Asynchrones Programmieren in Clojure

mit Hilfe der Bibliothek core.async

WS13/14

Vincent Elliott Wagner
Tobias Schwalm

April 2014

Referent: Prof. Dr. Burkhardt Renz

Inhaltsverzeichnis

Inhaltsverzeichnis I

1 Einleitung 1

2 Technologien 2
2.1 Was bedeutet asynchrone Programmierung? . . . 2
2.2 Blockierende und nichtblockierende I/O . . . 2
2.3 Communicating Sequential Processes . . . 3
2.3.1 Operatoren und Konstrukte in CSP . . . 4
2.3.2 Umsetzung der Konstrukte in core.async . . . 7
2.4 Continuation Passing Style am Beispiel von JavaScript . . . 7
2.5 Asynchrone Programmierung in Clojure . . . 9
2.6 Vergleich von CSP und dem Actor Model . . . 11
2.6.1 Das Actor Model . . . 11
2.6.2 Unterschiede und Gemeinsamkeiten . . . 12

3 Die Bibliothek core.async 13
3.1 API Code-Beispiele . . . 13
3.1.1 *Go*-Blocks . . . 13
3.1.2 Threads . . . 14
3.1.3 Timeout . . . 14
3.1.4 Filter . . . 14
3.1.5 Buffer . . . 15
3.2 ClojureScript . . . 15
3.3 Beispiele von Hoare zu CSP . . . 15

4 Zusammenfassung und Fazit 17

Abkürzungsverzeichnis II

Abbildungsverzeichnis III

Listingverzeichnis IV

Literaturverzeichnis V

1 Einleitung

Diese Ausarbeitung befasst sich mit *core.async*, einer Bibliothek zur asynchronen Programmierung in der Programmiersprache Clojure. Es werden zunächst die grundlegenden Begrifflichkeiten erklärt, andere State-of-the-Art Programmierparadigmen, Entwurfsmuster und Frameworks betrachtet und anschließend mit *core.async* verglichen. Anhand von Beispielen aus dem *Application Programming Interface* (API) des Frameworks, wird der aktuelle Entwicklungsstand des noch im Alpha-Stadium befindlichen Frameworks gezeigt und dessen Potential bewertet.

2 Technologien

2.1 Was bedeutet asynchrone Programmierung?

Zu Beginn stellt sich die Frage nach der Erläuterung des Begriffs *asynchrone Programmierung*. Betrachten wir eine imperative Programmiersprache ohne Multithreading, so lässt sich dieser Begriff leicht erklären. Eine Anwendung in einem imperativen Kontext hat ihren Entry-Point in der ersten Zeile des Programmcodes. Der Code wird streng sequentiell abgearbeitet. Jeder Befehl blockiert die Anwendung, solange bis seine Abarbeitung beendet ist. Dieses Verhalten einer Anwendung wird als *synchrone* Abarbeitung einer Befehlsfolge bezeichnet. Der Nachteil ist hier, dass Wartezeiten zwischen Berechnungen entstehen. Lange Berechnungen reduzieren die Performance der Anwendung, da stets auf ihre Beendigung gewartet werden muss. Hier kommt *asynchrone Programmierung* zum Tragen. Der Begriff beschreibt eine sequentielle Befehlsabarbeitung, bei der keine Wartezeit zwischen Berechnungen entsteht. Die Grundlage hierfür stellen ein nichtblockierendes I/O-Modell, welches im nächsten Kapitel erklärt wird, sowie diverse Programmierparadigmen dar.

2.2 Blockierende und nichtblockierende I/O

Ein nichtblockierendes I/O-Modell ermöglicht die Verarbeitung von Befehlen bei zeitintensiven Berechnungen oder Datenbankabfragen ohne dass der Main-Thread der Anwendung blockiert wird. Es muss nicht auf das Ergebnis der Berechnung gewartet werden, bevor der nächste Befehl ausgeführt werden kann. Der Mechanismus kann beispielsweise mit Hilfe einer Ereignisschleife umgesetzt werden, die die Vorgänge in sogenannten Worker-Threads auslagert und nach Beendigung der Aufgaben die Callback-Funktionen aufruft, die im Main-Thread abgearbeitet werden (siehe Abb. 2.1). Diese Technik nennt sich *Continuation Passing Style* (CPS) und wird im Kapitel 2.4 am Beispiel von JavaScript erklärt. Ein nichtblockierendes I/O-Modell stellt die Grundlage für asynchrone Programmierung dar.

