und sicherer gestaltet werden können. Ein Beispiel für die Verwendung des `mut`-Schlüsselworts ist in Abb. 2.5 dargestellt.

```
let a = 3;
// Diese Operationen führt zu einem Fehler, da a immutable ist.
a += 2;

let mut b = 4;
// Diese Operation ist gültig, da b als mutable gekennzeichnet wurde.
b += 2;
```

Abbildung 2.5: Variablen, die nicht explizit als mutable gekennzeichnet sind, können in Rust nicht verändert werden.

2.1.7 Das Borrowing-Modell

Theoretisch wäre das Prinzip von Moves für die Funktion von Rust als Programmiersprache ausreichend, es müsste aber jeder Wert, der an eine Funktion übergeben wird, auch wieder zurückgegeben werden, damit dieser nicht am Ende des Scopes der Funktion, wie in Abschnitt 2.1.3 beschrieben, freigegeben wird. Für mehrere Rückgabewerte könnte beispielsweise die Rückgabe über Tupel realisiert werden, allerdings wäre der dadurch entstehende Code sehr schlecht les- und wartbar. Ein Beispiel für eine Über- und Rückgabe des Besitzes dieser Art ist in Abb. 2.6 dargestellt.

```
fn append_hello(mut input: String) -> String {
    input.push_str("hello");
    input
}
```

Abbildung 2.6: Der an die Funktion übergebene Wert `input` muss wieder zurückgegeben werden, da dieser ansonsten am Ende des Scopes der Funktion freigegeben wird.

Da die Entwicklung gemäß diesem Schema sehr einschränkend und vor allem für mehrere Funktionsparameter nicht praktikabel wäre, wurde in Rust das Borrowing-Modell, das mit dem Ownership-Modell sehr stark gekoppelt ist, implementiert. Das

Borrowing-Modell besagt, dass, zusätzlich zum Besitz eines Wertes, zwei Arten von Referenzen auf Werte in Rust existieren können:

- Shared References: Ermöglichen einen Lesezugriff auf den referenzierten Wert.
- Exclusive References: Ermöglichen Lese- und Schreibzugriff auf den referenzierten Wert.

Ein Beispiel für die Verwendung einer Shared Reference ist in Abb. 2.7 dargestellt.

```
fn read_string(input: &String) -> () {
    println!("{}", input);
}
```

Abbildung 2.7: Die Shared Reference `input` ermöglicht einen lesenden Zugriff auf den referenzierten Wert.

Referenzen in Rust sind vergleichbar mit Pointern in anderen Sprachen, weisen aber zusätzlich folgende Eigenschaften auf:

- Referenzen sind immer gültig, das heißt, dass eine Referenz nie auf `null` zeigt und auch nur so lange wie der referenzierte Wert existiert.
- Es kann höchstens eine Exclusive Reference oder beliebig viele Shared References auf einen bestimmten Wert existieren.

Die erste Regel garantiert, dass in Rust nie auf einen ungültigen Speicher zugegriffen werden kann, weil der Compiler die Einhaltung der Regeln prüft und Programme, die einen ungültigen Zugriff verursachen würden, nicht akzeptiert. Wie der Compiler überprüft, ob eine Referenz gültig ist, wird in Abschnitt 2.1.8 beschrieben.

Die zweite Regel stellt sicher, dass keine Race Conditions in Multithreading-Umgebungen auftreten können. Wenn eine Exclusive Reference auf einen Wert existiert, kann diese den Inhalt beliebig ändern und lesen, da garantiert ist, dass keine andere Referenz zu diesem Wert existiert und diesen lesen oder ändern könnte. Sind ausschließlich Shared References auf einen Wert vorhanden, so können alle Shared References gleichzeitig den referenzierten Wert lesen, da garantiert ist, dass der Inhalt nicht verändert werden kann. Referenzen können mittels des Dereferenzierungsoperators $*$ dereferenziert werden, um den referenzierten Wert hinter der Referenz zu erhalten.

Das Borrowing-Modell hat keinen Einfluss auf den Besitz eines Werts, da eine Referenz einen Wert nur von dessen Owner ausleiht - daher auch der Name Borrowing-Modell. Wird eine Referenz am Ende eines Scopes freigegeben, so hat dies keinerlei Auswirkungen auf den referenzierten Wert, da nur die Referenz auf diesen freigegeben wird.

Das in Abb. 2.6 dargestellte Beispiel ist in Abb. 2.8 mit einer Exclusive Reference dargestellt. Bei der Übergabe einer Referenz muss diese nicht zurückgegeben werden, da die Änderung am tatsächlichen Wert über die Referenz durchgeführt wird und diese am Ende des Scopes der Funktion ohne Einfluss auf den tatsächlichen Wert freigegeben wird.

```
fn append_hello(input: &mut String) -> () {
    input.push_str("hello");
}
```

Abbildung 2.8: Die Exclusive Reference `input` muss nicht zurückgegeben werden.

2.1.8 Lifetimes

Die in Abschnitt 2.1.7 beschriebene Eigenschaft von Referenzen, dass diese immer gültig sind, erfordert, dass der Compiler ein Verständnis für die Gültigkeitsdauer von Referenzen besitzt. Das Grundproblem ergibt sich aufgrund der Frage, wie lange eine Referenz respektive der referenzierte Speicher gültig ist beziehungsweise sein muss, damit ausschließlich auf gültigen Speicher zugegriffen wird. Dieses grundlegende Problem tritt auch in anderen Sprachen, in denen Pointer verwendet werden, auf. Die Verantwortung für die richtige Handhabung obliegt hierbei dem Entwickelnden. Im Gegensatz hierzu überprüft in Rust der Compiler die richtige Verwendung von Referenzen. Diese Überprüfung ist dank sogenannter Lifetimes möglich. Jede Referenz in Rust besitzt eine Lifetime, die entweder explizit angegeben wird, oder die der Compiler, sofern möglich, automatisch ermittelt. Lifetimes beginnen immer mit einem Apostroph, gefolgt von einem Namen in Kleinbuchstaben, wobei häufig sehr kurze Namen wie `'a` für die Bezeichnung von Lifetimes verwendet werden. Der Compiler kann, basierend auf den implizit und explizit definierten Lifetimes, sicherstellen, dass keine Referenz länger als der referenzierte Wert existieren darf, indem die tatsächliche

Lifetime mit der benötigten Lifetime verglichen wird. Ein Beispiel für die Verwendung von Lifetimes innerhalb einer Funktion ist in Abb. 2.9 dargestellt.

```
fn substring(input: &'a str) -> &'a str{
    // Implementation der Funktion
}
```

Abbildung 2.9: Eine Funktion, die eine Shared Reference auf einen Teil eines Strings zurückgibt, ohne den originalen String zu kopieren. Die zurückgegebene Referenz darf maximal so lange verwendet werden, wie der originale String existiert. Dies wird durch die gemeinsame Lifetime `'a` signalisiert.

```
// Dieses Scope hat die Lifetime 'a
{
    // Diese Referenz hat die Lifetime 'a
    let ref1;
    // Dieses Scope hat die Lifetime 'b
    {
        let var = String::from("hello world!");
        // Diese Referenz hat die Lifetime 'b
        ref1 = &var;
    }
    // Von ref1 referenzierter Speicher muss mindestens für die
    // Lifetime 'a gültig sein. 'b erfüllt dieses Kriterium
    // nicht, weshalb dieser Code nicht compiliert.
    println!("{}", ref1);
}
```

Abbildung 2.10: Jedem Scope wird durch den Compiler eine implizite Lifetime zugewiesen. In diesem Beispiel sind die impliziten Lifetimes durch `'a` und `'b` für das Verständnis gekennzeichnet. In der Praxis sind diese Namen der Lifetimes nicht direkt bekannt, sondern werden durch den Compiler automatisch generiert, damit dieser die Gültigkeit verwendeter Referenzen überprüfen kann.

Weil jede Referenz in Rust eine implizite oder explizite Lifetime besitzt, können Lifetimes an allen Stellen, an denen Referenzen verwendet werden können, vorkommen. In Abb. 2.10 ist die Funktionsweise von impliziten Lifetimes in Kombination mit Scopes veranschaulicht. Abhängig von dem jeweiligen Use Case müssen zusätzliche Regeln beachtet werden, die an dieser Stelle nicht genauer beschrieben werden und in

The Rust Reference (2022) dokumentiert sind. Durch die Unterstützung des Compilers bei der Verwendung von Referenzen ist garantiert, dass in jedem Programm, das vom Compiler akzeptiert wurde, Referenzen richtig verwendet werden und keine ungültigen Speicherzugriffe auftreten können.

2.1.9 Structs

Eine der Hauptdatenstrukturen von Rust ist das Struct. Das Struct ist ein Datencontainer, der beliebige Werte und Referenzen zu Werten speichern kann. Grundsätzlich entspricht die Definition des Structs in Rust der des Structs in C, wobei ein Struct in Rust zusätzlich assoziierte Funktionen, Methoden, Konstanten und Typen besitzen kann. Assoziierte Funktionen sind Funktionen, die an den Typ eines Structs gebunden sind. In objektorientierten Sprachen entsprechen assoziierte Funktionen den Klassenmethoden. Das Analogon zu den Objektmethoden in objektorientierten Sprachen sind Methoden in Rust, die zu dem Wert eines Structs gehören. Diese grenzen sich zu den assoziierten Funktionen ab, indem sie als ersten Parameter `self`, `mut self`, `&self` oder `&mut self` besitzen. Die Unterschiede zwischen den vier Arten von Methoden ergeben sich durch die zuvor definierten Regeln des Ownership-Modells und den zugehörigen Prinzipien:

- `self` bedeutet, dass der Wert des Structs in die Funktion mittels eines Moves übergeben wird. Das Struct kann nicht verändert werden, da `self` immutable ist.

- `mut self` entspricht der Übergabe von `self` in die Funktion. Das Struct kann aufgrund des `mut`-Schlüsselwortes verändert werden.

- `&self` entspricht der Übergabe einer Shared Reference zum Wert des Structs. Die Werte des Structs können in dieser Methode gelesen aber nicht verändert werden.

- `&mut self` ist eine Exclusive Reference zum Wert des Structs, das heißt, Methoden dieser Art können die Inhalte des Structs lesen und verändern.

Die Entscheidung, wann welche Variante von Methode verwendet wird, hängt stark vom jeweiligen Use Case ab. Allgemein gilt, dass das Schlüsselwort `mut` nur verwendet werden sollte, wenn die Methode oder Funktion schreibenden Zugriff auf die Daten

des Structs benötigt. Die Entscheidung zwischen der Übergabe durch einen Move und einer Referenz muss von Fall zu Fall getroffen werden.

Wichtig zu erwähnen ist, dass im verwendenden Code der erste Parameter, also eine Variante von `self`, von Methoden nicht manuell übergeben werden muss, sondern dass der Compiler die benötigte Art von Referenz automatisch, sofern möglich, erzeugt und übergibt. Dieses Verhalten ist in dem in Abb. 2.11 dargestellten Beispiel in der `run`-Funktion ersichtlich. Die Variable `house` besitzt den Wert des Structs. Für den Aufruf von `house.add_resident();` wird eine Shared Reference benötigt. Der Compiler erzeugt diese automatisch aus der `house` Variable und übergibt sie an die Methode. Analoges gilt für die weiteren Methodenaufrufe, wobei auch der Compiler bei diesen automatischen Umwandlungen alle zuvor beschriebenen Regeln des Typsystems in Rust einhalten muss.

2.1.10 Enums

Enums in Rust sind als Algebraic Data Types (ADTs) implementiert. Der Vorteil von ADTs gegenüber klassischen Enums wie beispielsweise in C ist, dass ADTs zusätzlich zu der Darstellung verschiedener Varianten Daten beinhalten können, wodurch beliebig tiefe Datentypen erzeugt werden können. Dies erleichtert die Modellierung verschiedenster Szenarien, wobei der Compiler garantiert, dass immer alle Varianten berücksichtigt werden. Mittels des `match`-Schlüsselworts, welches Pattern Matching implementiert, werden in Rust Enums destrukturiert, um die jeweilige Variante zu bestimmen und eventuell beinhaltete Daten zu extrahieren. Der Compiler überprüft bei jeder `match`-Verwendung, ob alle Varianten berücksichtigt wurden oder ob ein sogenanntes Wildcard-Pattern existiert, das alle nicht explizit definierten Varianten verarbeitet. In Abb. 2.12 ist ein Beispiel für die Verwendung von Enums in Rust dargestellt.

2.1.11 Traits

Um in Rust konkrete Implementierungen von deren bereitgestellten Funktionen trennen zu können, werden Traits verwendet. Traits können aus assoziierten Funktionen, Typen und Konstanten bestehen, beschreiben eine Funktion oder Schnittstelle eines Typs

```rust
struct House {
    residents: u32
}

impl House {
    // Assoziierte Funktion
    fn new() -> House {
        House {
            residents: 0
        }
    }
    // &mut self weil residents bearbeitet werden muss
    fn add_resident(&mut self) -> () {
        self.residents += 1;
    }
    // &self da nur Lesezugriff nötig
    fn get_residents(&self) -> u32 {
        self.residents
    }
    // self weil Move dazu führt, dass der Speicher am Ende des Scopes
    // freigegeben wird.
    fn destroy(self) -> () {
        // self wird hier gedropped
    }
}

fn run() {
    let mut house = House::new();
    house.add_resident();
    let residents = house.get_residents();
    house.destroy();
    // house kann ab hier nicht mehr verwendet werden, da der Wert
    // in destroy gedropped wurde
}
```

Abbildung 2.11: Beispiel für ein Struct mit verschiedenen assoziierten Funktionen und Methoden.

und sind vergleichbar mit Interfaces in Java und Typklassen in Haskell. In Abb. 2.13 ist ein Beispiel für die Definition eines Traits dargestellt.

Traits können einerseits in Trait Bounds in generischen Datentypen, wie in Ab-

```rust
enum Direction { Left, Right }
enum Action { SayHello, Move(Direction, u32) }
fn move_player(player: &mut Player, input: Action) {
    match input {
        Action::SayHello => { println!("Hello!"); }
        Action::Move(direction, distance) => {
            match direction {
                Direction::Left => { player.add_x_pos(-distance); }
                Direction::Right => { player.add_x_pos(distance); }
            }
        }
    }
}

fn run() {
    let mut player = Player::new();
    move_player(&mut player, Action::SayHello);
    move_player(&mut player, Action::Move(Direction::Left, 10));
}
```

Abbildung 2.12: Enums können für die Modellierung verschiedenster Szenarien verwendet werden.

```rust
trait Default {
    fn default() -> Self;
}
```

Abbildung 2.13: Das `Default`-Trait wird verwendet, um für einen Typ einen Standardwert zu definieren.

schnitt 2.1.13 beschrieben, verwendet werden, andererseits können Traits in Trait Objects, wie in Abschnitt 2.1.14 beschrieben, verwendet werden. Der Unterschied zwischen diesen beiden Optionen ist, dass generische Datentypen Static Dispatch für die Ausführung des Codes verwenden und Trait Objects Dynamic Dispatch. Die Details dieser beiden grundsätzlich verschiedenen Ansätze werden in den jeweiligen Kapiteln beschrieben. Je nach Anwendungsfall muss die für das bestehende Problem bessere Lösung ermittelt werden.

Über sogenannte Supertraits kann ein Trait definieren, dass Datentypen, die dieses Trait implementieren möchten, die beschriebenen Supertraits ebenfalls implementieren müssen. Ein Beispiel hierfür ist in Abb. 2.14 dargestellt. Die Definition von Supertraits

ermöglicht die Verwendung der von den Supertraits bereitgestellten Funktionalitäten innerhalb der Implementation des Traits.

```
// Leerer Marker-Trait
trait Animal {}
// Say kann für einen Typ nur implementiert werden, wenn Animal auch
// implementiert wurde.
trait Say: Animal {
    fn say() -> ();
}
```

Abbildung 2.14: Durch die Angabe von Supertraits kann ein Trait die Implementation anderer Traits voraussetzen.

Häufig wird eine Abstraktion über Datentypen benötigt, die mehr als ein Trait implementieren müssen, damit diese verwendbar sind. Einschränkungen dieser Art können durch die Aneinanderreihung der Traits mit + formuliert werden. Ein Beispiel für die Trait-Kombination ist in Abb. 2.15 dargestellt.

```
trait Title {
    fn get_title(&self) -> String;
}
trait Dated {
    fn get_date(&self) -> Date;
}

// Akzeptiert jeden Typ A, der Title und Dated implementiert.
fn process<A: Title + Dated>(article: A) {
    let title = article.get_title();
    let creation_date = article.get_date();
}
```

Abbildung 2.15: Muss ein Typ mehrere Traits implementieren, so kann dies über die Kombination der Traits mittels + ausgedrückt werden.

2.1.12 Smart Pointer

Smart Pointer sind Datentypen, die einen Speicher besitzen und die das `Deref`- und das `Drop`-Trait implementieren. In der Regel weisen Smart Pointer zusätzlich eine

oder mehrere spezielle Funktionen auf, die die Funktionen des gekapselten Wertes erweitern.

Einige ausgewählte Beispiele für Smart Pointer sind:

- `Box<T>`: Speichert einen Wert auf dem Heap.
- `String`: Hält Daten auf dem Heap und garantiert, dass diese eine gültige UTF-8-Repräsentation haben und bietet zusätzliche für Strings typische Funktionen an.
- `Rc<T>`: Speichert einen Wert auf dem Heap und hält intern eine Anzahl von aktiven Referenzen. Dies ermöglicht, dass mehrere Owner gleichzeitig existieren können, was gemäß dem Ownership-Modell nicht direkt möglich wäre.

Die Implementation des `Drop`-Traits garantiert, dass sämtliche verwendeten Ressourcen gemäß dem in Rust üblichen Speicherfreigabeprinzip freigegeben werden. Durch die Implementation des `Deref`-Traits können Smart Pointer wie normale Referenzen mittels des Dereferenzierungsoperators $*$ dereferenziert werden, da die im `Deref`-Trait zu implementierende `deref()`-Funktion eine normale Referenz zum gehaltenen Wert zurückgibt. Diese Referenz kann dann, wie in Rust üblich, mittels $*$ dereferenziert werden, wodurch ein Smart Pointer als transparente Erweiterung des gekapselten Werts verwendet werden kann. Das Aufrufen der `deref()`-Funktion übernimmt der Compiler automatisch, weshalb ein manueller Aufruf dieser Funktion nur nötig ist, wenn mehrere Indirektionen vorhanden sind, die der Compiler nicht auflösen kann. Die automatische Referenzierung und Dereferenzierung durch den Compiler wird Deref Coercion genannt und ermöglicht besser lesbaren Code zu entwickeln, wobei der Compiler die korrekte Verwendung der Referenzen garantiert. Ein Beispiel für den Deref-Coercion-Mechanismus ist in Abb. 2.16 dargestellt.

```
// Box ist ein Smart Pointer, der den Wert 3 auf dem Heap speichert.
let mut boxed_number: Box<i32> = Box::new(3);

// x1 und x2 sind aufgrund der Deref Coercion identisch
let x1: &i32 = &(*(boxed_number.deref()));
let x2: &i32 = &boxed_number;
```

Abbildung 2.16: Smart Pointer können mittels des Dereferenzierungsoperators $*$ wie normale Referenzen dereferenziert werden.

2.1.13 Generische Datentypen

Generische Datentypen werden verwendet, um Datentypen und Funktionen über ein oder mehrere Typen zu generalisieren. Generische Datentypen verfügen über mindestens einen generischen Typparameter, der in spitzigen Klammern geschrieben wird. Ein sehr einfacher, aber in Rust häufig verwendeter generischer Datentyp ist der `Option<T>`-Typ, der in Abb. 2.17 dargestellt ist. `Option<T>` wird verwendet, um zu signalisieren, dass das Ergebnis einer Funktion eventuell nicht vorhanden sein könnte. Dieser Typ ist aufgrund seiner Einfachheit ein optimales Beispiel für einen generischen Datentyp, da dieser über nur einen Typparameter verfügt und dessen Aufgabe, nämlich die Repräsentation des Typs des optionalen Rückgabewerts einer Berechnung, sofort ersichtlich ist.

```
enum Option<T> {
    Some(T),
    None
}
```

Abbildung 2.17: Der Option-Typ ist ein generisches Enum.

Enums, Structs, Traits und Funktionen können durch die Verwendung generischer Typparameter über eine beliebige Anzahl an Typen generalisiert werden. Durch die Verwendung von sogenannten Trait Bounds, also die Angabe, dass ein generischer Typ ein oder mehrere Traits implementieren muss, kann die benötigte Funktion des generischen Typs von der tatsächlichen Implementierung entkoppelt werden. Ein Beispiel für dieses Prinzip ist in Abb. 2.18 dargestellt.

Bei der Verwendung generischer Datentypen wird für jeden konkreten Typ, der als Typparameter in einem generischen Datentyp verwendet wird, durch den Compiler eine Kopie der entsprechenden Funktion beziehungsweise des Datentyps generiert. Dieser Prozess wird Monomorphization genannt. Durch die Generierung von typspezifischem Code kann der Compiler diesen pro konkretem Typ optimieren, wodurch Performancesteigerungen möglich sind. Der generierte Code wird zur Laufzeit mittels Static Dispatch ausgeführt, wodurch eine höhere Performance als bei der Verwendung von Dynamic Dispatch möglich ist, da im Rahmen von Dynamic Dispatch bei dem Aufruf einer Methode eine Indirektion aufgrund des Virtual Method Table auftritt. Ein

```rust
trait SayHello {
    fn say_hello(&self) -> ();
}
// Trait Bound: jeder Typ T, der SayHello implementiert
fn say<T: SayHello>(t: &T) {
    t.say_hello();
}

struct Person1 {}
impl SayHello for Person1 {
    fn say_hello(&self) -> () { println!("hello from person 1!") }
}
struct Person2 {}
impl SayHello for Person2 {
    fn say_hello(&self) -> () { println!("hello from person 2!") }
}

fn run() {
    let p1 = Person1 {};
    let p2 = Person2 {};
    say(&p1);
    say(&p2);
}
```

Abbildung 2.18: Durch die Angabe von Trait Bounds in generischen Datentypen können die Funktionen der angegeben Traits verwendet werden, ohne deren tatsächliche Implementierung kennen zu müssen.

Nachteil der Monomorphization ist, dass die generierte Binärdatei in der Regel größer ist, da Code-Segmente durch den Monomorphization-Prozess dupliziert werden.

Ein weiterer Nachteil generischer Datentypen ist, dass keine Vermischung verschiedener Datentypen innerhalb eines generischen Datentyps möglich ist. Wird beispielsweise eine Liste von Elementen benötigt, deren Typen sich unterscheiden, die aber ein gemeinsames Trait implementieren, so ist die Implementierung dieses Datentyps mittels generischer Datentypen nicht möglich, da aufgrund der Monomorphization eine generische Liste `Vec<T>` Elemente genau eines konkreten Typs speichern kann. Das beschriebene Szenario wäre ein typischer Anwendungsfall für Dynamic Dispatch, das in Rust mittels Trait Objects, die in Abschnitt 2.1.14 beschrieben werden, implementiert werden kann.

2.1.14 Trait Objects

Trait Objects repräsentieren den Wert eines Typs, der ein oder mehrere bestimmte Traits implementiert, wobei der tatsächliche Wert zur Compile-Zeit nicht bekannt ist. Aufgrund dieser Definition können auch nur die Funktionen eines Trait Objects verwendet werden, die in den entsprechenden Traits beschrieben sind. Damit ein Trait als Basis für ein Trait Object verwendet werden kann, muss dieses bestimmte Eigenschaften erfüllen, um sinnvoll und sicher im Kontext von Trait Objects verwendet werden zu können. Diese Regeln sind detailliert in *The Rust Reference* (2022) beschrieben und werden Object Safety Rules genannt. Trait Objects werden in Rust mit dem Schlüsselwort `dyn`, gefolgt vom jeweiligen Trait, gekennzeichnet. Eine Kombination mehrerer Traits, wie in Abschnitt 2.1.13 beschrieben, ist ebenfalls möglich.

Da der konkrete Typ eines Trait Object nicht bekannt ist, ist die Größe des jeweiligen Werts zur Compile-Zeit ebenfalls nicht bekannt. Typen, die diese Eigenschaft aufweisen, werden in Rust Dynamically Sized Types (DSTs) genannt. Damit der Compiler die Speicherauslegung von Programmen generieren kann, muss die Größe aller Werte zur Compile-Zeit bekannt sein. Diese Einschränkung ist in Kombination mit Trait Objects dahingehend vereinbar, als dass jedes Trait Object nicht direkt verwendet werden kann, sondern immer hinter einem Referenztyp verwendet werden muss, da Referenztypen eine konstante Größe besitzen. Referenztypen, hinter denen sich ein Trait Object befindet, bestehen in Rust aus zwei Pointern. Ein Pointer zeigt hierbei auf den tatsächlichen Wert, der die Traits des Trait Objects implementiert. Der zweite Pointer zeigt auf einen Virtual Method Table, in dem für jede Funktion der Traits ein Pointer zu der entsprechenden Implementierung des konkreten Typs gespeichert ist. Zur Laufzeit muss bei dem Aufruf einer Funktion eines Trait Objects zuerst der entsprechende Pointer im Virtual Method Table gefunden werden und anschließend die Funktion hinter diesem aufgerufen werden. Eine Limitierung dieses Systems ist, dass keine direkte Möglichkeit besteht, aus einem Trait Object den konkreten Typ zu extrahieren. Eine Funktionalität, die genau diese Limitierung umgeht und eine dynamische Typisierung ermöglicht, wird in Abschnitt 2.1.17 beschrieben. Die durch Trait Objects erreichte Entkopplung der tatsächlichen Implementierungen von den durch Traits beschriebenen Schnittstellen wird Dynamic Dispatch genannt und wird für die Lösung bestimmter Problemstellungen, in denen die tatsächlichen Werte zur Compile-Zeit nicht bekannt sind, benötigt.

Trait Objects lösen das in Abschnitt 2.1.13 beschriebene Problem von Listen, in

denen gemischte Datentypen mit einem gemeinsamen Trait gespeichert werden sollen. Basierend auf dem gemeinsamen Trait werden Referenzen zu Trait Objects in einer Liste gespeichert. Eine mögliche Lösung dieses Beispiels ist in Abb. 2.19 dargestellt.

```
trait SayHello {
    fn say(&self) -> ();
}
struct A {}
impl SayHello for A {
    fn say(&self) -> () { println!("Hello from A"); }
}
struct B {}
impl SayHello for B {
    fn say(&self) -> () { println!("Hello from B"); }
}

fn run() {
    // Box ist ein Smart Pointer und kann daher als Referenztyp
    // verwendet werden
    let list: Vec<Box<dyn SayHello>> = vec![
        Box::new(A{}),
        Box::new(B{})
    ];
    for elem in list {
        elem.say();
    }
}
```

Abbildung 2.19: In der Liste werden Trait Objects des Traits `SayHello` gespeichert. Als Referenztyp wird `Box<T>` verwendet.

2.1.15 Modelle der Nebenläufigkeit

In Rust existieren zwei grundlegende Arten von Nebenläufigkeit:

- Nebenläufigkeit basierend auf der direkten Verwendung von Threads.
- Nebenläufigkeit mittels Futures.

Der erste Ansatz basiert auf der direkten Verwendung von Threads und erfordert die in der Softwareentwicklung üblichen Maßnahmen, um potenzielle Probleme gleichzeitiger Lese- und/oder Schreibzugriffe mehrere Threads auf denselben Speicher zu verhindern. Viele der möglichen Probleme werden bereits im Vorhinein durch die in Rust existierenden Einschränkungen wie beispielsweise die des Ownership-Modells ausgeschlossen, oft werden aber Operationen benötigt, die in Rust aufgrund der strikten Regeln nicht direkt implementierbar sind. Ein Beispiel hierfür wäre ein geteilter Speicher, auf den mehrere Threads schreibend zugreifen können. In Rust steht eine Vielzahl an Hilfsfunktionen und Typen wie beispielsweise `Mutex<T>` und `Rc<T>` zur Verfügung, um Probleme dieser Art lösen zu können. Üblicherweise sollte die Anzahl an verwendeten Threads relativ zu den vorhandenen CPU-Kernen niedrig gehalten werden, da die vom Betriebssystem durchgeführten Kontextwechsel verhältnismäßig rechenintensiv sind. Bei einer großen Anzahl an Threads werden viele Kontextwechsel durchgeführt, wodurch die vorhandenen Ressourcen nicht optimal genutzt werden.

Der zweite Ansatz löst dieses Problem, indem es die Einheiten der Ausführung, die Threads, von den eigentlichen Berechnungen trennt. Alle Berechnungen in diesem Modell werden durch sogenannte Futures repräsentiert. Ein Future ist die Repräsentation einer Berechnung, die zu bestimmten Zeitpunkten, an denen ein noch nicht vorhandenes Ergebnis benötigt wird, unterbrochen und zu einem späteren Zeitpunkt, an dem das Ergebnis vorhanden ist, fortgesetzt werden kann. Für die Ausführung von Futures wird die Implementation einer Async-Runtime benötigt. In Rust werden alle Schnittstellen der Async-Runtime durch Traits beschrieben, weshalb jede beliebige Async-Runtime-Implementierung für die Ausführung von Futures verwendet werden kann. Es gibt keine in die Sprache integrierte Async-Runtime-Implementation, da Rust als Systemprogrammiersprache entwickelt wurde und dies für die meisten Anwendungsfälle einen nicht benötigten Overhead bedeuten würde. Eine Async-Runtime führt die Futures aus und dient gleichzeitig als Schnittstelle zum IO-Eventsystem des Betriebssystems.

Die Schnittstelle von Futures definiert eine `poll()`-Methode, die als Ergebnis entweder `Ready<T>` oder `Pending` zurückgibt. `Ready<T>` bedeutet, dass der Rückgabewert vom Typ `T` der Berechnung vorhanden ist und die Berechnung des Futures somit abgeschlossen ist. `Pending` signalisiert, dass das Future auf ein für die weitere Berechnung benötigtes Ergebnis wartet oder bei dem letzten Aufruf der `poll()`-Methode nicht abgeschlossen werden konnte. Sobald das für die weitere Berechnung benötigte Ergebnis vorhanden ist, wird die `poll`-Methode des Futures erneut durch die Async-Runtime

aufgerufen und somit die Berechnung vorangetrieben. Die `poll()`-Methode jedes Futures wird von der Async-Runtime so lange aufgerufen, bis alle vorhandenen Futures abgeschlossen sind.

Die Verwendung einer Async-Runtime hat den Vorteil gegenüber Threads, dass eine sehr große Anzahl an Futures gleichzeitig existieren und ausgeführt werden kann, da in der Regel Threads eines Thread-Pools für die Ausführung von Futures wiederverwendet werden, wodurch die Anzahl der Kontextwechsel minimiert wird und somit kein übermäßiger Overhead entsteht. Ein weiterer Vorteil einer Async-Runtime ist, dass durch die Anbindung an das IO-Eventsystem des Betriebssystems IO-Operationen, deren Ergebnis in der Regel nicht sofort vorhanden sind, sehr gut gehandhabt werden können. Weil Futures eine zusätzliche Abstraktion über Threads implementieren, sind diese nicht so performant wie die direkte Verwendung von Threads.

Weil dieses Kapitel nur als grobe Übersicht über die vorhandenen Möglichkeiten der Nebenläufigkeit in Rust dienen soll, sei an dieser Stelle für weitere Details der beiden Ansätze auf *The Rust Reference* (2022) verwiesen.

2.1.16 Closures

Aufgrund des funktionalen Einflusses anderer Sprachen auf Rust wurden Closures in die Sprache integriert. In Rust existieren drei verschiedene Arten von Closures, die sich aufgrund des Ownership-Modells erklären lassen:

- `Fn`: Hält mindestens eine Shared Reference auf die Umgebung.
- `FnMut`: Hält mindestens eine Exclusive Reference auf die Umgebung.
- `FnOnce`: Kann den Besitz der in der Umgebung befindlichen Werte übernehmen und ist nur einmal aufrufbar.

Jede Closure in Rust bekommt, abhängig von der jeweiligen Verwendung, eine oder mehrere der drei beschriebenen Closure-Arten, die durch Marker-Traits repräsentiert werden, zugewiesen. Jede Closure in Rust implementiert `FnOnce`, kann also zumindest einmal aufgerufen werden. Implementiert eine Closure zusätzlich `FnMut`, so kann diese Veränderungen an den Werten der Umgebung durchführen und mehrmals aufgerufen werden. Muss eine Closure mehrfach aufgerufen werden können, aber keine Änderungen

an den Werten der Umgebung durchführen, so implementiert diese `Fn`. Ein Beispiel für die drei Arten von Closures ist in Abb. 2.20 dargestellt.

Da jede Closure in Rust das `FnOnce`-Trait implementiert, besteht eine Zuweisungskompatibilität von `Fn` und `FnMut` wenn eine Instanz von `FnOnce` benötigt wird. Wie an allen Stellen in Rust wird auch für die durch Closures implizit erstellten Referenzen die Einhaltung der Ownership-Modell-Regeln durch den Compiler sichergestellt.

```
let mut text = String::from("hello");
let closure_fn = || {
    println!("{}", text);
};
let closure_fnmut = || {
    text.push_str(" world");
};
let closure_fnonce = || {
    drop(text);
};
```

Abbildung 2.20: Die Art jeder Closure wird vom Compiler automatisch, basierend auf der Verwendung der Umgebung, bestimmt.

2.1.17 Das Any-Trait

Rust ist eine statisch typisierte Sprache, was bedeutet, dass zur Compile-Zeit alle Datentypen bekannt sein müssen. Die Informationen über die Datentypen werden durch die Kompilierung eliminiert und sind zur Laufzeit nicht mehr vorhanden. In gewissen Situationen wird der Zugriff auf die Metadaten von Typen zur Laufzeit benötigt. In Rust ist dies über das Any-Trait möglich. Trait Objects, die das Any-Trait implementieren, verfügen über eine pro Typ eindeutige `TypeId`, anhand derer der Typ bestimmt werden kann, und können, anders als bei anderen Trait Objects üblich, zum tatsächlichen Typ zurückkonvertiert werden. Die Konvertierung von einem Trait Object zu dessen tatsächlichem Typ wird Downcasting genannt. Dieser Mechanismus ermöglicht eine dynamische Typisierung in einer statisch typisierten Sprache. Die meisten in Rust vorhandenen Datentypen implementieren das Any-Trait, weshalb dieses Prinzip in fast allen Fällen, in denen eine dynamische Typisierung benötigt wird, anwendbar ist.

```
fn process(input: Box<dyn Any>) {
    // input muss dereferenziert werden, da ansonsten type_id() auf Box
    // aufgerufen werden würde und nicht auf dem eigentlichen Typ des
    // Trait Objects.
    if (*input).type_id() == TypeId::of::<String>() {
        let string = input.downcast_ref::<String>()
            .expect("Downcast nicht möglich");
        println!("String: {}", string);
    } else if (*input).type_id() == TypeId::of::<u32>() {
        let num = input.downcast_ref::<u32>()
            .expect("Downcast nicht möglich");
        println!("u32: {}", num);
    }
}

fn run() {
    let input1 = Box::new(String::from("hello"));
    let input2 = Box::new(123_u32);
    process(input1);
    process(input2);
}
```

Abbildung 2.21: Durch die Verwendung des Any-Traits stehen zur Laufzeit Metainformationen über die Datentypen von Trait Objects zur Verfügung. Trait Objects können dadurch zu ihrem tatsächlichen Typ zurückkonvertiert werden.

2.2 Das Aktor-Modell

Das Aktor-Modell basiert auf den in Hewitt, Bishop und Steiger (1973) beschriebenen Ideen für die Modellierung von Nebenläufigkeit und dem in Hoare (1978) beschriebenen Konzept des Message Passings und wird in Agha (1985) detailliert beschrieben. Im Aktor-Modell entsprechen sogenannte Aktoren der Basis jeder Berechnung. Aktoren kommunizieren untereinander ausschließlich über Nachrichten. Bei dem Empfangen einer Nachricht kann ein Aktor:

- Nachrichten an andere Aktoren senden.
- Neue Aktoren erzeugen.
- Das Verhalten für das Empfangen zukünftiger Nachrichten ändern.

Alle empfangenen Nachrichten werden in einem Postfach gespeichert, das gemäß dem FIFO-Prinzip abgearbeitet wird. Zu jedem Zeitpunkt wird genau eine Nachricht verarbeitet. Der Zustand eines Aktors wird in dem klassischen Aktor-Modell durch dessen Verhalten repräsentiert. Im Aktor-Modell existiert keine Garantie für die Einhaltung der Empfangsreihenfolge von Nachrichten, was bedeutet, dass zwei gleichzeitig gesendete Nachrichten in einer nicht deterministischen Reihenfolge empfangen werden können. Des Weiteren wird jede Nachricht nach dem Best-Effort-Prinzip zugestellt und kann aufgrund von potenziell auftretenden Fehlern oder fehlenden Kapazitäten verloren gehen. Wenn das Empfangen einer Nachricht aufgrund der Anwendung benötigt wird, so muss ein Protokoll definiert werden, das diese Anforderung erfüllt und beispielsweise die Bestätigung empfangener Nachrichten erfordert.

Das Aktor-Modell besitzt den Vorteil, dass keine in anderen Multithreading-Umgebungen üblichen Probleme auftreten können, da kein Speicher zwischen mehreren Aktoren geteilt wird und pro Aktor nur eine Nachricht zu einem bestimmten Zeitpunkt verarbeitet wird. Dies ermöglicht die parallele Ausführung von Aktoren ohne weitere besondere Synchronisierungsmaßnahmen.

Weil Aktoren keinen Speicher teilen, erfüllt jeder Aktor das Lokalitätsprinzip, was bedeutet, dass einem Aktor zum Zeitpunkt der Verarbeitung einer Nachricht nur folgende Informationen zur Verfügung stehen:

- Informationen, die in der empfangenen Nachricht übertragen wurden.
- Informationen, die der Aktor zu einem früheren Zeitpunkt über eine Nachricht empfangen hat.
- Informationen, die im Rahmen der Verarbeitung der Nachricht generiert wurden.

Aktoren können entsprechend ihren logischen Aufgaben in Aktor-Systeme gruppiert werden.

Trotz der Einfachheit dieses Modells löst dieses viele der heute häufig auftretenden Probleme. Beispiele für Probleme, die durch das Aktor-Modell gelöst werden, sind:

- Aufgrund der vollständigen Isolierung von Aktoren und die ausschließliche Kommunikation über Nachrichten können diese ohne weitere Maßnahmen auf verschiedenen Systemen ausgeführt werden.

- Weil kein Speicher zwischen Aktoren geteilt wird, können keine gleichzeitigen Zugriffe auf denselben Speicher entstehen. Es werden keine weiteren Synchronisierungsmaßnahmen benötigt.
- Die Verwendung von Aktoren erzwingt eine logische Aufteilung der Aufgaben eines Systems auf Aktoren, wodurch das Gesamtsystem im Allgemeinen modularer und besser strukturiert wird.
- Die Best-Effort-Zustellung von Nachrichten erzwingt ein Design, das Ausfälle von Nachrichten und Aktoren vorsieht. Dadurch werden automatisch robustere Systeme entwickelt.

Weil alle der genannten Probleme vor allem häufig in Multithreading-Umgebungen und verteilten Systemen auftreten, werden Implementationen des Aktor-Modells auch meist in diesen Bereichen verwendet.

2.3 Existierende Aktor-Modell-Implementierungen in Rust

In diesem Kapitel werden einige ausgewählte Aktor-Bibliotheken in Rust beschrieben und deren Einschränkungen und Eigenschaften erläutert.

2.3.1 Actix

Actix ist eine Bibliothek in Rust, die eine Variante des Aktor-Modells implementiert. In Actix sind alle Aktoren statisch typisiert, was bedeutet, dass sich die unterstützten Nachrichtenarten und deren Verarbeitungsaktionen zur Laufzeit nicht ändern können. Damit diese strikte statische Typisierung ohne größeren Code-Aufwand funktioniert, werden Makros für die Generierung von Code für Nachrichtentypen verwendet. Weiters verwendet Actix intern eine Async-Runtime, wobei die Syntax derer durch die API bis zum Nutzenden der Bibliothek durchdringt.