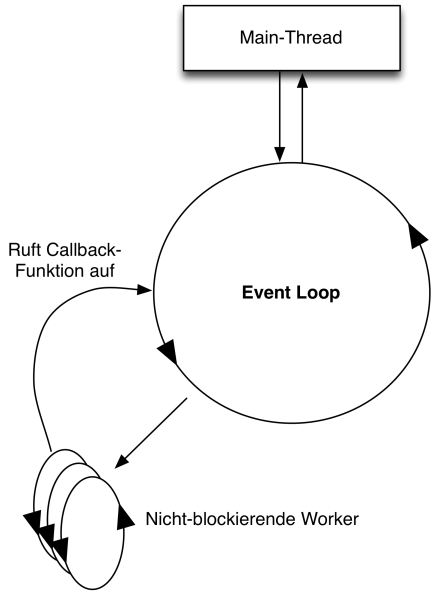

Abbildung 2.1: Event-Loop

2.3 Communicating Sequential Processes

Communicating Sequential Processes (CSP) ist eine von *C.A.R. Hoare*, in einem Paper[1] erstmals vorgestellte formale Modellierung von nebenläufigen Prozessen und deren Kommunikation untereinander, mit Hilfe von Events und Nachrichten.

Hoare hat erkannt, dass in der Zukunft neue Algorithmen und Modelle benötigt werden, die horizontal skaliert werden können. Durch die zwingende Kommunikation zwischen den Prozessen ist ein spezifiziertes Konzept von Nöten, wodurch die Effizienz gesteigert werden kann. Die 1978 übliche Kommunikation über einen *Shared Storage* und die Synchronisation über gegenseitigen Ausschluss, war zu komplex und zu fehleranfällig.[2]

Die in den folgenden Unterkapiteln verwendeten Symbole werden von Hoare in seinem Buch[3] beschrieben.

[1] C.A.R. Hoare, "Communication Sequential Processes", 1978.
[2] Ebd., Introduction.
[3] C.A.R. Hoare, *Communication Sequential Processes*, Juni 2004, Glossary of Symbols.

2.3.1 Operatoren und Konstrukte in CSP[1]

Hoare definiert in seinem Buch zu CSP einige Operatoren und Konstrukte, wovon die Essentiellen erläutert werden. Als Beispiel, wird das von Hoare in seinem Buch[2] verwendete Beispiel eines Schokoladenautomaten gezeigt, jedoch werden die Begriffe in die deutsche Sprache übersetzt.

Events

Events sind Aktionen, die jederzeit auftreten können. Dabei wird nicht zwischen eingehenden und ausgehenden Events unterschieden, diese Semantik ergibt sich durch den Kontext, der Verknüpfung der Prozesse. In Hoares Notation werden Events in Kleinbuchstaben geschrieben. Im Beispiel eines einfachen Schokoladenautomaten existieren die folgenden zwei Events:

münze
: Das Einwerfen einer Münze in den Automaten.

schokolade
: Das Entnehmen der Schokolade aus dem Auswurf des Automaten.

Um den Bezug zu *core.async* herzustellen, werden die Events um Channels erweitert, die Hoare in einem späteren Kapitel[3] im Buch definiert. Wenn der Automat mit Channels definiert wird, besitzt dieser nun den Channel Münzeinwurf (*in*) und den Channel Auswurf (*out*).

Aus der Sicht des Automaten sehen die Events nun folgendermaßen aus:

in.münze
: Das Entnehmen einer Münze aus dem Münzeinwurf.

out.schokolade
: Das Hineinlegen einer Schokolade in den Auswurf.

In nicht allen Fällen ist genau bekannt, um welches Event es sich auf dem Channel handelt. Damit eine Verbeitung dieser möglich ist, existieren die Operatoren *?* und *!*, mit denen Events in Variablen gespeichert werden (*?*-Operator) oder Event aus Variablen auf einen Channel geschrieben werden können (*!*-Operator). Da das Beispiel eines Schokoladenautomaten ungeeignet ist, wird ein neues Beispiel konstruiert.