2.3.2 Acteur

Acteur ist, ähnlich wie Actix, eine Rust-Bibliothek, deren Ziel es ist, Funktionalitäten zu bieten, die ähnlich derer des Aktor-Modells sind, wobei explizit in der

Projektbeschreibung in Bonet (2022) erwähnt wird, dass sich das Projekt nicht an das Aktor-Modell hält. Die Bibliothek ist als höhere Abstraktion über das Aktor-Modell gedacht, weshalb beispielsweise `async` explizit in jeder Verarbeitung einer Nachricht verwendet werden kann. Acteur ist, ähnlich wie Actix, statisch typisiert und erlaubt keine Änderungen des Verhaltens eines Aktors zur Laufzeit. Die Bibliothek wurde zum aktuellen Zeitpunkt seit über zwei Jahren nicht mehr gewartet und steht für die Übernahme durch einen Entwickelnden zur Verfügung.

2.3.3 Weitere Bibliotheken

Zum Zeitpunkt der Verfassung dieser Arbeit existieren viele weitere Implementationen des Aktor-Modells in Rust, wobei alle gemeinsam haben, dass diese Regeln des Aktor-Modells verletzen oder bestimmte Funktionen, wie beispielsweise das Ändern des Verhaltens eines Aktors zur Laufzeit, nicht ermöglichen.

Zum Beispiel wird durch die Verwendung von generischen Datentypen für die Implementation von Nachrichtenverarbeitungsaktionen verhindert, dass sich diese zur Laufzeit ändern können. Diese Einschränkung hat zur Folge, dass ein Aktor dieser Art nur eine zur Compile-Zeit fixierte Menge an Nachrichtentypen empfangen kann, wodurch das Erzeugen von Aktoren mit dynamischen Verhalten nicht möglich ist. Weil die Grundidee des Aktor-Modells besagt, dass ein Aktor sein Verhalten nach jeder Nachrichtenverarbeitung ändern kann, und das Verhalten eines Aktors die verarbeitbaren Nachrichtentypen beinhaltet, entspricht eine Modellierung dieser Art nicht der Idee des Aktor-Modells und schränkt die Verwendung dessen stark ein. Die meisten der existierenden Aktor-Modell-Implementationen in Rust folgen diesem statischen Ansatz.

Ein weiteres, häufig verwendetes Design, das nicht dem Aktor-Modell entspricht, ist die Rückgabe eines Futures bei dem Senden einer Nachricht an einen Aktor, wobei das Future die Antwort auf die gesendete Nachricht repräsentiert. Dieser Ansatz ist dahingehend problematisch, als dass der empfangende Aktor auf die Vervollständigung des Futures und somit das Ergebnis warten kann, wodurch die Ausführung von Aktoren blockiert wird.

Ein weiterer Punkt, der von keiner der existierenden Bibliotheken berücksichtigt wird, ist das Testen von Aktoren. Im Rahmen dieser Arbeit wird ein Test-Framework

entwickelt, das das Testen von Aktoren drastisch erleichtert und den manuell zu schreibenden Code minimiert, wodurch häufiges Testen unterstützt wird.

Die ausgewählten beschriebenen Punkte beschreiben repräsentativ die Unzulänglichkeiten existierender Aktor-Modell-Bibliotheken in Rust und motivieren die Ziele der im Rahmen dieser Arbeit entwickelten Bibliothek. In den folgenden Kapiteln werden die getroffenen Design-Entscheidungen ausführlich diskutiert und deren Auswirkungen auf das Funktionieren der Bibliothek beschrieben.

2.4 Beispiel einer existierenden Aktor-Modell-Implementierung: Erlang

Erlang ist eine erstmals 1986 veröffentlichte, funktionale Programmiersprache deren Ziel es ist, verteilte, ausfallsichere, hochverfügbare und gut skalierbare Systeme zu ermöglichen. Die Sprache ist dynamisch typisiert und bietet die für eine funktionale Programmiersprache üblichen Paradigmen wie beispielsweise Pattern Matching und Immutable Types an. Für die Speicherverwaltung wird ein automatischer Garbage-Collection-Mechanismus, der in Stenman (2022, S. 123ff) beschrieben wird, verwendet, der nicht mehr benötigten Speicher ohne Zutun des Entwickelnden freigibt.

Erlang war, gemäß Vinoski (2007), eine der ersten Programmiersprachen, die das Aktor-Modell implementiert haben. Erlang grenzt sich gegenüber anderen Programmiersprachen dahingehend ab, als dass die gesamte Sprache auf ähnlichen Prinzipien wie denen des Aktor-Modells basiert. Da Erlang basierend auf praktischen Anforderungen entwickelt wurde, ergab sich die Ähnlichkeit zum Aktor-Modell dadurch, dass die Anforderungen genau denen entsprachen, die üblicherweise für den Einsatz eines Aktor-Modells sprechen würden.

In Erlang repräsentieren sogenannte Prozesse einen Aktor. Prozesse sind leichtgewichtige Green Threads, die in einer virtuellen Maschine, der BEAM, von einem preemptiven Scheduler ausgeführt werden. Die Kommunikation zwischen Prozessen geschieht ausschließlich durch Nachrichten, wobei gesendete Daten kopiert und nicht geteilt werden. Durch die vollständige Isolierung von Prozessen untereinander können Prozesse auf verschiedenen Servern verteilt werden und weiterhin miteinander kommunizieren. Diese Entkopplung garantiert eine sehr gute Skalierbarkeit von Erlang-Systemen.

Eine der Hauptparadigmen von Erlang ist „Let it crash“. Die Idee hierbei besteht darin, dass Prozesse, in denen ein Fehler auftritt, zerstört und komplett neu gestartet werden. Da ein Fehler in der Regel auf einen fehlerhaften Zustand eines Prozesses zurückzuführen ist, wird der Fehler somit behoben. Weil Prozesse sehr leichtgewichtig sind, sind die Kosten für die Zerstörung und Neuerstellung so minimal, dass sich dieses Paradigma sehr gut implementieren lässt. Damit dieses Prinzip funktioniert, muss das System erkennen, wenn ein Fehler in einem Prozess auftritt. Hierfür wurden sogenannte Supervision-Hierarchien eingeführt, in denen jeder Prozess von einem anderen Prozess überwacht wird. Der überwachende Prozess überprüft regelmäßig den Zustand des zu überwachenden Prozesses und startet diesen im Fehlerfall neu. Diese Hierarchien können beliebig tief und mit speziellen Neustartkriterien erweitert werden.

Vor allem bei Systemen, die hochverfügbar und gut skalierbar sein müssen, wird Erlang erfolgreich eingesetzt. Als Beispiel für eine Anwendung, die Erlang aufgrund der oben beschriebenen Vorteile extensiv in ihren Systemen verwendet, ist WhatsApp. Die konkreten Gründe für die Verwendung von Erlang in der Systemarchitektur von WhatsApp werden in Reed (2014) und Larvik (2018) näher beschrieben, entsprechen aber unter anderem den genannten Vorteilen.

3 Entwicklung der Bibliothek

In diesem Kapitel wird zuerst das Design der entwickelten Bibliothek vorgestellt, bevor die wichtigsten getroffenen Design-Entscheidungen diskutiert und erläutert werden. Anschließend wird die Implementation der Komponenten der Bibliothek beschrieben und anhand von Beispielen erklärt. In dem letzten Teil dieses Kapitels wird das entwickelte Test-Framework und die Ideen hinter der Implementation dessen erläutert.

3.1 Design

In diesem Kapitel wird das Design der entwickelten Bibliothek vorgestellt und anschließend die Verwendung derer anhand eines Einführungsbeispiels demonstriert.

3.1.1 Design der Bibliothek

Die Bibliothek wurde gemäß den in Agha (1985, S. 12ff) beschriebenen Grundkonzepten des Aktor-Models entworfen. Ein Aktor wird durch einen Zustand und ein Verhalten durch Nutzende der Bibliothek definiert:

- Der Zustand eines Aktors ist isoliert und kann nur durch den Aktor selbst gelesen und verändert werden.
- Das Verhalten eines Aktors definiert sämtliche mögliche Verhaltensmuster eines Aktors zu einem bestimmten Zeitpunkt wie beispielsweise, was ein Aktor direkt nach dem Start ausführen soll oder was ein Aktor vor dem Stoppen ausführen soll. Weiters definiert das Verhalten, welche Typen von Nachrichten ein Aktor empfangen kann und wie diese verarbeitet werden.

Ein Aktor kann bei der Verarbeitung einer Nachricht seinen Zustand und sein Verhalten ändern, Nachrichten verschicken, neue Aktoren erzeugen und die eigene Ausführung stoppen oder neu starten.

Nachrichten, deren Typ von einem Aktor nicht unterstützt werden, werden ignoriert und gelöscht. Diese Design-Entscheidung wird in Abschnitt 3.2.5 näher erläutert, ermöglicht aber eine effiziente Speichernutzung.

Weil sich das Verhalten eines Aktors nach jeder verarbeiteten Nachricht ändern kann und das Verhalten auch die unterstützten Nachrichtentypen eines Aktors definiert, können sich diese ebenfalls dynamisch ändern.

Für das Empfangen und temporäre Speichern von Nachrichten besitzt jeder Aktor ein Postfach. Das Postfach eines Aktors entspricht konzeptionell einer FIFO-Warteschlange, wobei jeweils die älteste Nachricht zuerst abgearbeitet wird. Nachrichten an einen Aktor können nur über dessen Adresse gesendet werden. Die Adresse eines Aktors identifiziert diesen eindeutig und ändert sich während der gesamten Lebenszeit eines Aktors nicht. Jede Adresse kann beliebig oft kopiert und weitergegeben werden, wodurch andere Aktoren die Möglichkeit erhalten, Nachrichten an den durch die Adresse referenzierten Aktor zu senden.

Grundsätzlich werden zwei Arten von Nachrichten unterstützt:

- Tell: Sendet eine Nachricht ohne Antwortadresse an einen Aktor. Hierbei wird keine Antwort erwartet.
- Ask: Sendet eine Nachricht mit Antwortadresse an einen Aktor. In der Regel wird eine Antwort erwartet.

Für jeden Datentyp in Rust können maximal ein Tell- und ein Ask-Verhalten pro Aktor definiert werden. Tell- und Ask-Nachrichten können, sofern der zu versendende Inhalt kopierbar ist, auch an alle sich in einem Aktor-System befindlichen Aktoren gesendet werden. Hierbei wird für jeden empfangenden Aktor eine Kopie der Originalnachricht erzeugt.

Damit ein Aktor ausgeführt werden kann, muss dieser zu einem Aktor-System hinzugefügt werden. Ein Aktor-System dient als Gruppierung mehrerer Aktoren, die untereinander kommunizieren können. Jeder Aktor muss mit einem eindeutigen, noch nicht verwendeten Namen im Aktor-System registriert werden. Aktoren können die

Adressen von anderen Aktoren im gleichen Aktor-System erhalten, indem diese entweder den Namen der Registrierung des Aktors kennen und im Aktor-System dessen Adresse abfragen, oder indem diese die Adresse über eine Nachricht erhalten.

Im Fehlerfall können Aktoren durch das Aktor-System neu gestartet werden, wobei die Art des Neustarts durch Strategien definiert wird. Jeder Neustartstrategie steht eine Kopie des initialen Zustands und Verhaltens eines Aktors zur Verfügung. Das Postfach, die in dem Postfach vorhandenen Nachrichten und die Adresse des Aktors bleiben bei einem Neustart erhalten.

Durch dieses Design werden alle Aspekte des originalen Aktor-Modells in der Bibliothek abgebildet. In den folgenden Kapiteln werden die Details zu den oben genannten Eigenschaften näher beschrieben.

3.1.2 Einführungsbeispiel

In diesem Kapitel wird ein einfaches Beispiel für die Verwendung der entwickelten Bibliothek vorgestellt, damit die in den folgenden Kapiteln beschriebenen Design-Entscheidungen, Mechanismen und Funktionen greifbarer werden.

In Abb. 3.1 ist ein Einführungsbeispiel zu der entwickelten Bibliothek dargestellt. Die Idee des dargestellten Beispiels ist, dass zwei Aktoren definiert werden, die sich abwechselnd eine Ping- und Pong-Nachricht senden. Weil Aktoren intern als Futures ausgeführt werden, muss die Main-Funktion als `async` markiert und Tokio als Runtime spezifiziert werden. Als Zustand der Aktoren wird das Struct `PingPong` definiert, das keine Daten beinhaltet, da die entsprechenden Aktoren keine Daten speichern müssen. Die Implementation des Traits `Clone` für das Struct `PingPong` wird automatisch durch den Compiler generiert und ist an dieser Stelle nötig, da beide Aktoren das gleiche Verhalten haben sollen und daher das Verhalten `behavior1` kopiert wird. Das Kopieren eines Verhaltens ist nur möglich, wenn der zugehörige Zustand auch kopierbar ist, weshalb `PingPong` `Clone` implementieren muss.

Als Nachricht wird das Enum `Ball` mit zwei Varianten definiert. Das Verhalten `behavior1` beschreibt, wie die Aktoren Ask-Nachrichten vom Typ `Ball` verarbeiten. Hierbei wird zuerst durch Pattern-Matching die Variante von `Ball` bestimmt und anschließend die Variante, die nicht empfangen wurde, an den Absender retourniert.

```rust
#[tokio::main]
async fn main() {
    #[derive(Clone)]
    struct PingPong {}
    enum Ball { Ping, Pong }

    let mut behavior1 = BehaviorBuilder::new()
    .on_ask::<Ball>(|msg, state, sender, ctx| -> BehaviorAction<PingPong> {
        match msg {
            Ball::Ping => {
                sender.ask(Ball::Pong, ctx.get_addr());
            },
            Ball::Pong => {
                sender.ask(Ball::Ping, ctx.get_addr());
            }
        }
        Behavior::keep()
    })
    .build();
    let behavior2 = behavior1.clone();

    let mut actor1 = Actor::new(PingPong {}, behavior1, MailboxType::Unbounded);
    let tx1 = actor1.get_addr();

    let mut actor2 = Actor::new(PingPong {}, behavior2, MailboxType::Unbounded);
    let tx2 = actor2.get_addr();

    let mut actor_sys = ActorSystem::new();
    actor_sys.spawn(actor1,  "a1".to_string());
    actor_sys.spawn(actor2,  "a2".to_string());
    tx1.ask(Ball::Ping, tx2);
    actor_sys.start().await;
}
```

Abbildung 3.1: Beispiel zweier Aktoren, die abwechselnd eine Ping- und Pong-Nachricht aneinander schicken.

Weil sich für dieses Beispiel kein Verhalten während der Laufzeit verändert, wird dies durch die Rückgabe von `Behavior::keep()` signalisiert.

Anschließend werden die beiden Aktoren mit den entsprechenden Verhalten und nicht limitierten Postfächern erzeugt, deren Adressen bestimmt und die Aktoren mit einem Aktor-System assoziiert. Um die Kommunikation zwischen den Aktoren zu initiieren, wird eine Ping-Nachricht an `actor1` mit der Absende-Adresse von `actor2` gesendet. Dies führt dazu, dass abwechselnd Ping- und Pong-Nachrichten zwischen den Aktoren ausgetauscht werden.

Dieses einfache Beispiel dient als Einstieg in die grundlegende Verwendung und Funk-

tionsweise der implementierten Aktor-Bibliothek, verwendet aber dementsprechend nur eine sehr geringe Anzahl der bereitgestellten Funktionen der Bibliothek. Weitere Funktionen und Mechanismen werden in späteren Kapiteln erläutert und in den Code-Beispielen der entwickelten Bibliothek praktisch präsentiert.

3.2 Design-Entscheidungen

In diesem Kapitel werden die wichtigsten Design-Entscheidungen, die im Rahmen der Entwicklung der Aktor-Bibliothek getroffen wurden, erläutert und potenzielle Alternativen diskutiert.

3.2.1 Nebenläufigkeit von Aktoren

Eine der anfangs wichtigsten Entscheidungen für die Implementation der Aktor-Bibliothek war die Wahl, wie Aktoren ausgeführt werden sollen. Aufgrund der parallelen und nebenläufigen Natur des Aktor-Modells war klar, dass ein nebenläufiger Programmieransatz benötigt wird. Wie in Abschnitt 2.1.15 beschrieben werden in Rust zwei Arten von Nebenläufigkeit unterstützt:

- Nebenläufigkeit basierend auf Threads
- Nebenläufigkeit basierend auf Futures

Die direkte Verwendung von Threads, also ein Thread pro Aktor, hätte den Vorteil, dass keine Async-Runtime benötigt werden würde und kein zusätzlicher Overhead aufgrund von Abstraktionen entstehen würde. Des Weiteren würde der Scheduler des Betriebssystems, der in der Regel eine sehr gute Performance aufweist, weil dieser Einsicht in alle Ressourcen des Systems besitzt, für die Verwaltung der Threads und somit auch der Aktoren verwendet werden. Da das Aktor-Modell die Erzeugung vieler Aktoren vorsieht, würde dieser Ansatz zur Erzeugung vieler Threads führen. Jeder Prozessor besitzt eine beschränkte Anzahl an Kernen und unterstützten aktiven Threads, weshalb der Scheduler bei einer großen Anzahl an Aktoren viele Kontextwechsel durchführen müsste, damit jeder Aktor genügend Rechenzeit für die Abarbeitung seiner Nachrichten bekommen würde. Kontextwechsel sind aus Betriebssystemsicht sehr teuer und daher zu vermeiden, weshalb dieser Ansatz verworfen wurde.

Würde jeder Aktor als Future repräsentiert werden, so könnte eine sehr große Anzahl an Aktoren erzeugt werden, da Futures von der Async-Runtime ausgeführt werden und diese in der Regel intern Thread-Pools verwenden, wodurch keine Kontextwechsel auftreten. Ein weiterer Vorteil bei der Verwendung von Futures wäre, dass die Ausführung der Aktoren von der tatsächlichen Hardware entkoppelt wäre, wodurch die Ausführung auf Systemen, auf denen keine Threads existieren, möglich wäre, wenn eine entsprechende Async-Runtime für das System existiert. Da die gesamte Async-Umgebung in Rust durch Traits von der tatsächlichen Runtime-Implementation entkoppelt ist, ermöglicht dies eine weitaus flexiblere Systemarchitektur und Kompatibilität mit Systemen wie beispielsweise Microcontrollern.

Aufgrund der oben genannten Vor- und Nachteile wurde für die Implementation dieser Bibliothek der Ansatz basierend auf Futures gewählt, wodurch eine größere Flexibilität und eine höhere Anzahl an Aktoren möglich ist. Jeder Aktor ruft innerhalb eines Futures seine Nachrichten aus dem Postfach ab und verarbeitet diese sequentiell. Stehen keine Nachrichten in einem Postfach zur Verfügung, so wird dieses Futures bis zum Eintreffen neuer Nachrichten pausiert. Diese Implementation ermöglicht eine effiziente Nutzung der vorhandenen Ressourcen und die Erzeugung und Ausführung einer großen Anzahl an Aktoren.

3.2.2 Nachrichtenverarbeitungsmodus

Gemäß dem Aktor-Modell verarbeitet ein Aktor zu einem Zeitpunkt maximal eine Nachricht. Aufgrund der zuvor beschriebenen Wahl der Repräsentation von Aktoren durch Futures und der Eigenschaft, dass Futures unterbrochen werden können, stellt sich die Frage, ob eine Nachrichtenverarbeitung unterbrochen werden können soll oder nicht.

Ein Vorteil einer möglichen Unterbrechung wäre, dass eine Verarbeitung, die auf ein Ergebnis wartet und daher blockiert ist, nicht die Verarbeitung weiterer Nachrichten blockieren würde. Diese Vorgehensweise würde auch dem Ansatz, der bei der Verwendung von Futures üblich ist, entsprechen, würde aber die Regel, dass ein Aktor zu einem Zeitpunkt maximal eine Nachricht verarbeitet, potenziell verletzen und eine zusätzliche Parallelität einführen, die die Nutzung der Bibliothek komplizierter gestalten würde.

Die Alternative, nämlich dass eine Nachrichtenverarbeitung nicht unterbrochen werden kann, hätte den Nachteil, dass eine blockierende Verarbeitung nicht nur die Ausführung anderer Futures, sondern auch den Thread der Async-Runtime, der das Future ausführt, bis zum Ende der Verarbeitung blockieren würde. Aus diesem Grund müsste sich jeder Nutzende der Bibliothek über die Funktionsweise des Aktor-Modells und den damit verbundenen Implikationen bewusst sein, um das Blockieren von Ressourcen zu vermeiden.