Hier handelt es sich um einen Automaten ($COPY$), der aus einem Channel *in* Events liest und den Wert in einen weiteren Channel *out* schreibt. Dieser Automat hat die folgenden Events:

[1] Siehe C.A.R. Hoare, *Communication Sequential Processes*, Juni 2004, Kap. 1.1.
[2] Ebd.
[3] Ebd., Kap. 4.2.

in?x
: Das Entnehmen eines Eventes aus dem *in*-Channel und das Speichern in der Variable x.

out!x
: Das Hineinlegen des Wertes der Variable x auf den *out*-Channel.

Alle nachfolgenden Konstrukte werden mit Hilfe von Channels erklärt.

Prozesse

Ein Prozess wird durch eine Kombination aus verschiedenen Events, die sequentiell auftreten können definiert. In dem ersten Beispiel eines Schokoladenautomaten existiert der Prozess, der den Schokoladenautomaten (SA) definiert. Der Prozess SA ist durch die Großbuchstaben als solches erkenntlich, wie es die Notation vorsieht. Auf ein Event muss stets ein Event oder ein Prozess folgen. Ein terminierbarer Automat kann dementsprechend nicht nach einem Event enden, sondern muss in einem anderen Prozess enden. Hierzu wird der Prozess $STOP$ definiert, der auf die eigentliche Ausführung des Prozesses folgt und den nicht mehr funktionsfähigen Schokoladenautomaten definiert.

Verkettung

Um die sequentielle Abfolge von Events, Prozessen und anderen Konstrukten darzustellen, definiert Hoare den $\rightarrow$-Operator. Der Schokoladenautomat, der eine Münze annimmt, eine Schokolade auswirft und danach kaputt geht, sieht unter Verwendung des Operators wie folgt aus:

$$(\text{in.münze} \rightarrow (out.schokolade \rightarrow STOP))$$

Alternativ kann der Prozess auch benannt und die Klammern können entfernt werden, wodurch die Ausführung nun dementsprechend aussieht:

$$SA = \text{in.münze} \rightarrow \text{out.schokolade} \rightarrow STOP)$$

Rekursion

Einfache Verkettung ermöglicht bereits zu diesem Zeitpunkt die Modellierung von komplexen Prozessen. Allerdings existieren in Prozessen üblicherweise redundante Abläufe, die nicht modelliert werden können oder nicht vom Entwickler zur Modellierung vorgesehen sind. Um nicht den gesamten Ablauf einen Prozesses modellieren zu müssen, sieht die Notation Rekursion vor. Im Folgenden wird der nicht terminierende Automat SA unter Anwendung der neuen Operatoren definiert:

$$SA = (\text{in.münze} \rightarrow \text{out.schokolade} \rightarrow SA)$$

Und das Beispiel des kopierenden Prozesses ($COPY$) sieht folgendermaßen aus.

$$COPY = (\text{in?x} \rightarrow \text{out!x} \rightarrow COPY)$$

Parallelität

Die zu diesem Zeitpunkt vorgestellten Notation ermöglicht ausschließlich die gleichzeitige Ausführung eines Prozesses. Eine sequentielle Durchführung aller Prozesse ist zwar möglich, verkompliziert die einzelnen Prozesse, da ein Scheduling definiert werden muss. Gleichwohl sieht die Notation die parallele Ausführung mehrerer Prozesse vor. Bei der gleichzeitigen Ausführung von Prozessen werden Events, die in beiden Alphabeten vorkommen, synchron ausgeführt und Events, die nur in einem Prozess vorkommen vom anderen Prozess ignoriert. Durch die synchrone Ausführung von zwei gleichen Events ist es möglich die Kommunikation zwischen zwei Prozessen zu modellieren. Der Operator, der Prozesse parallel ausführt ist der //-Operator.