Weil die Bibliothek das Aktor-Modell möglichst genau repräsentieren soll, wurde entschieden, dass eine Nachrichtenverarbeitung nicht unterbrochen werden kann und gegenüber dem Nutzenden der Bibliothek als synchrone Funktion erscheint, auch wenn diese intern als Future ausgeführt wird. Dies hat einerseits den Vorteil, dass Nutzende der Bibliothek keine Kenntnis über die Verwendung von Futures besitzen müssen, andererseits entspricht es der Idee des Aktor-Modells, dass Nachrichten sequentiell abgearbeitet werden, wodurch auch die Funktionsweise von Aktoren leichter verstanden werden kann.

3.2.3 Dynamische Typisierung von Nachrichten und Verhalten

Wie in Abschnitt 2.3 beschrieben basieren existierende Rust-Bibliotheken auf der Verwendung statischer Nachrichtentypen und Verarbeitungsaktionen aufgrund der Verwendung generischer Datentypen. Dieser Ansatz garantiert typsichere Programme und eine Optimierung des Codes durch den Compiler, da alle generischen Typen und Funktionen monomorphisiert werden können. Aufgrund dieser Tatsache führt die Verwendung generischer Datentypen im Allgemeinen zu einer höheren Performance als die Verwendung von Trait Objects, da bei Trait Objects eine zusätzliche Indirektion zur Laufzeit aufgrund des Virtual Method Tables auftritt und diese Indirektionen durch den Compiler nicht optimiert werden können. Ein Nachteil der Verwendung generischer Datentyp ist, dass die Typen zur Compilezeit fixiert werden und sich daher zur Laufzeit nicht ändern können, wodurch Funktionalitäten wie beispielsweise die Änderung des Verhaltens eines Aktors zur Laufzeit mit diesem Ansatz nicht möglich sind. Weil das Aktor-Modell Dynamiken dieser Art, wie in Agha (1985, S. 12ff) beschrieben, vorsieht, wurden die oben genannten Performanceeinbußen zugunsten der Entwicklung einer flexibleren Bibliothek akzeptiert und die Bibliothek größtenteils basierend auf Trait Objects implementiert. Aufgrund dieser Designentscheidung kann sich das Verhalten eines Aktors bei jeder Verarbeitung einer Nachricht ändern, wodurch

dynamische Aktoren direkt mithilfe der Bibliothek repräsentiert werden können. Des Weiteren kann durch dieses Design jeder Datentyp, der innerhalb einer Nachricht gespeichert werden kann, an jeden Aktor gesendet werden, wobei die entsprechende Verarbeitungsaktion dynamisch durch das Verhalten definiert wird. Die Anforderungen, die ein Datentyp erfüllen muss, damit dieser als Nachricht verwendet werden kann, werden in Abschnitt 3.3.3.3 näher erläutert.

3.2.4 Warteschlangen-Modell des Postfachs

Wie in Abschnitt 3.1 beschrieben besitzt jeder Aktor ein Postfach, in dem die abzuarbeitenden Nachrichten zwischengespeichert werden. Für die Implementation eines Postfachs stehen grundsätzlich zwei verschiedene Arten von Warteschlangen zur Verfügung:

- Unbounded Queue: Warteschlange ohne maximale Kapazität
- Bounded Queue: Warteschlange mit fest definierter Kapazität

Unbounded Queues haben keine Begrenzung für die Anzahl der gespeicherten Nachrichten. Sie besitzen den Nachteil, dass bei einer höheren Empfangsrate als Abarbeitungsrate des Aktors beliebig viel Speicher des Systems für die Zwischenspeicherung der Nachrichten konsumiert werden kann, wodurch das Gesamtsystem durch einen einzelnen Aktor beeinflusst werden kann. Im Gegenzug wird somit sichergestellt, dass wenn eine Nachricht eine Warteschlange erreicht, diese die Nachricht auch sicher speichert und nicht verwirft.

Bounded Queues haben eine begrenzte Anzahl an Nachrichten, die diese speichern können. Nach der Erreichung des Maximums der Speicherkapazität wird der Speicher aller neu empfangenen Nachrichten direkt verworfen und freigegeben. Einzelne Aktoren können das Gesamtsystem aufgrund dieses Mechanismus nur innerhalb bestimmter, fest definierter Grenzen beeinflussen. Das durch dieses Verhalten potenziell auftretende Szenario, dass Nachrichten von einem Aktor nicht erhalten werden können, weil dessen Warteschlange voll ist, kann durch die Definition von Protokollen umgangen werden. Beispielsweise kann ein Protokoll die Bestätigung des Empfangs jeder Nachricht erfordern, wodurch verlorene Nachrichten erneut gesendet werden können. Das Aktor-Modell sieht für Szenarien dieser Art keine Abstraktionen vor, da Nachrichten ausschließlich nach dem Best-Effort-Prinzip zugestellt werden. Ein entsprechendes

Protokoll ist somit durch den Nutzenden der Bibliothek zu definieren, sofern die Charakteristiken des Best-Effort-Prinzips nicht gewünscht sind.

Da die Entscheidung zwischen einer Bounded und Unbounded Queue stark vom jeweiligen Anwendungsfall abhängt, wurden im Rahmen dieser Bibliothek beide Optionen implementiert, sodass Nutzende der Bibliothek die von ihnen präferierte Warteschlange für jeden Aktor definieren können. Die Implementationsdetails hierzu werden in Abschnitt 3.3.3.2 näher beschrieben.

3.2.5 Verarbeitung nicht unterstützter Nachrichten

In diesem Kapitel wird die in Abschnitt 3.1 beschriebene Design-Entscheidung, dass Nachrichten, deren Verarbeitung durch den empfangenden Aktor nicht möglich ist, verworfen werden, diskutiert. Die Grundfrage, die sich stellt, ist, was mit Nachrichten geschehen soll, die von einem Aktor nicht verarbeitet werden können. Für diese Problemstellung sind grundsätzlich verschiedene Lösungsansätze denkbar. Die in der Praxis am häufigsten verwendeten Ansätze sind:

- Aufbewahrung der Nachricht
- Weiterleitung der Nachricht
- Löschen der Nachricht

Die Idee hinter der Aufbewahrung einer aktuell nicht verarbeitbaren Nachricht besteht darin, dass ein eventuell zukünftiges, neues Verhalten eines Aktors die Verarbeitung der aufbewahrten Nachricht ermöglicht. Ein Nachteil dieser Zwischenspeicherung ist, dass jeder Aktor Nachrichten speichert, deren Verarbeitung möglicherweise nie geschieht. Dieser Nachteil ist vor allem in Kombination mit Broadcast-Nachrichten problematisch, da diese in der Regel nur für spezifische Aktoren relevant sind und trotzdem durch alle anderen Aktoren gespeichert werden würden. Eine mögliche Lösung für dieses Problem wäre die manuelle Implementation einer leeren Verarbeitungsaktion für jeden Aktor, der die Nachricht nicht empfangen soll, wobei dies die Kenntnis aller möglichen Broadcast-Nachrichtentypen erfordern würde.

Bei dem zweiten Ansatz, nämlich der Weiterleitung von nicht unterstützten Nachrichten, stellt sich die Frage, wohin diese weitergeleitet werden sollen. Beispielsweise könnten diese an das Aktor-System weitergeleitet werden. Dieses Verhalten kann in

Kombination mit Broadcast-Nachrichten problematisch sein, weil eine mitunter sehr große Anzahl an Nachrichten an das Aktor-System weitergeleitet wird, wodurch dieses überlastet werden kann.

Der letzte Ansatz, nämlich das Löschen von nicht unterstützten Nachrichten, löscht alle Nachrichten, die zum Zeitpunkt der Verarbeitung durch das aktuelle Verhalten des Aktors nicht verarbeitet werden können. Dieser Mechanismus besitzt den Nachteil, dass Nachrichten, die aktuell nicht unterstützt werden, gelöscht werden und auch in Zukunft nicht mehr zur Verfügung stehen für eine eventuelle Verarbeitung durch ein neues Aktor-Verhalten. Dieser Nachteil hat aus performancetechnischer Sicht den Vorteil, dass Nachrichten nur bis zu deren Verarbeitung gespeichert werden. Ist eine Verarbeitung nicht möglich, so wird die entsprechende Nachricht und deren Speicher freigegeben.

Im Rahmen der Entwicklung dieser Bibliothek wurde der letzte Ansatz, nämlich das Löschen von nicht unterstützten Nachrichten, gewählt, da dieser eine bessere Ressourcennutzung ermöglicht und generisch genug ist, um alle möglichen Anwendungsfälle implementieren zu können. Funktionalitäten, die durch die anderen Ansätze möglich wären, können durch die gezielte Definition von Protokollen und Nachrichtentypen kompensiert werden.

3.3 Implementation

In diesem Kapitel wird die Implementation der entwickelten Aktor-Bibliothek beschrieben. Einführend wird erläutert, wie die Dokumentation der Bibliothek durchgeführt wurde, bevor anschließend die verwendeten existierenden Rust-Bibliotheken vorgestellt werden. Danach werden die Implementationen der wichtigsten Komponenten der Aktor-Bibliothek beschrieben und mit Code-Beispielen erklärt. Abschließend wird die Implementation des Test-Frameworks und dessen Funktionen vorgestellt.

Der Code und die Dokumentation der entwickelten Bibliothek sind unter `https://crates.io/crates/aector` öffentlich einsehbar.

3.3.1 Dokumentation

Für die Dokumentation der entwickelten Bibliothek wurde das in Rust integrierte Werkzeug cargo-doc verwendet, das für alle Bibliotheken in Rust verwendet wird. cargo-doc generiert, basierend auf sogenannten Doc Comments, die in den Code integriert werden, eine HTML-Seite, in der alle öffentlichen Code-Schnittstellen inklusive Verlinkungen und Beschreibungen visualisiert sind. In Doc Comments können auch Beispiel-Codes und Tests integriert werden, die durch den Compiler geprüft und in der generierten Dokumentation inklusive Syntax Highlighting dargestellt werden. Ein sehr großer Vorteil einer Dokumentation dieser Art ist, dass diese fest mit dem Code gekoppelt ist, somit auch versioniert gespeichert wird und aufgrund der Lokalität bei Code-Änderungen besser angepasst werden kann. Der Entwickelnde muss sich hierbei nicht um eine Formatierung kümmern, da diese automatisch mittels des oben genannten Werkzeugs generiert wird, und kann sich vollständig auf logische Verlinkungen und den Inhalt fokussieren.

3.3.2 Verwendete Rust-Bibliotheken

In diesem Kapitel werden einige ausgewählte Bibliotheken, die für die Entwicklung der Bibliothek verwendet wurden, beschrieben und deren Funktionalitäten erläutert.

3.3.2.1 Anyhow und Thiserror

Für das Fehlermanagement innerhalb der entwickelten Bibliothek werden die Bibliotheken Anyhow und Thiserror verwendet. Thiserror ermöglicht eine einfache Implementation und Definition von eigenen Fehlertypen mittels automatischer Code-Generierung basierend auf Attributen. Jeder Fehlertyp in Rust muss das `Error`-Trait und die zugehörigen Supertraits `Debug` und `Display` implementieren. Mittels der Code-Generierung von Thiserror werden diese Implementationen automatisch erzeugt. In Abb. 3.2 ist ein Beispiel für die Verwendung von Thiserror dargestellt. Als Vergleich ist die manuelle Implementation desselben Fehlertyps in Abb. 3.3 abgebildet.

Die Bibliothek Anyhow bietet eine Abstraktion über das `Error`-Trait, die erlaubt, verschiedene Fehlertypen von Funktionen zurückzugeben. Dies wird erreicht, indem der konkrete Fehlertyp in ein Trait Object umgewandelt wird, das zusätzliche Funktionen für Backtracing und Downcasting ermöglicht. Diese Abstraktion bietet ein besseres

```
#[derive(Error, Debug)]
enum CustomError {
    #[error("Error1")]
    Error1,
    #[error("Error 2 with data `{0}`")]
    Error2(String),
}
```

Abbildung 3.2: Durch die Verwendung der Thiserror-Attribute wird `Error`, `Display` und `Debug` automatisch implementiert.

Fehlermanagement als die manuelle Verwaltung und Erzeugung von Trait Objects an Stellen, an denen verschiedene Fehlertypen auftreten können. Da der tatsächliche Typ der Fehler eliminiert wird, wird dieser Ansatz nur verwendet, wenn die potenziellen Fehler nicht vermeidbarer Natur sind oder die Antwort auf alle Fehler identisch ist. Weiters kann das Typsystem von Rust bei der Verwendung dieser Bibliothek nicht überprüfen, ob alle möglichen Fehler abgehandelt werden, weshalb hier mit besonderer Vorsicht vorgegangen werden muss.

3.3.2.2 Tokio

Tokio ist, gemäß Lerche (2022), eine Async-Runtime für Rust, die zusätzliche Funktionen für die Entwicklung von asynchronen Anwendungen, wie beispielsweise Netzwerk- und Datei-Schnittstellen, bietet. Ein weiterer wichtiger Aspekt von Tokio ist, dass die Bibliothek performante Datentypen, wie beispielsweise Multi-Producer, Single-Consumer (MPSC)-Warteschlangen, bereitstellt. Alternative häufig verwendete Async-Runtimes wären Async-Std und Smol. Im Rahmen der Entwicklung dieser Bibliothek wurde Tokio verwendet, da diese die zum Zeitpunkt der Entwicklung am besten dokumentierte und am längsten erprobte Async-Runtime für Rust ist und effizientere Implementationen der für die Entwicklung dieser Bibliothek benötigten Datentypen bereitstellt.

3.3.3 Aktor

In der entwickelten Bibliothek wird ein Aktor durch zwei initiale Eigenschaften beschrieben:

```rust
enum CustomError { Error1, Error2(String) }

impl Error for CustomError {}
impl Debug for CustomError {
    fn fmt(&self, f: &mut Formatter<'_>) -> std::fmt::Result {
        match self {
            CustomError::Error1 => { write!(f, "Error1") },
            CustomError::Error2(data) => {
                write!(f, "Error 2({})", data)
            }
        }
    }
}
impl Display for CustomError {
    fn fmt(&self, f: &mut Formatter<'_>) -> std::fmt::Result {
        match self {
            CustomError::Error1 => { write!(f, "Error1") },
            CustomError::Error2(data) => {
                write!(f, "Error 2 with data {}", data)
            }
        }
    }
}
```

Abbildung 3.3: Manuelle Implementation des in Abb. 3.2 dargestellten Fehlertyps.

- Der Wert eines Typs, der den Zustand des Aktors repräsentiert.

- Ein Verhalten, das definiert, wie der Aktor Nachrichten bestimmter Typen abarbeitet und wie sich der Aktor bei Systemereignissen verhält.

Des Weiteren erhält jeder Aktor automatisch ein Postfach, in dem die empfangenen Nachrichten für die Weiterverarbeitung gespeichert werden, und eine Adresse, über die dem Aktor Nachrichten gesendet werden können.

In den folgenden Kapiteln werden die genannten Aspekte von Aktoren und zugehörige Hilfsdatentypen detaillierter erläutert und deren konkrete Implementationen beschrieben.

3.3.3.1 Zustand

Der Zustand eines Aktors wird durch den Wert eines Datentyps repräsentiert, der sich im Besitz des Aktors befindet. Ein Beispiel für die Erzeugung eines Aktors ist in Abb. 3.4 dargestellt. Als Zustand eines Aktors kann jeder Datentyp in Rust verwendet werden, der `Send + 'static` implementiert. Alle Typen, die sicher von einem Thread zu einem anderen Thread gesendet werden können, werden von dem Compiler automatisch mit dem `Send`-Trait markiert. Die Lifetime `'static` bedeutet in diesem Zusammenhang, dass der Typ des Zustands und alle zugehörigen Daten für die gesamte Laufzeit des Programms gültig sein müssen. Diese Einschränkung der Lifetime ermöglicht, dass ein Aktor seinen eigenen Zustand beliebig lange besitzen kann, ohne dass ein Zugriff auf einen ungültigen Speicher entstehen kann.

Der Zustand eines Aktors ist vollständig isoliert gegenüber externen Zugriffen und kann nur über das Verhalten des jeweiligen Aktors gelesen und bearbeitet werden. Der Datentyp des Zustandes eines Aktors ist nach der Erzeugung eines Aktors fixiert und kann sich während der Laufzeit nicht ändern.

Da ein Aktor zu einem bestimmten Zeitpunkt maximal eine Nachricht abarbeiten kann und der Zustand gegenüber externen Lese- und Schreibzugriffen isoliert ist, werden keine weiteren Synchronisierungsmaßnahmen für den Schutz gegen gleichzeitige Lese- und Schreibzugriffe benötigt.

```
struct ExampleState {
    name: String,
    flag: bool
}

let init_state = ExampleState {
    name: "hello".to_string(),
    flag: false
};
let behavior = BehaviorBuilder::new().build();
let actor = Actor::new(init_state, behavior, MailboxType::Unbounded);
```

Abbildung 3.4: Der Datentyp `ExampleState` repräsentiert den Zustand des erzeugten Aktors, wobei der Wert `init_state` dem initialen Zustand des Aktors entspricht.

3.3.3.2 Postfach

Gemäß dem Aktor-Modell besitzt jeder Aktor ein Postfach, das die empfangenen Nachrichten für die weitere Verarbeitung durch den Aktor in einer Warteschlange speichert. Weil mehrere Aktoren gleichzeitig Nachrichten an dasselbe Postfach senden können und der Aktor des Postfachs zum gleichen Zeitpunkt Nachrichten abarbeiten kann, erfordert die Implementation des Postfachs Synchronisierungsmaßnahmen, um die Datenintegrität dessen sicherstellen zu können.

Das in dieser Bibliothek implementierte Postfach basiert auf den von Tokio bereitgestellten Datentypen der Bounded und Unbounded MPSC-Warteschlangen, wobei sich der Nutzende der Bibliothek bei der Erzeugung jedes Aktors für eine der beiden Implementationen entscheiden kann. Bounded Queues sind Warteschlangen, die eine zur Compile-Zeit definierte maximale Kapazität aufweisen. Nachrichten, die nach dem Erreichen der maximalen Kapazität einer Bounded Queue eintreffen, werden automatisch gelöscht. Im Gegensatz hierzu stehen Unbounded Queues, die keine fixierte Kapazität aufweisen und nur durch den vorhandenen Arbeitsspeicher des ausführenden Systems limitiert sind. Mögliche Vor- und Nachteile von Bounded und Unbounded Queues sind in Abschnitt 3.2.4 beschrieben.

Alle Zugriffe auf eine MPSC-Warteschlange werden durch zwei verschiedene Datentypen repräsentiert:

- `Receiver<T>`/`UnboundedReceiver<T>`: Bietet die für das Empfangen von Nachrichten benötigten Funktionen an.
- `Sender<T>`/`UnboundedSender<T>`: Ermöglicht das Senden von Nachrichten an die Warteschlange.

Der generische Parameter `<T>` bezieht sich hierbei auf den Datentyp, dessen Werte in der Warteschlange gespeichert werden können. Wie der Name MPSC-Warteschlange bereits vermuten lässt, kann pro MPSC-Warteschlange nur eine empfangende Komponente, aber beliebig viele sendende Komponenten, existieren.

Um die verschiedenen Funktionsweisen von Bounded und Unbounded Warteschlangen abstrahieren zu können, wurde der Datentyp `Mailbox` implementiert. Dieser bietet nach außen hin eine neutrale Schnittstelle, die das Abarbeiten von Nachrichten aus der Warteschlange, unabhängig von deren tatsächlichen Implementierung, durch den Aktor ermöglicht.

Für die Abstraktion über den sendenden Teil wurde der Typ `Address` implementiert, dessen Details näher in Abschnitt 3.3.3.4 beschrieben werden.

Da, wie oben beschrieben, eine MPSC-Warteschlange generisch über einen spezifischen Typ `T` ist, könnte ein Aktor nur eine Art von Nachricht empfangen. Um diese Einschränkung umgehen zu können, wird jede Nachricht in eine `Message` verpackt und anschließend in der MPSC-Warteschlange gespeichert, wodurch beliebige viele, verschiedene Arten von Nachrichten von einem Aktor empfangen und verarbeitet werden können. Die Funktionsweise dieses speziellen Datentyps wird in Abschnitt 3.3.3.3 näher beschrieben.

3.3.3.3 Nachrichten

Ein Aktor muss verschiedene Arten und Typen von Nachrichten empfangen und verarbeiten können. Das Grundproblem bei der Implementation dieser Idee besteht darin, dass eine MPSC-Warteschlange ein generischer Datentyp ist und daher nur Elemente eines konkreten Datentyps speichern kann. Wie in Abschnitt 2.1.14 beschrieben können Trait Objects Problemstellungen dieser Art lösen, allerdings ermöglichen Trait Objects nur den Zugriff auf Funktionen, die in den entsprechenden Traits beschrieben sind. Da kein Trait für die Funktionen aller möglichen Nachrichten erzeugt werden kann, genügt dieser Ansatz nicht, um das gesamte Spektrum an Nachrichtentypen abdecken zu können. Trait Objects, deren Traits dem Any-Trait entsprechen, können, wie in Abschnitt 2.1.17 beschrieben, zurück zu ihrem tatsächlichen Typ konvertiert werden. Mit dem konvertierten Wert kann anschließend normal weitergearbeitet werden, als ob dieser nie ein Trait Object gewesen wäre. Das bedeutet, dass nach der Konvertierung ein voller, typsicherer Zugriff auf alle Funktionen des konkreten Werts und dessen Daten möglich ist.

Durch den Datentyp `Message` werden die oben genannten Konvertierungen typsicher, transparent und automatisch durchgeführt. Benutzende der Bibliothek kommen nie in direkten Kontakt mit dem `Message`-Typ und können ihre Aktoren und Verarbeitungsaktionen so definieren, als ob diese auf zur Compile-Zeit fixierten Typen arbeiten würden, wodurch nach außen hin die Bibliothek statisch typisiert erscheint. Alle Konvertierungen der Verarbeitungsaktionen und Nachrichten, die auf dynamischer Typisierung mittels dem Any-Trait basieren, werden intern auf eine sichere Art und Weise durchgeführt.

Alle Nachrichten, die von oder an einen Aktor gesendet werden, werden automatisch in einer `Message` transportiert, wobei der Datentyp einer Nachricht, die gekapselt werden soll, lediglich das `Any`- und `Send`-Trait implementieren muss, damit dieser die von `Message` bereitgestellten Funktionen nutzen kann. Das `Any`-Trait wird, wie in Abschnitt 2.1.17 beschrieben, von den meisten Datentypen in Rust implementiert.

Das `Send`-Trait ist ein Marker-Trait, also ein Trait, das keine tatsächliche Funktion bietet und das automatisch vom Compiler für Typen vergeben wird, deren Übergabe von einem Thread zu einem anderen Thread sicher ist.

```
pub struct Message {
    inner: Box<dyn Any + Send>,
    pub(crate) sender: Option<Addr>
}
```

Abbildung 3.5: Der Typ Message besitzt die zu übertragende Nachricht in `inner` und eine optionale Antwortadresse. `pub(crate)` bedeutet hierbei, dass auf dieses Feld nur von innerhalb der Bibliothek zugegriffen werden kann.

Weil der Datentyp `Message` den zu sendenden Wert besitzt, wird beim Senden einer Nachricht der Besitz des Wertes ebenfalls an den empfangenden Aktor mittels eines Moves übertragen. Weil ein Move in Rust einen sehr geringen rechnerischen Aufwand erfordert, ist diese Operation in Kombination mit dem Ownership-Modell effizient umsetzbar. Kann eine Nachricht nicht zugestellt oder verarbeitet werden, so obliegt dem Owner der `Message` die Speicherverwaltung und dieser kann diese, sofern nicht mehr benötigt, freigeben.

Aufgrund der Implementation von `Message` können fast alle Datentypen als Nachrichten zwischen Aktoren verwendet werden. Weil `Message` ein nicht-generischer Datentyp ist und als Abstraktion über Datentypen, die als Nachricht gesendet werden, verwendet wird, können Aktoren innerhalb ihres Postfachs verschiedenste Datentypen von Nachrichten speichern und verarbeiten, wodurch den Nutzenden der Bibliothek eine zusätzliche Flexibilität ermöglicht wird. Des Weiteren ist die so erreichte Entkopplung des generischen Postfachs, das nur Werte eines Datentyps speichern kann, von `Message`, das beliebige Datentypen beinhalten kann, essenziell für die in Abschnitt 3.3.3.5 beschriebene Dynamik, dass Aktoren während ihrer Laufzeit ihr Verhalten ändern können.

3.3.3.4 Adresse

Das Aktor-Modell sieht vor, dass Nachrichten an einen Aktor nur über dessen Adresse gesendet werden können. Dieses Konzept wird in der entwickelten Bibliothek durch den Datentyp `Address` realisiert. `Address` verwendet intern die in Abschnitt 3.3.3.2 beschriebenen Datentypen `Sender<T>` und `UnboundedSender<T>`, die für das Senden von Nachrichten an eine MPSC-Warteschlange, abhängig von dessen Implementation, zuständig sind. Des Weiteren kapselt `Address` alle technischen Implementationsdetails des Sendemechanismus und besitzt nach außen hin ausschließlich die Funktionen `ask` und `tell` für das Senden von Nachrichten. Das Senden von Nachrichten über eine Adresse ist in Abb. 3.6 dargestellt.

Wird `ask` oder `tell` mit einer zu sendenden Nachricht aufgerufen, so wird diese, sofern es die Kapazität des Postfachs hinter der Adresse erlaubt, am Ende der Warteschlange des empfangenden Postfachs eingefügt. Intern wird, wie in Abschnitt 3.3.3.3 beschrieben, der Wert zuerst innerhalb einer `Message` als Trait Object gekapselt und anschließend dem Postfach hinzugefügt. Die Sende-Operationen sind in allen möglichen Anwendungsfällen sicher, da die Abstraktion garantiert, dass keine Synchronisierungsprobleme im Rahmen des Sendeprozesses auftreten können, auch wenn sich der sendende und empfangende Aktor in verschiedenen Threads befinden und mehrere Aktoren gleichzeitig Nachrichten senden. Des Weiteren stellt der Typ `Address` sicher, dass durch das Senden von Nachrichten an einen nicht mehr aktiven Aktor keine Fehler auftreten, sondern der Speicher der gesendeten Nachrichten automatisch freigegeben wird. Diese Vorgehensweise ist mit dem Best-Effort-Zustellungsprinzip des Aktor-Modells vereinbar und verhindert die längerfristige Allokation von Speicher für Nachrichten, die von keinem Aktor abgearbeitet werden können.

Jede Adresse eines Aktors kann beliebig oft kopiert und geteilt werden, weshalb jeder Aktor gegenüber anderen Aktoren durch diesen Datentyp repräsentiert und identifiziert wird. Aufgrund der oben beschriebenen implementierten Synchronisierungsmaßnahmen können kopierte Adressen auch an andere Threads übergeben und von dort aus verwendet werden. Das Senden einer Nachricht über eine `Address` ist die einzige Möglichkeit innerhalb dieser Bibliothek, mit einem Aktor interagieren zu können.

```
let actor = Actor::new((), behavior, MailboxType::Unbounded);
let tx = actor.get_addr();
tx.tell("message 1".to_string());
tx.ask("message 2".to_string(), tx.clone());
```

Abbildung 3.6: Über die Adresse eines Aktors können Tell- und Ask-Nachrichten an diesen gesendet werden. Bei einer Ask-Nachricht muss zusätzlich eine Antwort-Adresse übergeben werden.

3.3.3.5 Verhalten

Jeder Aktor besitzt zu einem bestimmten Zeitpunkt genau ein Verhalten, das einerseits definiert, wie sich der Aktor bei bestimmten Ereignissen verhält, und andererseits, was für Arten und Typen von Nachrichten von dem Aktor wie zu verarbeiten sind. Beispiele für mögliche Ereignisse, auf die ein Aktor reagieren kann, sind das Starten eines Aktors, das Neustarten eines Aktors oder auch das Auftreten eines Fehlers während der Verarbeitung einer Nachricht. Unter Arten von Nachrichten ist die in Abschnitt 3.1 beschriebene Unterscheidung von Ask- und Tell-Nachrichten zu verstehen. Der Typ einer Nachricht entspricht dem Datentyp der Nachricht in Rust.

Das Verhalten eines Aktors wird durch den Datentyp `Behavior` repräsentiert. Die für die Reaktion auf Ereignisse benötigten Aktionen werden als Closures in `Behavior` gespeichert und bei dem Auftreten eines Ereignisses automatisch aufgerufen. Als Reaktion auf ein Ereignis steht jeder Aktion eine Exclusive Reference auf den Zustand des Aktors und eine Shared Reference auf den `ActorContext` des Aktors zur Verfügung. Die Exclusive Reference ermöglicht Lese- und Schreibzugriffe auf den Zustand des Aktors. Der Datentyp `ActorContext` ermöglicht die Interaktion mit dem Kontext des Aktors. Der Kontext eines Aktors beinhaltet alle von einem Aktor-System bereitgestellten Funktionen wie beispielsweise das Erzeugen neuer Aktoren und Funktionen für die Verwaltung des Aktors selbst, wie zum Beispiel das Stoppen oder Neustarten des Aktors.

Für jede Kombination von Art und Typ einer Nachricht kann genau eine Verarbeitungsaktion pro Verhalten definiert werden. Im Falle einer Ask-Nachricht stehen der Verarbeitungsaktion folgende Ressourcen zur Verfügung:

- Die gesendete Nachricht.
- Eine Exclusive Reference auf den Zustand des Aktors.

- Eine Shared Reference auf den Kontext des Aktors.
- Die Adresse, an die eine Antwort gesendet werden soll.

Für eine Tell-Nachricht stehen alle oben genannten Ressourcen außer der Antwortadresse zur Verfügung, da im Rahmen einer Tell-Nachricht keine Antwort erwartet wird. Ein Beispiel für eine Verarbeitungsaktion einer Ask-Nachricht ist in Abb. 3.7 dargestellt. Weil alle Verarbeitungsaktionen mittels Closures definiert werden, müssen die tatsächlichen Typen der Parameter nicht explizit angegeben werden.

```
let mut behavior = BehaviorBuilder::new()
.on_ask::<String>(|msg,state,reply_to,ctx| -> BehaviorAction<State> {
    Behavior::keep()
)})
.build();
```

Abbildung 3.7: Während der Verarbeitung einer Ask-Nachricht hat ein Aktor Zugriff auf die empfangene Nachricht, seinen Zustand, die Absenderadresse und seinen Aktor-Kontext.

Jede Verarbeitungsaktion besitzt als Rückgabewert ein optionales neues Verhalten, das durch den Datentyp `BehaviorAction<State>` repräsentiert wird. Mittels `Behavior::keep()` als Rückgabewert wird signalisiert, dass das derzeitige Verhalten des Aktors weiterhin verwendet werden soll.

Die Rückgabe von `Behavior::change(behavior)` mit einem neuen Verhalten `behavior` ersetzt das bisherige Verhalten des Aktors durch das neu übergebene Verhalten.

Dieser Mechanismus ermöglicht, dass ein Aktor bei jeder verarbeiteten Nachricht sein Verhalten ändern kann. Eine Dynamik dieser Art entspricht der in Agha (1985, S. 24ff) beschriebenen Funktionsweise von Aktoren. Ein Beispiel für das Ersetzen des Verhaltens innerhalb einer Verarbeitungsaktion ist in Abb. 3.8 dargestellt.

Weil jede Verarbeitungsaktion Zugriff auf ihren Aktor-Kontext, und somit auch auf Funktionen des Aktor-Systems, hat, kann ein Aktor weitere neue Aktoren erzeugen. Ein Beispiel hierfür ist in Abb. 3.9 dargestellt.

Jede Verarbeitungsaktion wird innerhalb des Verhaltens in einer Hashmap gespeichert, wobei pro Nachrichtenart eine Hashmap verwendet wird. Diese Auftrennung in zwei Hashmaps wird benötigt, da eine Hashmap in Rust ein generischer Datentyp ist, der

```
let mut behavior = BehaviorBuilder::new()
    .on_tell::<String>(|msg, state, ctx| -> BehaviorAction<()> {
        // Das bisherige Verhalten wird durch ein neues,
        // leeres Verhalten ersetzt.
        Behavior::change(BehaviorBuilder::new()
            .build()
        )
    })
    .build();
```

Abbildung 3.8: Das Verhalten eines Aktors kann sich am Ende jeder Nachrichtenverarbeitung ändern.

```
let mut behavior1 = BehaviorBuilder::new()
.on_tell::<String>(|msg,state,sender,ctx| -> BehaviorAction<State> {
    let new_actor = Actor::new(
        "new".to_string(),
        BehaviorBuilder::new().build(),
        MailboxType::Unbounded
    );
    ctx.spawn(new_actor, "new actor".to_string());
    Behavior::keep()
})
.build();
```

Abbildung 3.9: Durch den Aktor-Kontext kann ein Aktor während der Verarbeitung einer Nachricht neue Aktoren mit dem Aktor-System assoziieren.

Werte von nur einem Typ speichern kann. Da sich der Datentyp der Verarbeitungsaktionen für Ask- und Tell-Nachrichten unterscheidet, ist eine gemeinsame Speicherung in einer Hashmap nicht ohne zusätzlichen Aufwand möglich. Weil grundsätzlich nur zwei verschiedene Arten von Nachrichten in der Bibliothek unterstützt werden, wurde entschieden, dass die manuelle Trennung in zwei Hashmaps für den vorhandenen Anwendungsfall die bessere Wahl darstellt. Als Schlüssel für die Speicherung einer Aktion in der Hashmap wird die in Abschnitt 2.1.17 beschriebene `TypeId` des zu verarbeitenden Nachrichtentyps verwendet, die für jeden Datentyp in Rust eindeutig ist. Weil zu einem bestimmten Zeitpunkt ein Aktor genau eine Nachricht abarbeiten kann, ist die Verwendung einer normalen Hashmap ohne weitere Synchronisierungsmaßnahmen ausreichend.

Die Erzeugung eines `Behavior` ist nur durch die Verwendung eines `BehaviorBuilder` möglich. Der Typ `BehaviorBuilder` implementiert das Builder-Pattern und überprüft die oben genannten Einschränkungen wie beispielsweise, dass maximal eine Verarbeitungsaktion pro Typ und Art einer Nachricht definiert werden kann. Aufgrund dieser Designentscheidung können neue Verhalten einfach und intuitiv durch die Nutzenden der Bibliothek erzeugt werden, wobei potenzielle Fehler bereits bei der Erzeugung eines Verhaltens detektiert werden können. Ein Beispiel für die Erzeugung eines Verhaltens ist in Abb. 3.10 dargestellt.

```
let mut behavior = BehaviorBuilder::new()
    .on_start(|state, ctx| {
        println!("Actor gestartet!");
    })
    .on_tell::<String>(|msg, state, ctx| -> BehaviorAction<()> {
        println!("Tell-Nachricht mit String erhalten: {}", msg);
        Behavior::keep()
    })
    .build();
```

Abbildung 3.10: Ein Verhalten kann nur über den entsprechenden Builder erzeugt werden.

Die Definition einer Verarbeitungsaktion ist, wie in Abb. 3.10 ersichtlich, typsicher, das heißt, dass innerhalb einer Verarbeitungsaktion der tatsächliche Typ einer Nachricht bekannt ist und alle damit assoziierten Funktionen verwendet werden können. Wie bereits in Abschnitt 3.3.3.2 beschrieben, wird jede Nachricht in einem Postfach intern in einer `Message` gespeichert. Bevor die durch den Entwickelnden definierte Verarbeitungsaktion für eine Nachricht aus dem Postfach aufgerufen wird, wird die `Message` automatisch auf ihren tatsächlichen beinhalteten Typ reduziert. Diese Konvertierung ist aufgrund der in Abschnitt 3.3.3.3 beschriebenen Implementation von `Message` möglich und bietet eine zum Entwickelnden hin typsichere Schnittstelle, die keine manuellen Konvertierungen von Datentypen erfordert.

Damit Verarbeitungsaktionen für verschiedene Arten von Nachrichten in den Hashmaps des Verhaltens gespeichert werden können, werden die durch den Entwickelnden definierten, typsicheren Verarbeitungsaktionen intern als Closures basierend auf `Message` anstatt den tatsächlichen Nachrichtentypen gespeichert. Alle Umwandlungen, die im Rahmen dieses Mechanismus durchgeführt werden, sind aufgrund der Implementation

der Bibliothek sicher und nach außen hin nicht sichtbar.

Wird eine Nachricht von einem Aktor abgearbeitet, so werden folgende Schritte durchlaufen:

- `Message` wird aus dem Postfach entfernt.
- `TypeId` der Nachricht innerhalb der `Message` wird bestimmt.
- Art der `Message` wird bestimmt.
- Verarbeitungsaktion wird mit der Nachricht, die in der `Message` beinhaltet ist, aufgerufen.
- Wenn ein neues Verhalten am Ende der Verarbeitungsaktion zurückgegeben wird, ersetzt dieses das alte Verhalten des Aktors.

Wird im Rahmen der Verarbeitung einer Nachricht keine passende Verarbeitungsaktion gefunden, so bedeutet dies, dass der Aktor die empfangene Nachricht nicht unterstützt. Nachrichten dieser Art werden ignoriert und der zugehörige Speicher automatisch freigegeben. Die Diskussion, ob Nachrichten, die zum Zeitpunkt des Eintreffens mit dem aktuellen Verhalten nicht abgearbeitet werden können, zwischengespeichert werden sollen oder nicht, wird in Abschnitt 3.2.5 geführt.

Aufgrund der strukturellen Trennung des Verhaltens eines Aktors von dessen Zustand ist der dynamische Wechsel des Verhaltens zur Laufzeit ohne Einfluss auf den Zustand möglich, wodurch ein Aktor als eine äußerst flexible Berechnungseinheit betrachtet werden kann.

3.3.4 Aktor-System

Ein Aktor-System repräsentiert eine Menge von Aktoren, die miteinander kommunizieren können. Im Rahmen der entwickelten Bibliothek wird ein Aktor-System durch den Typ `ActorSystem` repräsentiert. Jeder Aktor muss mit einem Aktor-System assoziiert sein, damit dieser ausgeführt wird. Nach der Übergabe eines Aktors an ein Aktor-System erzeugt dieses ein Async-Future, in dem der Aktor respektive dessen Verarbeitungsaktionen ausgeführt werden. Der Scheduler der Async-Runtime verwaltet alle auszuführenden Async-Futures und somit auch die Ausführung der Aktoren. Bei dem Erhalt einer Nachricht wird das Async-Future eines Aktors benachrichtigt, was

wiederum dem Scheduler signalisiert, dass das entsprechende Future erneut ausgeführt werden kann. Die Ausführung von Futures basiert in Tokio auf Thread Pools, was bedeutet, dass jede Ausführung eines Futures durch einen anderen Thread durchgeführt werden kann. Dies ist auch der Grund, wieso der Zustand eines Aktors das `Send`-Trait implementieren und die Lifetime `'static` besitzen muss, da die Aktoren von dem ausführenden Future besitzt werden und diese bei der Ausführung zwischen verschiedenen Threads bewegt werden müssen.

Die Implementation des Aktor-Systems stellt sicher, dass technische Details wie dieses vom Benutzenden der Bibliothek nicht berücksichtigt werden müssen. Die Erzeugung eines Aktor-Systems ist in Abb. 3.11 dargestellt.

```
let mut actor_sys = ActorSystem::new();
actor_sys.spawn(actor1,  "actor1".to_string());
actor_sys.spawn(actor2,  "actor2".to_string());
actor_sys.start().await;
```

Abbildung 3.11: Durch die Erzeugung eines Aktor-Systems können Aktoren ausgeführt werden. Weil die Ausführung von Aktoren intern auf der Verwendung von Futures basiert, muss der Aufruf der `start()`-Methode des Aktor-Systems mit dem `await`-Schlüsselwort ausgeführt werden.