Zur Verdeutlichung werden die folgenden zwei Prozesse definiert:

$$mensa : SA = \mu X \bullet (\text{in.münze} \rightarrow \text{out.schokolade} \rightarrow X)$$

$$student : KUNDE = \mu X \bullet (\text{geldzählen} \rightarrow \text{mensa.in.münze} \rightarrow \text{mensa.out.schokolade} \rightarrow X)$$

$$(mensa : SA)||(student : KUNDE)$$

Der erste Prozess ist der bereits bekannte Schokoladenautomat. Hierbei handelt es sich um eine Instanz mit dem Namen *mensa* und dem Prozesstyp *SA*. μX definiert eine Funktion mit dem Bezeichner X, die dem Prozess *SA* zugewiesen wird. Dadurch kann innerhalb der Funktion der Name X, unabhängig vom Namen der Instanz, verwendet werden. Der zweite Prozess ist ein Kunde mit dem Namen *student*, der den Schokoladenautomat bedient. Er benutzt die Channels Münzeinwurf (*in*) und Ausgabe (*out*) des Schokoladenautomaten und zählt zu Beginn sein Geld. Damit beide Prozesse nun parallel auszuführen werden, wird der //-Operator verwendet.

Die Aktion *in.münze* des Mensa-Prozesses und die Aktion *mensa.in.münze* des Studenten-Prozesses laufen simultan ab.[1] Der Prozess blockiert, wenn der andere Prozess nicht bereit ist. Eine Art von Puffer der die Blockierung verhindern würde, müsste durch einen anderen Prozess modelliert werden.[2].

[1] vgl C.A.R. Hoare, *Communication Sequential Processes*, Juni 2004, Seite 117.
[2] vgl. ebd., Seite 133.

Zusätzliche Konstrukte

Die oben vorgestellten Konstrukte reichen aus, um einen einfachen Automaten zu definieren, der immer den gleichen Ablauf hat. Für komplexere Algorithmen sind Verzweigungen von Nöten, um alternative Abläufe modellieren zu können. Die Notation sieht für diesen Fall den /-Operator vor, der zwischen zwei Abläufen unterscheiden kann.

2.3.2 Umsetzung der Konstrukte in core.async

In *core.async* ist ein Prozess eine Sequenz von verschiedenen Befehlen, die entweder durch einen Thread ausgeführt oder in einem *go*-Block in einen Zustandsautomat umgewandelt werden.

Die Kommunikation zwischen mehreren Prozessen erfolgt über Channels, die mit Hilfe der Funktion *chan* erzeugt werden. Das Hineinlegen eines Wertes (Event), wird in der Notation nicht vom Herausnehmen unterschieden, außer das Event soll zwischengespeichert werden, in diesem Fall existieren die oben erklärten Operatoren (*?* und *!*). In *core.async* wird das Hineinlegen eines Werts in einen Channel mit den Funktionen *>!!* (bei der Verwendung von Threads) oder *>!* (in einem *go*-Block) durchgeführt. Das Herausnehmen eines Werts aus dem Channel wird mit den Funktionen *<!!* und *<!* realisiert.

Alle übrigen Konstrukte, wie Rekursion, Verkettung, Verzweigungen etc. werden durch Clojure abgebildet.

2.4 Continuation Passing Style am Beispiel von JavaScript

Continuation Passing Style beschreibt einen Programmierstil, bei dem die Fortführung (Continuation) eines Programms an ein Unterprogramm übergeben wird. Zumeist werden die Continuations in Form von Funktionszeigern als Parameter an eine Funktion übergeben. In JavaScript werden diese als Callback-Funktionen bezeichnet. Der Einsatz von CPS ermöglicht hier die asynchrone Abarbeitung von Funktionen. Betrachten wir das Code-Beispiel aus Listing 2.1. Im Folgenden werden die Bestandteile des Code Beispiels erklärt:

Die Funktion divides
: ermittelt, ob eine Zahl n ein Teiler der Zahl m ist und liefert einen boolschen Wert zurück.

Die Funktion calcDividers
: nimmt eine Zahl n entgegen sowie eine Continuation in Form einer Callback-Funktion. Mit Hilfe einer Schleife wird pro Iterationsschritt ein Teiler der Zahl

n ermittelt. Sind alle Teiler berechnet, so wird die übergebene Callback-Funktion aus der Parameterliste aufgerufen, als Parameter wird ein Array aller Teiler übergeben.

Die Funktion printDividers
repräsentiert die Continuation in dem Code-Beispiel. Das Programm wird an dieser Stelle fortgesetzt, sobald die Berechnung beendet ist. An dieser Stelle werden alle Teiler ausgegeben.