Tritt im Rahmen der Abarbeitung einer Nachricht ein Fehler auf, so wird das Aktor-System über diesen benachrichtigt und die Ausführung des Aktors gestoppt. Wurde ein Aktor gemeinsam mit einer Überwachungsstrategie gestartet, so entscheidet diese, wie die weitere Vorgehensweise ist. Die Funktionsweise von Überwachungsstrategien wird in Abschnitt 3.3.4.2 detailliert beschrieben. Wurde ein Aktor ohne eine Überwachungsstrategie gestartet, so werden dessen Ressourcen automatisch freigegeben und der Aktor aus dem Aktor-System entfernt.

Die Adressen aller Aktoren, die mit einem Aktor-System assoziiert sind, werden gemeinsam mit einem eindeutigen Namen gespeichert. Damit ein Aktor die Adresse eines anderen Aktors erhält, kann dieser, sofern der Namen des anderen Aktors bekannt ist, diese über das Aktor-System erhalten. Ein Beispiel hierfür ist in Abb. 3.12 dargestellt.

Eine zweite Möglichkeit für den Erhalt einer Adresse wird in Abschnitt 3.3.4.1 beschrieben.

```
let mut behavior = BehaviorBuilder::new()
.on_ask::<String>(|msg,state,reply_to,ctx| -> BehaviorAction<State> {
    // query gibt eine optionale Adresse zurück, weil der
    // gesuchte Aktor mitunter nicht existiert
    match ctx.query("another_actor") {
        None => {}
        Some(addr) => { addr.tell("hello".to_string()); }
    }
    Behavior::keep()
    )})
.build();
```

Abbildung 3.12: Aktoren können ihren Aktor-Kontext verwenden, um Zugriff auf Funktionen des Aktor-Systems zu erlangen. In diesem Beispiel wird die Adresse eines Aktors über dessen Name bestimmt.

3.3.4.1 Broadcast-Nachrichten

Damit Aktoren miteinander kommunizieren können, müssen diese die Adresse des empfangenden Aktors kennen. Eine mögliche Strategie für das Finden der Adresse eines Aktors ist, eine Nachricht an alle sich in einem Aktor-System befindlichen Aktoren zu senden. Die Aktoren, die eine entsprechende Funktion anbieten, antworten dann auf diese Nachricht mit ihrer Adresse. Damit diese Strategie implementiert werden kann, verfügt jedes Aktor-System über eine Funktion, die eine gegebene Nachricht an alle sich in einem Aktor-System befindlichen Aktoren sendet. Ein Beispiel für einen Aktor, der nach seinem Start eine Nachricht an alle Aktoren in dem Aktor-System sendet, ist in Abb. 3.13 dargestellt. Die alternative Verwendung dieser Funktionalität, nämlich das Versenden von Broadcast-Nachrichten direkt über das Aktor-System, ist in Abb. 3.14 abgebildet.

Da bei dem Senden einer Nachricht auch dessen Besitz übertragen wird, müssen Nachrichten, die via Broadcast an alle Aktoren in einem Aktor-System gesendet werden sollen, das `Clone`-Trait implementieren. Jeder Aktor erhält eine Kopie der Originalnachricht und den Besitz der Kopie. Aufgrund dieses Mechanismus ist die Verwendung von Broadcast-Nachrichten, sofern möglich, zu vermeiden, da eine große Anzahl an Kopien erzeugt wird. Des Weiteren sind die Datentypen von Nachrichten, die via Broadcast versendet werden sollen, möglichst klein zu wählen, um die Speichernutzung zu reduzieren und den Kopieraufwand gering zu halten.

```
let behavior = BehaviorBuilder::new()
.on_start(|state, ctx| {
    // Sendet "hello world" an alle Aktoren im Aktor-System
    ctx.broadcast_tell("hello world".to_string());
})
.build();
let actor = Actor::new((), behavior, MailboxType::Unbounded);
```

Abbildung 3.13: Ein Aktor kann über seinen Kontext Nachrichten an alle sich in einem Aktor-System befindlichen Aktoren senden.

```
let mut actor_sys = ActorSystem::new();
actor_sys.spawn(actor1,  "actor1".to_string());
actor_sys.spawn(actor2,  "actor2".to_string());
actor_sys.broadcast_tell("broadcast!".to_string());
actor_sys.start().await;
```

Abbildung 3.14: Broadcast-Nachrichten können direkt über das Aktor-System versendet werden.

3.3.4.2 Überwachungsstrategien

Tritt bei der Abarbeitung einer Nachricht eines Aktors ein Fehler auf, so kann auf diesen auf verschiedene Arten reagiert werden. Im einfachsten Fall wird der Aktor gestoppt und dessen Ressourcen freigegeben. Da dieses Verhalten nicht immer wünschenswert ist, kann für einen Aktor eine Überwachungsstrategie definiert werden. Eine Überwachungsstrategie wird über das Auftreten eines Fehlers informiert und hält eine Kopie des initialen Zustands und Verhaltens eines Aktors. Diese Eigenschaft von Überwachungsstrategien impliziert, dass der Zustand und das Verhalten eines Aktors kopierbar sein muss, damit eine initiale Kopie erzeugt werden kann. Die Implementation des Verhaltens wurde entsprechend diesem Prinzip entwickelt und implementiert immer automatisch das `Clone`-Trait. Soll eine Überwachungsstrategie gemeinsam mit einem Aktor verwendet werden, so muss der Entwickelnde sicherstellen, dass der Zustand des Aktors das `Clone`-Trait implementiert. Das Postfach, die darin befindlichen Nachrichten und die Adresse eines Aktors werden von einem auftretenden Fehler nicht beeinflusst und bleiben erhalten.

Damit Nutzende der Bibliothek eigene Überwachungsstrategien definieren können, muss

ein Typ, der als Überwachungsstrategie verwendet werden soll, nur das `Supervision`-Trait implementieren. Ein Beispiel für die Implementation einer Überwachungsstrategie ist in Abb. 3.15 dargestellt.

```
struct MyStrategy {}
impl<S: Send + Clone> SupervisionStrategy<S> for MyStrategy {
    fn apply(&mut self, exit_reason: ExitReason, backup: &Backup<S>,
    actor: &mut Actor<S>) -> SuperVisionAction {
        // exit_reason entspricht dem Grund für das Aufrufen
        // der Überwachungsstrategie
        match exit_reason {
            ExitReason::Kill => { return Exit;}
            ExitReason::Restart => {
                actor.apply_backup(&backup);
                return Restart;
            },
            ExitReason::Error => {
                actor.apply_backup(&backup);
                return Restart;
            }
        }
    }
}
```

Abbildung 3.15: Durch die Implementation des `SupervisionStrategy<S>`-Traits können neue Überwachungsstrategien definiert werden.

Die anzuwendende Überwachungsstrategie wird gemeinsam mit dem Aktor bei dessen Registrierung in das Aktor-System übergeben. Ein Beispiel für die Registrierung eines Aktors mit einer Überwachungsstrategie ist in Abb. 3.16 dargestellt.

Eine mögliche Reaktion auf einen Fehler ist, den Aktor auf dessen Initialzustand zurückzusetzen und die Ausführung fortzusetzen. Diese Überwachungsstrategie wird durch `SimpleRestartStrategy` implementiert und ist in die Bibliothek integriert.

3.4 Test-Framework

Aufgrund der parallelen und nebenläufigen Natur des Aktor-Modells erfordert die Entwicklung von Tests für Aktoren einen relativen hohen programmiertechnischen

```
let mut actor_sys = ActorSystem::new();
// Für actor1 wird die SimpleRestartStrategy verwendet
actor_sys.spawn_with_supervision(
    actor1, SimpleRestartStrategy::new(), "actor1".to_string()
);
// actor2 hat keine Überwachungsstrategie und wird beim Auftreten
// eines Fehlers beendet.
actor_sys.spawn(actor2, "actor2".to_string());
```

Abbildung 3.16: Jeder Aktor kann mit einer Überwachungsstrategie im Aktor-System erfasst werden.

Aufwand. Damit ein Aktor getestet werden kann, muss ein zweiter Aktor implementiert werden, der den ersteren testet, da eine Interaktion mit Aktoren ausschließlich über Nachrichten geschehen kann und diese Regel für das Testen von Aktoren nicht verletzt werden soll, um in einer möglichst realistischen Testumgebung zu testen. Die manuelle Implementation von Test-Aktoren kann Fehler verursachen, wodurch Fehler im eigentlich zu testenden Aktor nicht gefunden werden. Die in die Bibliothek integrierte Test-Funktionalität ermöglicht eine einfache Definition von Test-Aktoren, ohne dass konkrete Aktoren manuell implementiert werden müssen. Mittels der implementierten Test-Funktionalität können die Reaktionen eines Aktor auf verschiedene Arten und Typen von Nachrichten getestet und deren Auswirkungen auf den inneren Zustand evaluiert werden.

Um den Aufwand für die Implementation eines Aktors, der einen anderen Aktor testet, zu reduzieren, wurden die Gemeinsamkeiten, die ein Test-Aktor benötigt, gesammelt und analysiert. Damit ein Aktor verwendet werden kann, um einen anderen Aktor testen zu können, muss dieser im Allgemeinen:

- Nachrichten senden können.
- Nachrichten empfangen und deren Inhalte überprüfen können.
- die Nachrichtenreihenfolge prüfen können.
- den Zustand des zu testenden Aktors lesen und prüfen können.

Aufgrund der genannten Anforderungen hat sich ergeben, dass Tests dieser Art mittels eines endlichen Automaten innerhalb eines Aktors dargestellt werden können. Die Übergänge und Zustände des endlichen Automaten werden automatisch aus einer

Testdefinition generiert und im Zustand eines Test-Aktors gespeichert. Test-Aktoren werden mittels des Builder-Patterns definiert, wodurch eine sehr einfache und schnelle Definition von Testfällen möglich ist. Ein Beispiel für die Definition eines Test-Aktors ist in Abb. 3.17 dargestellt. Die Details dieses Beispiels werden im Folgenden näher erklärt.

```
let behavior = BehaviorBuilder::new()
    .on_tell::<i32>(|num, state, ctx| -> BehaviorAction<i32> {
        *state += num;
        Behavior::keep()
    })
    .enable_state_checks()
    .build();

let actor = Actor::new(0, behavior, MailboxType::Unbounded);
let addr = actor.get_addr();
let sys = ActorSystem::new();
sys.spawn(actor, "actor to be tested".to_string());

let test_actor = ActorTestBuilder::new(addr)
    .check(|state: &i32| *state == 0)
    .tell(10)
    .check(|state| *state == 10)
    .tell(ActorManageMessage::Kill)
    .build();

let test_res = sys.spawn_test(test_actor).await;
assert_eq!(test_res, true);
```

Abbildung 3.17: Ein Beispiel für das Testen eines einfachen Aktors mittels der integrierten Test-Funktionalität.

Die integrierte Test-Funktionalität ermöglicht den lesenden Zugriff auf den Zustand eines Aktors über ein automatisch implementiertes Protokoll, das für jeden Aktor mittels der Methode `enable_state_checks()` in `BehaviorBuilder`, wie in Abb. 3.18 dargestellt, für die Ausführung von Tests aktiviert werden kann.

Das Protokoll ermöglicht das Senden einer Nachricht mit einer gekapselten Funktion mittels der `check`-Methode, wobei die gekapselte Funktion lesenden Zugriff auf den Zustand des zu testenden Aktors hat. Diese Funktion wird von dem zu testenden

```
let behavior = BehaviorBuilder::new()
.on_ask::<bool>(|msg, mut state, addr, ctx| -> BehaviorAction<()> {
    Behavior::keep()
})
.enable_state_checks()
.build();
```

Abbildung 3.18: Durch den Aufruf von `enable_state_checks()` werden die für das Testen benötigten Protokolle aktiviert und automatisch implementiert.

Aktor mit dessen Zustand ausgeführt. Das Ergebnis der Funktion, das immer vom Typ `bool` sein muss, wird automatisch vom zu testenden Aktor mittels einer Nachricht an den Test-Aktor zurückgesendet. Damit eine Überprüfung dieser Art als gültig akzeptiert wird, muss der Rückgabewert `true` ergeben. Ein Beispiel für die Definition einer Zustandsüberprüfung ist in Abb. 3.19 dargestellt. Durch diesen minimalinvasiven Mechanismus können beliebige lesende Zugriffe auf den Zustand eines zu testenden Aktors durchgeführt werden, ohne dass der zu testende Aktor zusätzlichen manuell implementieren Code integrieren muss, damit dieser testbar ist.

```
// Zustand des zu testenden Aktors
struct State {
    last_msg: bool
}
// Verhalten des Aktors und Erzeugung dessen wird an dieser Stelle
// ausgelassen

let test_actor: Actor<TestActor<State>> =
    ActorTestBuilder::new(actor_addr)
    // Diese Closure wird von dem zu testenden Aktor ausgeführt
    .check(|state: &State| state.last_msg == true)
    .build();
```

Abbildung 3.19: Mittels der `check`-Methode können beliebige lesende Abfragen auf den Zustand eines zu testenden Aktors durchgeführt werden.

Ändert sich der Zustand eines Aktors am Ende der Verarbeitung einer Nachricht, so ist das neue Verhalten ebenfalls mittels des Aufrufs von `enable_state_checks()` zu erzeugen, damit der neue Zustand testbar ist. Diese Implementation verletzt nicht

die Isolierung des Zustands eines Aktors nach außen, da sämtliche Tests ebenfalls auf Nachrichten basieren.

Damit die Antworten eines zu testenden Aktors empfangen und deren Inhalte geprüft werden können, implementiert jeder Test-Aktor automatisch die Verarbeitungsaktionen für die erwarteten Nachrichtentypen, basierend auf der Test-Definition. In Abb. 3.20 ist ein Beispiel für einen Test-Aktor dargestellt, der Ask-Nachrichten sendet und die Inhalte der Antwortnachrichten überprüft.

```
struct State {
    last_msg: bool
}

let behavior = BehaviorBuilder::new()
    .on_ask::<bool>(|msg, mut state, addr, ctx| -> BehaviorAction<State> {
        addr.tell(state.last_msg);
        state.last_msg = msg;
        Behavior::keep()
    })
    .enable_state_checks()
    .build();

let actor = Actor::new(
    State {last_msg: true}, behavior, MailboxType::Unbounded
);
let addr = actor.get_addr();

let sys = ActorSystem::new();
sys.spawn(actor, "actor to be tested".to_string());

let test_actor: Actor<TestActor<State>> = ActorTestBuilder::new(addr)
    .check(|state: &State| state.last_msg == true)
    .ask(false, Response::Tell(|answer: bool| answer == true))
    .check(|state: &State| state.last_msg == false)
    .ask(true, Response::Tell(|answer: bool| answer == false))
    .check(|state: &State| state.last_msg == true)
    .tell(ActorManageMessage::Kill)
    .build();

let test_res = sys.spawn_test(test_actor).await;
assert_eq!(test_res, true);
sys.start().await;
```

Abbildung 3.20: Ein Beispiel für das Testen eines einfachen Aktors mit erwarteten Antwortnachrichten und Zustandsüberprüfungen.

Damit ein Test als erfolgreich abgeschlossen gilt, muss der erzeugte endliche Automat den Endzustand erreichen. In diesem Fall gibt der Test-Aktor als Ergebnis `true`

zurück. Die Verwendung eines `bool`-Ergebnisses ermöglicht eine einfache Integration in existierende Test-Frameworks von Rust. Wird eine definierte Test-Bedingung verletzt, so wird die verletzte Bedingung in der Konsole ausgegeben und der Test abgebrochen.

Ein Test-Aktor unterstützt folgende Funktionen:

- `check`: Überprüft den Zustand des zu testenden Aktors mittels der gegebenen Funktion.
- `tell`: Sendet eine Tell-Nachricht.
- `ask`: Sendet eine Ask-Nachricht und überprüft die Antwort.
- `expect_ask`: Erwartet eine Ask-Nachricht mit definiertem Inhalt.
- `expect_tell`: Erwartet eine Tell-Nachricht mit definiertem Inhalt.

Des Weiteren überprüft ein Test-Aktor die Einhaltung der Reihenfolge der empfangenen Nachrichten gemäß der Testdefinition, indem sich der Zustand des generierten internen endlichen Automaten bei dem Empfang jeder Nachricht ändert und dieser mit der definierten Testdefinition verglichen wird. Es dürfen keine zusätzlichen Nachrichten vom zu testenden Aktor an den Test-Aktor gesendet werden, da dies einem nicht definierten Verhalten entsprechen würde. Der Test wird in einem solchen Fall automatisch als ungültig deklariert. Ein Beispiel für die Definition von sequenziellen Testaktionen ist in Abb. 3.21 dargestellt.

```
let test_actor: Actor<TestActor<State>> = ActorTestBuilder::new(addr)
    .check(|state: &State| state.last_msg == true)
    .ask(false, Response::Tell(|answer: bool| answer == true))
    .check(|state: &State| state.last_msg == false)
    .ask(true, Response::Tell(|answer: bool| answer == false))
    .check(|state: &State| state.last_msg == true)
    .tell(ActorManageMessage::Kill)
    .build();
```

Abbildung 3.21: Die definierten Testaktionen werden strikt sequenziell abgearbeitet und überprüft.

Abhängig von den verwendeten Testfunktionen können wartende Zustände entstehen. Beispielsweise versetzt der Aufruf von `expect_tell` den Test-Aktor in einen Zustand, in dem dieser auf die definierte Tell-Nachricht wartet. Der Test wird erst nach dem

Empfang dieser fortgesetzt. Das interne Protokoll, das für Zustandsüberprüfung mittels `check` verwendet wird, wurde so implementiert, dass dieses keinen Einfluss auf die sequenzielle Abarbeitung der Tests gemäß Testdefinition hat und gegenüber den Nutzenden der Bibliothek vollständig transparent ist. Wartende Zustände, die durch noch fehlende Rückgabewerte von `check`-Aufrufen entstehen, werden intern verwaltet.

Durch die beschriebene, einfache Definition von Test-Aktoren mittels der integrierten Test-Funktionalität werden Fehler bei der manuellen Implementation von Test-Aktoren vermieden, der Aufwand für die Implementation von Tests reduziert und häufiges Testen durch die Nutzenden der Bibliothek ermöglicht. Des Weiteren werden keine Regeln des Aktor-Modells verletzt, da das Test-Framework ebenfalls ausschließlich Nachrichten für das Testen der Funktionalitäten verwendet, wodurch das Testen von Aktoren in einer realistischen Umgebung möglich ist.

4 Praxisbeispiel: Das Schelling-Segregationsmodell mit Aktoren

Um die Praxistauglichkeit der entwickelten Bibliothek exemplarisch zeigen zu können, wurde das Schelling-Segregegationsmodell, was in Abschnitt 4.1 näher beschrieben wird, als agentenbasierte Simulation mit Aktoren implementiert. Bei diesem Ansatz wird jeder Agent der Simulation als Aktor dargestellt, wobei alle Interaktionen zwischen den Aktoren ausschließlich über Nachrichten modelliert werden. Der detaillierte Modellaufbau, der für die Implementationen verwendet wird, wird in Abschnitt 4.2 erläutert.

Die Simulation wurde einerseits in Rust mit der entwickelten Bibliothek, andererseits in Java mit Akka Actor als Vergleich implementiert. In beiden Implementationen wurden die sprachspezifischen Details des Modells minimiert, wodurch ein Vergleich der Implementationen der Aktor-Bibliotheken ermöglicht wurde. In den folgenden Kapiteln wird der Modellaufbau, die Implementationen in den beiden Sprachen, ein Vergleich der Implementationen und Benchmarks vorgestellt. Im letzten Kapitel werden die Ergebnisse und Erkenntnisse dieses Beispiels zusammengefasst.

4.1 Das Schelling-Segregationsmodell

Das Schelling-Segregationsmodell ist ein in Schelling (1971) beschriebenes Modell, das die Segregation von Personengruppen aufgrund von diskriminatorischem Verhalten modelliert. Das Modell besteht aus einem $N \times N$-Gitter, wobei sich auf jedem Feld des Gitters zu einem bestimmten Zeitpunkt maximal eine Person befinden kann. Es existieren zwei disjunkte Gruppen von Personen. Alle Personen werden zu Beginn zufällig auf dem $N \times N$-Gitter platziert. Ein Beispiel hierfür ist in Abb. 4.1 dargestellt.