Die Funktion main
stellt den Entry-Point des Beispiels dar. Die Funktion *calcDividers* wird für die Zahl *65535* aufgerufen und *printDividers* wird als Callback-Funktion übergeben.

Listing 2.1: JavaScript Callback Beispiel

```
1  function divides(n, m) {
2    return (m % n == 0);
3  }
4
5  function calcDividers(n, callback) {
6    var dividers = Array();
7    for(var i = 0; i<n; i++) {
8      if(divides(i, n)) {
9        dividers.push(i);
10     }
11   }
12   callback(dividers);
13 }
14
15 function printDividers(dividers) {
16   var str = "{";
17   for(var i=0; i<dividers.length; i++) {
18     if(i > 0) {
19       str = str.concat(",␣");
20     }
21     str = str.concat(dividers[i]);
22   }
23   str = str.concat("}");
24   console.log(str);
25 }
26
27 function main() {
28   calcDividers(65535, printDividers);
29   console.log("Calculation␣done.");
30 }
31
32 main();
```

Asynchronität

Wird das Code-Beispiel auf Listing 2.1 in einem JavaScript-Interpreter ausgeführt, so lässt sich kein asynchrones Verhalten der Anwendung, wie eingangs beschrieben, erkennen. Listing 2.2 zeigt, wie *calcDividers* asynchron aufgerufen werden kann. Hier wird die Funktion *calcDividers* als Continuation mit Hilfe von *setTimeout* an den JavaScript Event-Handler übergeben, der diese nach Ablauf des festgelegten Timeouts, in diesem Fall *0 ms*, aufruft. Dieser Aufruf erlaubt, dass die nachfolgenden Befehle sofort ausgeführt werden. Wird das Code-Beispiel in einem JavaScript-Interpreter ausgeführt, so wird der Text "Calculation started..." vor der Ausgabe des Ergebnisses angezeigt. Das Ergebnis wird asynchron ausgegeben, wenn die Abarbeitung von *calcDividers* abgeschlossen ist.

Listing 2.2: JavaScript setTimeout Beispiel

```
1 ...
2 function main() {
3   setTimeout(function() { calcDividers(65535, printDividers) }, 0);
4   console.log("Calculation␣started...")
5 }
6
7 main();
```

2.5 Asynchrone Programmierung in Clojure

Clojure ist eine Sprache, die zu Java- oder CIL-Bytecode und JavaScript (ClojureScript) kompiliert werden und somit von einer *Java Virtual Machine* (JVM), *Common Language Runtime* (CLR) oder einem JavaScript-Interpreter ausgeführt werden kann.

In den beiden ersten Laufzeitumgebungen können Threads zur asynchronen und gleichzeitigen Verarbeitung von Befehlen verwendet werden. In JavaScript existiert der im vorherigen Kapitel vorgestellte Event-Handler.

Das folgende Beispiel, sowie alle restlichen Erläuterungen und Beispiele beziehen sich auf Clojure in Verwendung mit einer JVM.

Beispiel anhand der Funktion *future*

Eine Methode eine Aufgabe auszulagern, die dann asynchron und concurrent abgearbeitet wird, funktioniert über die *future*-Funktion (siehe Listing 2.3). Die Funktion übergibt den übergebenen Body an einen Thread in einem *ExecutionService* und gibt ein Future-Objekt an den Main-Thread zurück.

Der Main-Thread kann die Ausführung fortsetzen und weitere Aufgaben erledigen. Im Beispiel wird zusätzlich überprüft, ob es sich beim zurückgegebene Objekt um ein Future-Objekt handelt und mit der Funktion *realized?*, ob bereits ein Ergebnis vorliegt. Wenn die Aufgabe erledigt ist, kann der Main-Thread das Ergebnis der Berechnung über das Future anfragen, indem dieser mit der Funktion *deref* oder dem Makro @ das Objekt dereferenziert. Falls die Aufgabe noch nicht erledigt ist, blockiert der Main-Thread solange, bis das Ergebnis vorliegt.

Listing 2.3: Das asynchrone Ausführen von Befehlen mit der Funktion *future*

```
1  (println (-> (Thread/currentThread) (.getName)))
2  (let [f (future
3            (Thread/sleep 1000)
4            (println (-> (Thread/currentThread) (.getName)) "I␣am␣done")
5            42)]
6    (println (future? f))
7    (println (realized? f))
8    (Thread/sleep 2000)
9    (println (realized? f))
10   (println @f))
11 ...
12 [stdout]:nREPL-worker-7
13 [stdout]:true
14 [stdout]:false
15 [stdout]:clojure-agent-send-off-pool-9 I am done
16 [stdout]:true
17 [stdout]:42
```

Dieses Beispiel verdeutlicht die Verwendung von Threads in Clojure. Falls eine gezieltere Kontrolle über die Definition und Ausführung von Threads gefordert ist, können über die *Java-Interop*-Schnittstelle Threads selbstständig definiert oder es kann das *ExecutionService*-Framework verwendet werden.

Software Transactional Memory

Bei der gleichzeitigen Verarbeitung von Aufgaben und der Benutzung eines *Shared Storages* existiert die Problematik, dass *Race Conditions* entstehen können und die im Objekt gekapselten Daten somit inkonsistent werden können.

In Clojure garantiert das *Software Transactional Memory* (STM) bei der Veränderung von Referenzen, dass eine solche Inkonsistenz verhindert wird. Das STM von Clojure verwendet hierfür ein *Multiversion Concurrency Control* (MVCC) mit Snapshot Isolation. Dadurch kann das Phänomen des *Write Skews*[1] auftreten, das vom Entwickler erkannt und besonders behandelt werden muss.

[1] Burkhardt Renz, *Programmieren in Clojure - Identität, Zustand und Synchronisationskontrolle*, 2013, S. Seite 17.

Falls kein STM erwünscht ist, kann auch das Synchronisationsprinzip von Java über Sperren verwendet werden, indem der kritische Abschnitt der Funktion *locking* übergeben wird.

2.6 Vergleich von CSP und dem Actor Model

Zur Modellierung von nebenläufigen Prozessen existieren neben *Communicating Sequential Processes* (CSP) auch andere Modelle. Ein verbreitetes Modell ist das Actor model (Aktorenmodell). Beide Modelle erfüllen den gleichen Zweck, dadurch haben sie viele Gemeinsamkeiten. Um CSP besser verstehen zu können, ist eine Darstellung des Aktorenmodells und ein Vergleich beider Modelle von Vorteil.

Eine Gegenüberstellung von *Continuation Passing Style* (CPS), Synchronisation mittels gegenseitigen Ausschlusses und CSP macht nur bedingt Sinn, da es sich bei Ersterem um Programmierparadigmen handelt und bei CSP um ein Designmuster.

Im Folgenden wird zuerst komprimiert das Aktorenmodell dargestellt und anschließend werden beide Modelle miteinander verglichen.

2.6.1 Das Actor Model

Das Aktorenmodell ist ein von Carl Hewitt u. a. beschriebenes Modell zur Modellierung von künstlichen Intelligenzen, das sich für nebenläufige und parallele Programmierung eignet. Veröffentlicht wurde das Modell 1973 in dem Paper *A Universal Modular ACTOR Formalism for Artificial Intelligence*[1]. Von Carl Hewitt wurde das Modell mit dem Paper *Actor Model of Computation: Scalable Robust Information Systems*[2] 2011 aufbereitet.

Aktoren bestehen aus folgenden fundamentalen Teilen:

Prozess
: Ein Aktor benötigt einen Prozess, der die Aufgaben des Aktors ausführt. Üblicherweise ist das ein Thread.

Speicher
: Zustände des Aktors werden in einem Speicher gesichert, damit unterschiedlich auf Nachrichten reagiert werden kann.

Kommunikation
: Aktoren müssen untereinander kommunizieren können und sich Nachrichten senden können.

[1] Carl Hewitt u. a., "A Universal Modular ACTOR Formalism for Artificial Intelligence", 1973.
[2] Carl Hewitt, *Actor Model of Computation: Scalable Robust Information Systems*, 2011.