In einem Simulationsschritt berechnet jede Person den Anteil B von Personen in ihrer Nachbarschaft, die der gleichen Personengruppe angehören wie die Person selbst. Ist der vorhandene Anteil B geringer als ein vordefinierter Minimalwert B_{min}, so wechselt die Person zu einem zufälligen, noch nicht besetzten Feld im Gitter. Wenn der vorhandene Anteil größer als der definierte Minimalwert ist, so bleibt die Person auf dem Feld. Nach mehreren Simulationsschritten kann, abhängig von dem definierten Minimalwert B_{min}, eine Segregation der Personengruppen entstehen. Eine beispielhafte Segregation ist in Abb. 4.2 dargestellt.

Das beschriebene Modell kann basierend auf Aktoren simuliert werden, weshalb dieses Modell als Praxisbeispiel für die entwickelte Aktor-Bibliothek gewählt wurde. Das entsprechende Design wird in Abschnitt 4.2 näher beschrieben.

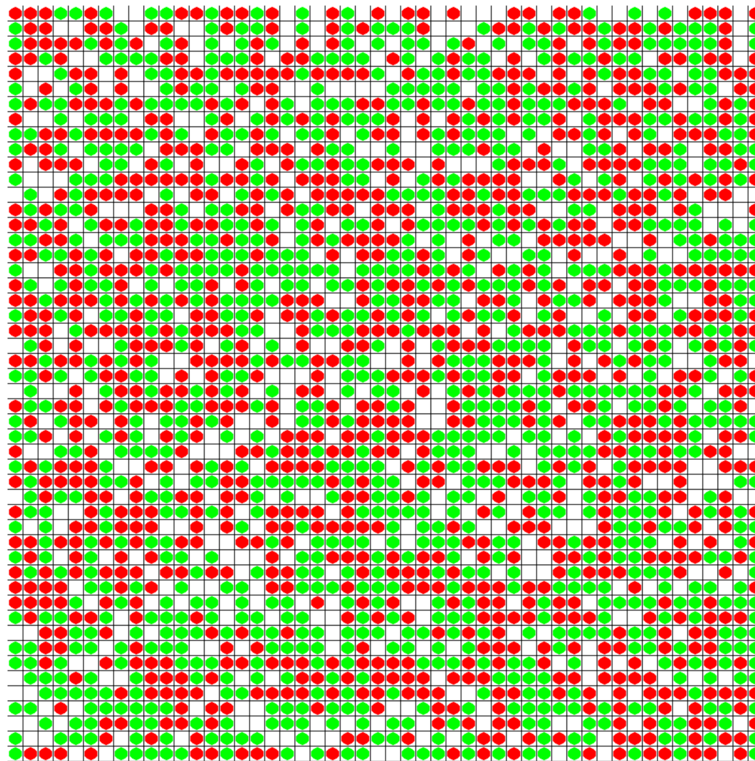


Abbildung 4.1: 50×50-Gitter mit 2.000 zufällig platzierten Personen. Rote und grüne Punkte entsprechen den Personen und ihren Personengruppen, weiße Punkte leeren Feldern.

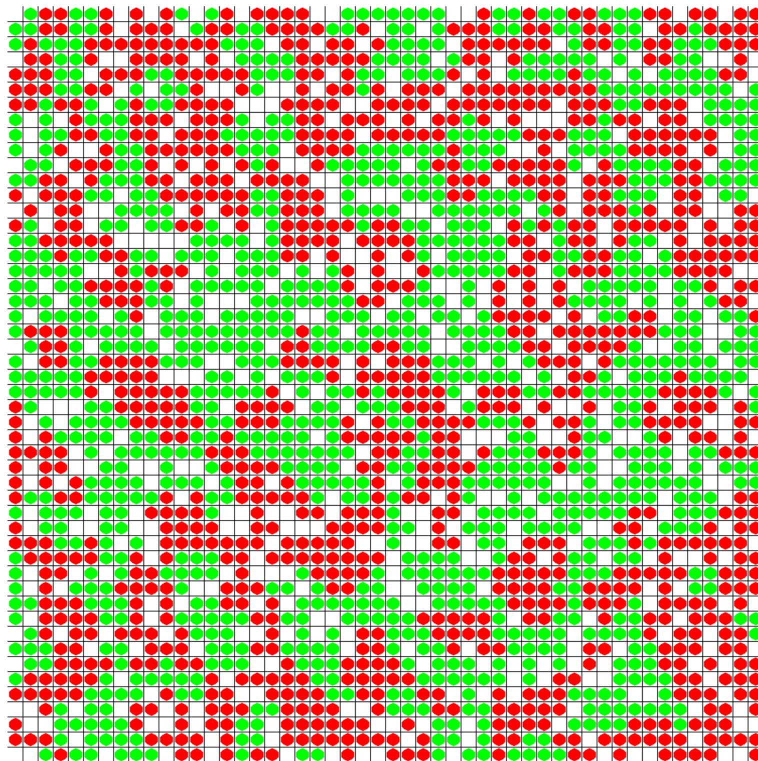


Abbildung 4.2: Nach 100 Simulationsschritten, einem minimalen Gruppenanteil $B_{\text{min}} = 0.4$ und einer Nachbarschaftsgröße von einem Feld sind die Gruppen bereits sichtlich separiert.

4.2 Modell-Design

Das in Abschnitt 4.1 beschriebene Schelling-Segregationsmodell kann mittels Aktoren modelliert werden. Hierzu werden folgende Aktoren definiert:

- Person
- Grid
- Simulation

Ein Person-Aktor repräsentiert eine Person und besitzt folgende grundlegenden Eigenschaften:

- Gruppenzugehörigkeit

- Aktuelle Position auf dem Gitter
- Minimaler Anteil an Personen der gleichen Gruppe B_{min}

Bei dem Start eines Person-Aktor fordert dieser eine initiale Position in dem Gitter vom Grid-Aktor an. Der Grid-Aktor modelliert das $N \times N$-Gitter und verwaltet die besetzten und freien Position. Der Grid-Aktor sendet der Person eine zufällige freie Position auf dem Gitter und reserviert diese. Nach dem Erhalt einer initialen Position informiert der Person-Aktor den Simulation-Aktor, dass dieser initialisiert ist. Dieser erste Schritt ist für einen einzelnen Person-Aktor in Abb. 4.3 dargestellt, wird aber für alle Person-Aktoren in der Simulation durchgeführt.

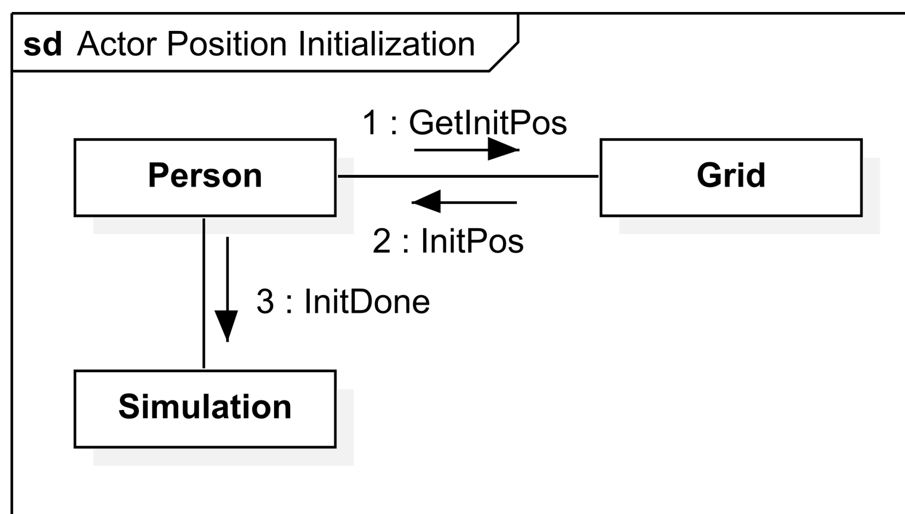


Abbildung 4.3: Jeder Person-Aktor erhält eine zufällige Startposition von dem Grid-Aktor und informiert anschließend den Simulation-Aktor darüber.

Der Simulation-Aktor verwaltet die Ausführung der Simulation und übernimmt die Synchronisation der Simulationsschritte. Nach dem Erhalt aller Initialisierungsnachrichten der Person-Aktoren sendet der Simulation-Aktor eine Nachricht an alle Person-Aktoren, damit diese einen Simulationsschritt ausführen.

Bei dem Erhalt einer solchen Aufforderung fordert jeder Person-Aktor eine Kopie des aktuellen Zustands des Gitters an, damit dieser den Anteil an Personen der gleichen Gruppe B in seiner Nachbarschaft bestimmen kann. Der Grid-Aktor sendet anschließend eine Kopie des Gitters an den Person-Aktor.

Der Person-Aktor berechnet B für seine derzeitige Position. Wenn $B \geq B_{\text{min}}$, dann bleibt der Person-Aktor auf seiner derzeitigen Position und sendet eine Nachricht an den Simulation-Aktor, dass der Simulationsschritt für diesen abgeschlossen ist. Dieses Szenario ist in Abb. 4.4 abgebildet.

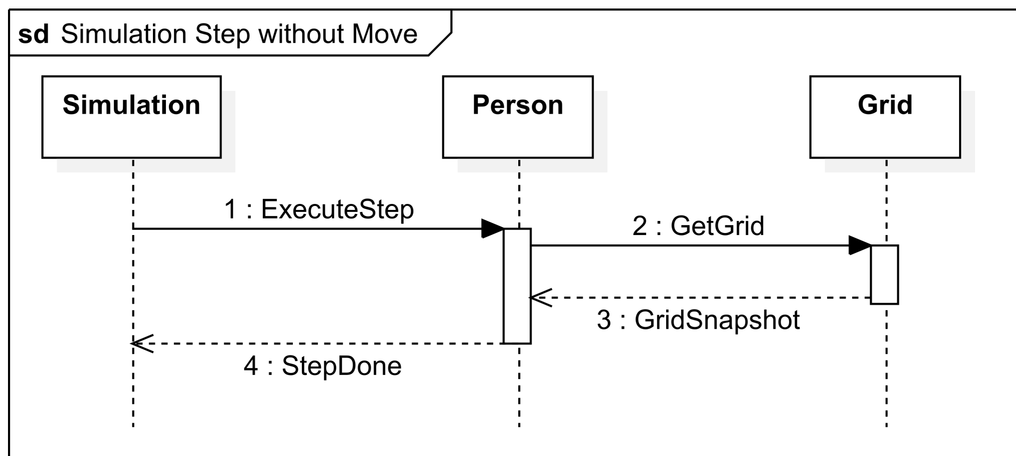


Abbildung 4.4: Person-Aktoren, die mit ihrer Umgebung zufrieden sind, wechseln den Standort nicht.

Wenn $B < B_{\text{min}}$, dann retourniert der Person-Aktor seine derzeitige Position an den Grid-Aktor und fordert eine neue Position an. Der Grid-Aktor wählt eine neue zufällige Position aus, reserviert diese und sendet sie an den Person-Aktor. Der Person-Aktor übernimmt die neue Position und informiert den Simulation-Aktor darüber, dass der Simulationsschritt für diesen Person-Aktor abgeschlossen ist. Dieser Kommunikationsaustausch ist in Abb. 4.5 dargestellt.

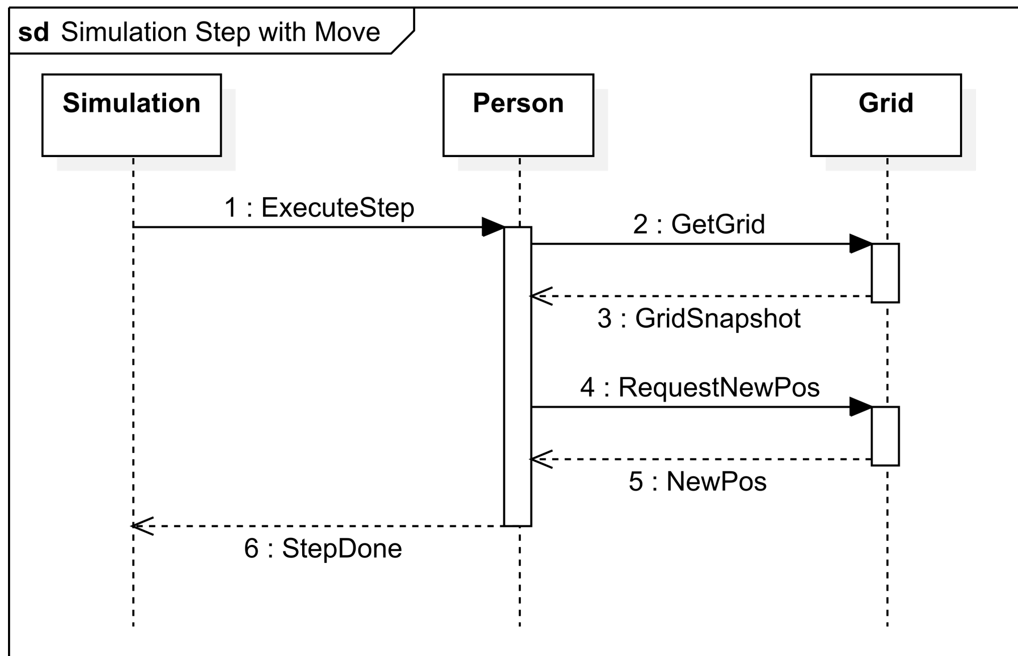


Abbildung 4.5: Wenn $B < B_{\mathrm{min}}$, dann retourniert der Person-Aktor seine derzeitige Position und erhält eine neue, zufällige Position.

Der Simulation-Aktor wartet, bis jeder Person-Aktor seinen Simulationsschritt abgeschlossen hat und initiiert anschließend den nächsten Simulationsschritt.

Aufgrund der beschriebenen Modellierung entstehen in einer Simulation mit P Person-Aktoren pro Simulationsschritt $4P$ bis $6P$ Nachrichten, abhängig davon, wie viele Person-Aktoren ihre Position wechseln.

Diese Modellierung des Schelling-Segregationsmodells als agentenbasierte Simulation mit Aktoren wird in den nächsten Kapiteln in Java Akka und in der entwickelten Rust-Bibliothek implementiert und verglichen.

4.3 Implementation

Das in Abschnitt 4.2 beschriebene Modell wurde in Java Akka und der entwickelten Rust-Bibliothek implementiert. Die Implementation des beschriebenen Modells und des zugehörigen Protokolls war sehr gut möglich, da aufgrund des vordefinierten Designs von Aktoren und Nachrichten diese nur noch in den entsprechenden Bibliotheken

implementiert werden mussten. Im Rahmen der Implementation wurde darauf geachtet, dass sprachspezifische Eigenheiten bestmöglich vermieden wurden, damit ein objektiver Vergleich der Implementationen möglich ist.

Die in Abb. 4.6 und Abb. 4.7 dargestellten Code-Ausschnitte demonstrieren exemplarisch die Ähnlichkeiten der beiden Implementationen anhand der Definition von Verarbeitungsaktionen von Aktoren. Weil beide Bibliotheken den Grundideen des Aktor-Modells folgen, ist eine entsprechende Ähnlichkeit in der Verwendung erkennbar.

```java
@Override
public Receive<Protocol> createReceive() {
    return newReceiveBuilder()
    .onMessage(Protocol.InitPos.class, this::onInitPos)
    .build();
}

private Behavior<Protocol> onInitPos(Protocol.InitPos msg) {
    this.cur_pos = msg.init_pos;
    this.sim.tell(new Protocol.InitDone());
    return this;
}
```

Abbildung 4.6: In Java Akka werden Verarbeitungsaktionen von Aktoren als Methoden der Aktor-Klasse implementiert.

```rust
let mut person_behavior = BehaviorBuilder::new()
.on_tell::<InitPos>(|msg, state, ctx| -> BehaviorAction<Person> {
    state.pos = Some(msg.pos);
    state.sim_actor.tell(InitDone{});
    Behavior::keep()
})
```

Abbildung 4.7: In der entwickelten Rust-Bibliothek werden Verarbeitungsaktionen über das Verhalten definiert.

Eine weitere Ähnlichkeit ist auch in der Definition von Zuständen von Aktoren sichtbar. In Abb. 4.8 und Abb. 4.9 ist der Zustand von Person-Aktoren in Java und Rust dargestellt. Bis auf wenige Unterschiede, wie beispielsweise, dass in Java eine Referenz `null` sein kann, wohingegen dies in Rust mittels des `Option`-Typs explizit

modelliert werden muss, ist die grundsätzliche Verwendung beider Bibliotheken sehr ähnlich. Diese Ähnlichkeit hat den Vorteil, dass Nutzende, die mit der Verwendung von Java Akka vertraut sind, sehr schnell produktiven Code mit der entwickelten Bibliothek entwickeln können und umgekehrt.

```
public class Person extends AbstractBehavior<Protocol> {
    private PopulationClass population_type;
    private int id;
    private ActorRef<Protocol> grid;
    private ActorRef<Protocol> sim;
    private Position cur_pos;
    private double min_happiness;
    private int neighborhood_size;
}
```

Abbildung 4.8: In Java Akka wird der Zustand eines Person-Aktors durch die Attribute der entsprechenden Aktor-Klasse repräsentiert.

```
pub struct Person {
    population_type: PopulationType,
    id: u32,
    grid_actor: Addr,
    sim_actor: Addr,
    pos: Option<Pos>,
    min_happiness: f32,
    neighbourhood_size: i32
}
```

Abbildung 4.9: In der entwickelten Rust-Bibliothek wird der Zustand von Person-Aktoren durch ein Struct abgebildet.

4.4 Benchmarks

Die beiden Implementationen des Schelling-Segregationsmodells in Java Akka und der entwickelten Rust-Bibliothek wurden basierend auf der durchschnittlichen Laufzeit von 100 Simulationsschritten für verschiedene Gitter- und Populationsgrößen verglichen. Für die Ermittlung der Benchmarks in Java wurde Java Microbenchmark Harness (JMH) als Benchmarking-Framework mit folgenden Parametern verwendet:

		Laufzeit über 100 Simulationsschritte			
		Java Akka		Rust	
Gitter	**Personen**	μ	σ	μ	σ
50x50	2000	552 ms	14 ms	162 ms	3 ms
100x100	8000	2122 ms	52 ms	812 ms	16 ms
200x200	32000	9099 ms	69 ms	6475 ms	80 ms
300x300	72000	21156 ms	166 ms	28648 ms	2892 ms

Tabelle 4.1: Benchmark-Ergebnisse über je 20 Durchläufe pro Experiment.

- 2 Forks
- 2 Warmup Iterations
- 10 Iterations

Für die Rust-Implementation wurde das Benchmarking-Framework Criterion mit einer Stichprobengröße von 20 verwendet. Verglichen wurden folgende Szenarien:

- 50×50-Gitter mit 2.000 Personen
- 100×100-Gitter mit 8.000 Personen
- 200×200-Gitter mit 32.000 Personen
- 300×300-Gitter mit 72.000 Personen

In allen Szenarien wurden folgende Modell-Parameter verwendet:

- 100 Simulationsschritte
- $B_{\text{min}} = 0.6$
- Nachbarschaftsgröße von 5 Feldern

In Tabelle 4.1 sind die Ergebnisse der Benchmarks dargestellt. Für kleinere Simulationen bis zu 32.000 Personen weist die Rust-Implementation eine geringere durchschnittliche Laufzeit auf als die Java Akka Implementation. Bei Simulationen mit einer größeren Anzahl an Personen und somit einer größeren Anzahl an Nachrichten pro Simulationsschritt hat die Java Akka Implementation sowohl eine geringere durchschnittliche Laufzeit, als auch niedrige Standardabweichungen der Laufzeiten.

4.5 Zusammenfassung und Diskussion

Die Implementation der in Abschnitt 4.2 beschriebenen agentenbasierten Simulation des Schelling-Segregationsmodells mit Aktoren war sowohl in Java Akka als auch in der entwickelten Rust-Bibliothek sehr gut möglich. Aufgrund des bereits im Vorhinein modellierten Protokolls und den entsprechenden Aktoren und Nachrichten konnte das Design direkt mit den durch die Aktor-Bibliotheken zur Verfügung gestellten Funktionalitäten implementiert werden. Die Implementation mit der entwickelten Rust-Bibliothek hat zu einem zu Java Akka sehr ähnlichen Code geführt, was auf die Ausrichtung der beiden Bibliotheken auf das Aktor-Modell zurückzuführen ist. Der Aufwand für die beiden Implementationen war ungefähr gleich groß.

Ein Unterschied zwischen den Implementationen konnte im Rahmen der Durchführung der Benchmarks festgestellt werden. Für kleinere Simulationen wies die Rust-Implementation eine bessere Laufzeit auf, wobei für größere Simulationen die Java Akka Implementation bessere Laufzeiten aufwies. Mögliche Ursachen für dieses Verhalten können einerseits sprachspezifische Eigenschaften wie beispielsweise Optimierungen der JVM oder des Rust-Compilers, andererseits die unterschiedlichen Implementationen der Aktor-Bibliotheken respektive deren Repräsentationen von Aktoren sein. Für eine detaillierte Bestimmung der Ursachen der Perfomance-Unterschiede müssten weitere Experimente und Benchmarks durchgeführt werden.

Das Ziel dieses Kapitels, nämlich zu zeigen, dass die entwickelte Bibliothek für praktische Anwendungen verwendet werden kann, wurde erfüllt, da das beschriebene Modell mit einem zu Java Akka vergleichbaren Aufwand mit ähnlichen Performance-Merkmalen implementiert werden konnte.

5 Diskussion und Interpretation

In diesem Kapitel werden die Erfahrungen, die im Rahmen der Entwicklung der Aktor-Bibliothek mit Rust gesammelt wurden, beschrieben und diskutiert. Anschließend werden vorhandene Einschränkungen der entwickelten Bibliothek aufgezeigt, anhand von Code-Beispielen erläutert und deren Auswirkungen beschrieben. Abschließend wird die Verwendbarkeit der entwickelten Bibliothek diskutiert.

5.1 Rust als Sprache

Im Rahmen der Entwicklung dieser Bibliothek hat sich die Programmiersprache Rust als sehr ausgereift und praktisch erwiesen. Die statische Typisierung ermöglicht das Design von Software basierend auf Typen, wodurch der entwickelte Code sehr robust gegenüber Fehlern, die häufig in dynamisch typisierten Sprachen auftreten, ist. Trotz des manuellen Speicherverwaltungsansatzes muss sich der Entwickelnde nicht explizit um dieses kümmern, da die in Rust implementierten Regeln die Speicherverwaltung implizit durchführen und der Compiler eventuell vorhandene Verletzungen der Regeln aufzeigt, wodurch keine ungültigen Zugriffe auf Speicher entstehen können und jeder Speicher innerhalb der Laufzeit des Programms garantiert freigegeben wird. Die strikten Regeln des Ownership-Modells haben initial einen sehr einschränkenden Charakter, weshalb anfangs viel Zeit mit den in Rust üblichen Design-Patterns und Vorgehensweisen für bestimmte Probleme verbracht werden muss. Nach der initialen Einarbeitungsphase erweisen sich die strikten Regeln allerdings als sehr hilfreich und ermöglichen die Entwicklung von performanter und sicherer Software mit dem Compiler als unterstützendes Hilfselement. Der in Rust integrierte Paket-Manager cargo besitzt mehrere Funktionen, unter anderem die automatische Installation von Bibliotheken, die Generierung der Dokumentation, das Testen von Code und die Erzeugung von ausführbaren Binärdateien. Die Verwendung von cargo ist sehr intuitiv und praktisch, da nur ein Werkzeug für die gesamte Entwicklung von Rust-Projekten benötigt wird.

Die in Rust existierenden Bibliotheken sind zum Großteil sehr gut dokumentiert, getestet und von hoher Qualität.

Insgesamt ist Rust vor allem als Ersatz für C und C++ sehr zu empfehlen. Je nach Anwendungsgebiet kann es auch als Ersatz für dynamisch typisierte Sprachen wie beispielsweise Python verwendet werden, wobei hier vor allem überprüft werden muss, ob die benötigten Bibliotheken in einer entsprechenden Qualität vorhanden sind.

5.2 Einschränkungen der Bibliothek

Trotz der zahlreichen implementierten Features besitzt die entwickelte Bibliothek noch Einschränkungen, die in diesem Kapitel erläutert werden.

Im Rahmen der Durchführung der Benchmarks für Abschnitt 4.4 wurde festgestellt, dass die entwickelte Bibliothek im Vergleich zu Java Akka eine durchgängig niedrigere Ressourcennutzung aufweist. Diese im ersten Moment positiv erscheinende Erkenntnis könnte der Grund für die schlechtere Performance der entwickelten Bibliothek bei einer großen Anzahl an Aktoren sein, weil die vorhandenen Ressourcen möglicherweise nicht optimal verwendet werden. Potenzielle Ursachen für dieses Verhalten sind mit hoher Wahrscheinlichkeit in der Implementation der Tokio Async-Runtime zu finden, da diese für IO-bound Operationen optimiert wurde. Diese Tatsache ist darauf zurückzuführen, dass Futures in Rust im Allgemeinen für Berechnungen verwendet werden, die auf ein länger nicht vorhandenes Ein-/Ausgabeergebnis warten müssen. Weil das in Kapitel 4 beschriebene Modell keine Teiloperationen aufweist, die länger auf ein externes Ergebnis warten müssen, und gleichzeitig eine sehr große Anzahl an Nachrichten, konkret 288.000 bis 432.000 Nachrichten pro Simulationsschritt bei 72.000 Personen, versendet werden, könnte die schlechtere Performance der Bibliothek auf die Verwendung von Futures zurückzuführen sein. Da das Ziel des Aktor-Modells im Allgemeinen nicht eine Performance-Optimierung ist und dieser Effekt erst bei einer großen Anzahl an Nachrichten und Aktoren effektiv auftritt, sind die Auswirkungen dieser Einschränkung für dieses Anwendungsgebiet vernachlässigbar.

Das Aktor-Modell sieht vor, dass die Zustände von Aktoren vollständig voneinander isoliert sind, was umgekehrt bedeutet, dass Aktoren keinen gemeinsamen Speicher besitzen dürfen. Diese Einschränkung konnte in der entwickelten Bibliothek bis zu einem bestimmten Grad durch das Design eingeschränkt werden. Eine Umgehung

dieser Einschränkung erfordert einen zusätzlichen, sehr gezielten Einsatz von Smart Pointern, wodurch das Teilen von Speicher sehr explizit und gleichzeitig auch sicher gemacht wird. Die Details zu den Möglichkeiten, wie Speicher in der entwickelten Bibliothek auf verschiedene Arten geteilt werden kann, werden in Abschnitt 5.3 und Abschnitt 5.4 beschrieben.

5.3 Teilen von Speicher mittels Nachrichten

Aufgrund der vorhandenen Mechanismen von Rust kann nicht verhindert werden, dass Speicher geteilt wird. Das Teilen von Speicher ist allerdings nur mit bestimmtem Aufwand und in sicherer Weise möglich. In diesem Kapitel wird beschrieben, wie Speicher über Nachrichten geteilt werden kann.

Die einzige Einschränkung eines Datentyps, damit dessen Werte als Nachricht verwendet werden können, ist, dass dieser, wie in Abschnitt 3.3.3.3 beschrieben, `Any + Send` implementieren muss. Weil das `Any`-Trait als Supertrait `'static` besitzt, bedeutet dies, dass ein Datentyp, der als Nachricht verwendet werden soll, die Traits `Any + Send` implementieren und die Lifetime `'static` erfüllen muss. Wie in Abschnitt 3.1 beschrieben sieht das Aktor-Model vor, dass kein Speicher zwischen Aktoren geteilt werden soll. Dieser Aspekt wurde in der Entwicklung der Bibliothek berücksichtigt, kann aber mit den derzeitig verfügbaren Sprachfeatures von Rust nicht vollständig erzwungen werden. Im Folgenden wird eine beispielhafte Umgehung dieser Regel demonstriert und anschließend erklärt, wieso die Möglichkeit einer solchen per se kein Problem darstellt.

`Arc<Mutex<String>>` ist beispielsweise ein Datentyp, der alle oben beschriebenen Anforderungen erfüllt. Wie in Abschnitt 2.1.12 beschrieben sind `Arc` und `Mutex` Smart Pointer, die die Funktionen des enthaltenen Datentyps erweitern. In diesem konkreten Fall fügt `Mutex` einen Synchronisierungsmechanismus zu `String` hinzu, wodurch `Mutex<String>` sicher von mehreren Threads gleichzeitig bearbeitet werden kann. Der Smart Pointer `Arc` ermöglicht, dass der Typ `Mutex<String>` mehrere Owner in verschiedenen Threads gleichzeitig besitzen kann, wobei der Wert erst freigegeben wird, wenn der letzte Owner freigegeben wurde. Der Datentyp `Arc<Mutex<String>>` ermöglicht also, dass einerseits mehrere Owner existieren können, andererseits, dass diese den Wert auf eine sichere Art und Weise bearbeiten können. Weil dieser Datentyp als Nachricht verwendet werden kann, können somit Daten über Nachrichten geteilt

werden. Ein Beispiel für das Teilen von Speicher auf diese Art ist in Abb. 5.1 dargestellt. Da dies nur durch die Verwendung von speziellen Maßnahmen, in diesem Fall Smart Pointer, möglich ist und der Compiler somit sicherstellen kann, dass alle Operationen sicher sind, ist die Möglichkeit einer solchen Umgehung der Regeln nicht problematisch. Des Weiteren ist der Aufwand für eine Umgehung so groß, dass dies im Rahmen einer normalen Verwendung der Bibliothek sehr unwahrscheinlich versehentlich auftreten kann, sehr explizit sichtbar ist und somit vom Nutzenden der Bibliothek gewünscht ist.

```
type SharedMsg = Arc<Mutex<String>>;
let mut behavior = BehaviorBuilder::new()
.on_tell::<SharedMsg>(|msg, state, ctx| -> BehaviorAction<()> {
    // Mutex Lock erhalten
    let mut str = msg.lock().unwrap();
    // Inhalt des Strings leeren
    str.clear();
    ctx.kill();
    Behavior::keep()
    // Mutex Lock wird bei Drop freigegeben
})
.build();
// Der Speicher des Strings "hello world" wird geteilt
let shared_str = Arc::new(Mutex::new(String::from("hello world!")));
let actor = Actor::new((), behavior, MailboxType::Unbounded);
let tx = actor.get_addr();

let actor_sys = ActorSystem::new();
actor_sys.spawn(actor,  "actor".to_owned());
// shared_str.clone() klont nur die Arc-Referenz, nicht den Inhalt
tx.tell(shared_str.clone());
actor_sys.start().await;
// shared_str ist jetzt leer
println!("{}", shared_str.lock().unwrap());
```

Abbildung 5.1: Über Smart Pointer kann Speicher durch Nachrichten geteilt werden.

5.4 Teilen von Aktor-Zuständen

Eine zweite Möglichkeit, Speicher im Rahmen dieser Bibliothek zu Teilen, ist über den Zustand von Aktoren. Datentypen, die als Zustand eines Aktors verwendet werden können, sind, wie in Abschnitt 3.3.3.1 beschrieben, alle Datentypen, die `Send + 'static` implementieren. Ähnlich wie in Abschnitt 5.3 erläutert ist eine

Umgehung der Regel, dass kein Speicher zwischen Aktoren geteilt werden kann, durch die Verwendung von Smart Pointern möglich.

Wird beispielsweise als Zustand eines Aktors `Arc<Mutex<String>>` gewählt, so kann dieser über Nachrichten, aber auch mit anderen Code-Stellen geteilt werden. Eine Beispielimplementation dieser Idee ist in Abb. 5.2 dargestellt. Weil der Zugriff auf den Speicher über `Mutex` geschützt wird, verletzt das Teilen von Speicher zwar die Regeln des Aktor-Modells, ist aber aus speichertechnischer Sicht eine sichere Operation und in bestimmten Umständen für die Implementation spezieller Funktionalitäten nötig.

Beispielsweise wurde für die Implementation der graphischen Darstellung des in Kapitel 4 beschriebenen Projekts der Zustand des Gitters mit dem Thread der Rendering-Bibliothek geteilt, wodurch dieser die Daten ohne die Erzeugung von Kopien darstellen kann. In diesem Beispiel wurde das Teilen von Speicher verwendet, um eine sichere und performante Schnittstelle zwischen Aktoren und normalem Code herzustellen.

```
type SharedState = Arc<Mutex<String>>;
let mut behavior = BehaviorBuilder::new()
.on_tell::<()>(|msg, state, ctx| -> BehaviorAction<SharedState> {
    // Zustand wurde extern geleert
    println!("State: {}", state.lock().unwrap());
    Behavior::keep()
})
.build();

let shared_state = Arc::new(Mutex::new(String::from("hello world!")));
let actor = Actor::new(shared_state.clone(), behavior, MailboxType::Unbounded);
let tx = actor.get_addr();

let actor_sys = ActorSystem::new();
actor_sys.spawn(actor,  "actor".to_owned());

// Zustands-String leeren zu ""
shared_state.lock().unwrap().clear();
tx.tell(());
actor_sys.start().await;
```

Abbildung 5.2: Über Smart Pointer kann der Zustand von Aktoren geteilt werden.

5.5 Verwendbarkeit der entwickelten Bibliothek

Die in Kapitel 4 beschriebene Implementation des Schelling-Segregationsmodells zeigt, dass praktische Problemstellungen mit der entwickelten Bibliothek gelöst werden können. Des Weiteren zeigt der Vergleich mit der Implementation in Java Akka, dass Rust als Systemprogrammiersprache für die Entwicklung dieses Modells keinen höheren Code-Aufwand als Java erfordert und die Performance für kleinere Simulationen in Rust signifikant besser ist. Der Code-Vergleich von Java Akka und Rust verdeutlicht weiters, dass die entwickelte Bibliothek für Nutzende, die mit dem Aktor-Modell bereits vertraut sind, einen schnellen Einstieg in Rust ermöglicht.

Das in Abschnitt 3.4 beschriebene Test-Framework ermöglicht das Testen von Aktoren auf eine schnelle und sichere Art und Weise, wobei der manuell zu schreibende Test-Code drastisch reduziert und somit häufiges Testen motiviert wird.

Die generierte Dokumentation der Bibliothek, die bereitgestellten Code-Beispiele und das strikte Typsystem von Rust ermöglichen einen schnellen und intuitiven Start in die Verwendung der Bibliothek. Alle beschriebenen Aspekte sprechen für eine gute Verwendbarkeit der Bibliothek in praktischen Projekten.

6 Zusammenfassung und Ausblick

Nach einer Einführung der wichtigsten Konzepte von Rust und dem Aktor-Modell wurden existierende Aktor-Modell Implementationen in Rust und Erlang betrachtet. Aufgrund der Einschränkungen der existierenden Bibliotheken in Rust und dem Fehlen einer strukturierten Analyse über die Implementation einer Aktor-Bibliothek in Rust wurde das Schreiben dieser Arbeit motiviert.

Anschließend wurden die grundsätzlichen Funktionalitäten einer Aktor-Bibliothek entworfen und vorgestellt. Für alle essenziellen Komponenten der Aktor-Bibliothek wurden potenzielle Implementationsmöglichkeiten in Rust diskutiert, bevor die konkreten Implementationsdetails erklärt wurden. Zusätzlich zu der Aktor-Bibliothek wurde ein Test-Framework entwickelt, das das Testen von Aktoren drastisch erleichtert.

Anhand der Implementation des Schelling-Segregationsmodells in Java Akka und der entwickelten Rust Bibliothek wurde gezeigt, dass praktische Problemstellungen mit der entwickelten Bibliothek effizient und mit geringem Code-Aufwand gelöst werden können. Abschließend wurden Einschränkungen der entwickelten Aktor-Bibliothek vorgestellt und deren Auswirkungen diskutiert.

Durch die Entwicklung der Aktor-Bibliothek in Rust konnte gezeigt werden, dass sich Rust für die Entwicklung einer solchen gut eignet. Rust bietet alle für die Entwicklung einer Aktor-Bibliothek benötigten Funktionalitäten, Datentypen und Bibliotheken und schränkt das Design einer solchen in keiner Weise ein. Des Weiteren wurde gezeigt, dass die speziellen Eigenheiten von Rust, wie zum Beispiel das Ownership-Modell und das Borrowing-Modell, die Entwicklung von sicherem und effizientem Code ermöglichen.

Eine Problemstellung, die in zukünftigen Arbeiten bearbeitet werden könnte, wäre die Erweiterung der entwickelten Bibliothek auf mehrere Systeme. Hierfür wären Anpassungen der Adresse von Aktoren und dem Aktor-System nötig. Des Weiteren müsste ein effizienter Algorithmus für das Verteilen von Aktoren auf mehrere Systeme, basierend auf den vorhandenen Ressourcen, entwickelt werden.

Eine weitere interessante Aufgabenstellung wäre das Portieren der entwickelten Bibliothek für Embedded-Systeme. Hierfür müsste zuerst die verwendete Async-Runtime durch eine embedded-fähige Async-Runtime ersetzt werden. Anschließend müssten alle Datentypen der Bibliothek entsprechend den Anforderungen des Ziel-Embedded-Systems angepasst werden. Weil viele moderne Microcontroller bereits verhältnismäßig große Arbeitsspeichermengen zur Verfügung haben, ist mitunter sogar die Verwendung von Heap-basierten Datentypen mit einem entsprechenden Embedded-Allokator direkt möglich. Dies müsste im Rahmen der Arbeit evaluiert werden.

Literatur

Agha, Gul Abdulnabi (1. Juni 1985). „ACTORS: A Model of Concurrent Computation in Distributed Systems“. In: Accepted: 2004-10-20T20:10:20Z. URL: https://dspace.mit.edu/handle/1721.1/6952 (besucht am 20. 02. 2022).

Armstrong, Joe (Dez. 2003). „Making reliable distributed systems in the presence of software errors“. Diss. Stockholm, Sweden: The Royal Institute of Technology. 295 S.

Bonet, David (17. Apr. 2022). *Acteur Actor System*. original-date: 2020-02-23T22:59:48Z. URL: https://github.com/DavidBM/acteur-rs (besucht am 28. 05. 2022).

Hewitt, Carl, Peter Bishop und Richard Steiger (20. Aug. 1973). „A universal modular ACTOR formalism for artificial intelligence“. In: *Proceedings of the 3rd international joint conference on Artificial intelligence*. IJCAI'73. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., S. 235–245. (Besucht am 24. 04. 2022).

Hoare, C. A. R. (1. Aug. 1978). „Communicating sequential processes“. In: *Communications of the ACM* 21.8, S. 666–677. ISSN: 0001-0782. DOI: 10.1145/359576.359585. URL: https://doi.org/10.1145/359576.359585 (besucht am 21. 05. 2022).

Larvik, Anton (31. Mai 2018). „Boosting Erlang superpowers at WhatsApp“. Code Beam STO. URL: https://www.youtube.com/watch?v=87lW4Llsj7E&ab_channel=CodeSync (besucht am 06. 09. 2022).

Lerche, Carl (2022). *Tokio - An asynchronous Rust runtime*. URL: https://tokio.rs/ (besucht am 14. 05. 2022).

Reed, Rick (3. Juli 2014). „That's 'Billion' with a 'B': Scaling to the Next Level at WhatsApp“. Erlang Factory. URL: https://www.youtube.com/watch?v=c12cYAUTXXs&ab_channel=ErlangSolutions (besucht am 06. 09. 2022).

Schelling, Thomas C. (1. Juli 1971). „Dynamic models of segregation“. In: *The Journal of Mathematical Sociology* 1.2, S. 143–186. ISSN: 0022-250X. DOI: 10.1080/0022250X.1971.9989794. URL: https://doi.org/10.1080/0022250X.1971.9989794 (besucht am 02. 06. 2022).

Stenman, Erik (19. Feb. 2022). *The BEAM Book*. original-date: 2017-03-23T18:25:31Z. URL: https://github.com/happi/theBeamBook (besucht am 20. 02. 2022).

The Rust Reference (2022). URL: https://doc.rust-lang.org/stable/reference/introduction.html (besucht am 30. 04. 2022).

Vinoski, Steve (Sep. 2007). „Concurrency with Erlang“. In: *IEEE Internet Computing* 11.5. Conference Name: IEEE Internet Computing, S. 90–93. ISSN: 1941-0131. DOI: 10.1109/MIC.2007.104.

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich vorliegende Masterarbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Stellen sind als solche kenntlich gemacht. Die Arbeit wurde bisher weder in gleicher noch in ähnlicher Form einer anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Dornbirn, am 1. Juli 2022 André Hopfgartner