Ein Aktor ist für sich alleine genommen kein Aktor, erst in einem Aktorensystem existiert er. Das Aktorensystem stellt ein Kommunikationsmedium bereit, das nach „best effort“ versucht die Nachrichten zu übertragen. Dementsprechend können Nachrichten verloren gehen oder andere Nachrichten überholen. Aktoren verfügen über eine Adresse, worüber andere Aktoren ihnen Nachrichten senden können. Nachrichten werden in das Aktorsystem gelegt und ein Aktor bekommt „irgendwie“ die Nachricht zugestellt, wenn er sie benötigt. Puffer, Mailboxen und Queues werden im Aktorenmodell **nicht** explizit definiert[1]. Die technische Umsetzung übernehmen die Implementierungen. Ein Aktor verarbeitet mit seinem Prozess nur eine Nachricht zu einem Zeitpunkt und da Aktoren keinen gemeinsamen Speicher benutzen dürfen, können auch keine Race-Conditions auftreten. Das Aktorenmodell ist nicht deterministisch, da die Kommunikation zwischen Aktoren nicht genau definiert ist und die Nachrichten zufällig beim Empfänger eintreffen können, wodurch Determinismus nicht garantiert werden kann.

2.6.2 Unterschiede und Gemeinsamkeiten

Es sind beides theoretische Modelle zur Beschreibung von Parallelität und Nebenläufigkeit. Beide Modelle unterscheiden sich bereits in ihrer Art der Definition. CSP ist eine Prozess-Algebra und basiert auf mathematischen Gesetzen, wodurch keine Unklarheiten existieren. Das Aktorenmodell hingegen orientiert sich an der Physik und kann aufgrund dessen bestimmte Bereiche nicht genau spezifizieren, sodass Axiome als Begründung verwendet werden, wodurch das Modell vereinfacht wird. Das Aktorenmodell ist aufgrund der Axiome nicht deterministisch. Determinismus muss zusätzlich spezifiziert werden. Abläufe in CSP sind deterministisch. Um die Automaten simpel zu halten, kann Nichtdeterminismus mit einem besonderen Operator ($\sqcap$) speziell modelliert werden.
Das Aktorenmodell definiert nicht, wie eine Nachricht empfangen wird. Dies kann über einen blockierenden Aufruf passieren oder eine eingehende Nachricht kann eine Methode aufrufen, wie es zum Beispiel in dem Aktorenframework *Akka* umgesetzt wird. In CSP kann es nur blockierende bzw. parkende Operationen geben, da ein Prozess auf mehrere Channels gleichzeitig warten kann.
Der Datenaustausch zwischen zwei Prozessen in CSP ist simultan, sodass ein Prozess blockiert, wenn kein anderer Prozess mit der gegensätzlichen Aktion auf den Channel zugreift. Ein Puffer muss mit einem weiteren Prozess modelliert werden. Die Kommunikation mit Nachrichten zwischen zwei Aktoren passiert asynchron, da das Aktorensystem die Zustellung der Nachrichten übernimmt. Eine mengenmäßige Begrenzung ist nicht vorgesehen. In technischen Implementierungen des Aktorenmodells wie *Akka* existieren Begrenzungen seitens der Konfiguration oder der Hardware. Zudem sichert *Akka* auch zu, dass Nachrichten nicht verloren gehen und sich nicht überholen können.

[1] Carl Hewitt, *Actor Model of Computation: Scalable Robust Information Systems*, 2011, Seite 3, Rechte Spalte.

3 Die Bibliothek core.async

3.1 API Code-Beispiele

Dieses Kapitel demonstriert die Basis-Funktionalität des Frameworks anhand von Code-Beispielen. Zwecks Vereinfachung der Beispiele wird eingangs die Funktion *read-chan* aus Listing 3.1 definiert, die eine einheitliche Ausgabe der Werte auf dem ihr übergebenen Channel realisiert. Diese Funktion entnimmt mit *loop* und *recur* solange alle Werte aus einem Channel, bis dieser geschlossen wird. Das ein Channel geschlossen ist, ist durch das Entnehmen des Wertes *nil* realisierbar.

Listing 3.1: *read-chan* Hilfsfunktion

```
(defn read-chan
  ([c] (read-chan "" c))
  ([str c]
  (thread
    (loop [val (<!! c)]
      (when (not= val nil)
        (println str val)
        (recur (<!! c))))
      (println "END."))))
```

3.1.1 *Go*-Blocks

Das folgende Code Beispiel aus Listing 3.2 demonstriert den Parking-Mechanismus. Zu Beginn wird ein ungepufferter Channel *c* erzeugt. Anschließend wird mittels *>!* und *<!* in Kombination mit *go* darauf zugegriffen. Ist der Channel belegt, so wird die *>!* Funktion geparkt. Ein ähnliches Verhalten ist bei *<!* erkennbar, welche geparkt wird, wenn kein Wert auf dem Channel vorliegt. Ist ein Wert vorhanden, so wird dieser vom Channel entfernt und auf der Konsole ausgegeben.

Listing 3.2: *Go*-Blocks

```
(let [c (chan)]
  (go
    (>! c "test"))
  (go
    (println (<! c))))
```

3.1.2 Threads

Das obige Beispiel aus Listing 3.2 lässt sich auch mit blockierenden Funktionen in Threads ausführen. Die Korrelate zu den Put- und Take-Operation sind hier *>!!* und *<!!*, welche den jeweiligen Thread, in dem sie ausgeführt werden beim Lesen bzw. Schreiben blockieren (siehe Listing 3.3).

Listing 3.3: Thread

```
(let [c (chan)]
  (thread
    (>!! c "test"))
  (thread
    (println (<!! c))))
```

3.1.3 Timeout

Channels werden in ihrer Lebensdauer zeitlich beschränkt, wenn ihre Erzeugung mit Hilfe der *timeout*-Funktion erfolgt ist. Diese Kanäle werden nach Ablauf des Timeouts automatisch geschlossen. Das Beispiel aus Listing 3.4 demonstriert dieses Verhalten.

Zu Beginn wird ein Channel mittels *timeout* erzeugt und neuer Wert asynchron mit Hilfe von *put!* darauf abgelegt. In einem Thread werden nun, wie in der Funktion *read-chan* aus Listing 3.1 solange Werte vom Channel abgerufen, bis dieser seine Schließung nach vier Sekunden mittels *nil*-Wert signalisiert.

Listing 3.4: Timeout

```
(let [c (timeout 4000)]
  (put! c "test")
  (thread
  (loop [val (<!! c)]
    (when (not= val nil)
      (println val)
      (recur (<!! c))))
    (println "timeout")))
```

3.1.4 Filter

Channels können mit einem Filter versehen werden, der die eingehenden Werte auf die Erfüllung eines Prädikats prüft. Das Beispiel aus Listing 3.5 veranschaulicht die Verwendung der *filter>* Funktion, die diese Funktionalität bereitstellt. Die Funktion erhält ein Prädikat, sowie einen Ausgabe-Channel und gibt einen Eingabe-Channel zurück. Der von der Funktion *filter>* erzeugte Prozess prüft, ob die eingehenden Werte vom Typ String sind. Werte, die dem Prädikat entsprechen, werden auf den Ausgabe-Channel gelegt, alle anderen werden verworfen.

Listing 3.5: Filter

```
(let [out (chan)
      c (filter> string? out)]
  (thread
    (>!! c "abc")
    (>!! c 123))
  (read-chan out))
```

3.1.5 Buffer

Channels sind standardmäßig ungepuffert, d.h. es kann maximal ein Wert abgelegt werden. Die Schreiber werden solange geparkt bzw. blockiert, bis der Channel wieder frei ist. Dieses Verhalten kann bei der Erzeugung des Channels eingestellt werden (siehe Listing 3.6). Im Listing wird bei der Erzeugung des Channels ein Puffer der Größe vier übergeben, der zuvor mit der Funktion *buffer* erzeugt wurde. Alternativ kann die Größe des Kanal-Puffers direkt der *chan*-Funktion als Zahlwert übergeben werden. Der Channel akzeptiert nun vier Werte, anschließend werden alle Schreiber geparkt bzw. blockiert.

Listing 3.6: Buffer

```
(let [c (chan (buffer 4))]
  (dotimes [n 4]
    (>!! c n))
  (dotimes [n 4]
    (println (<!! c))))
```

3.2 ClojureScript

ClojureScript ist ein Compiler, der eine Kompilierung von Clojure Code zu JavaScript ermöglicht.[1] Das core.async Framework ist ClojureScript kompatibel, dennoch werden vereinzelte vorgestellte Konstrukte nicht unterstützt. Da JavaScript Single-Threaded ist, existiert keine Thread-Unterstützung, demnach stehen die Funktionen *>!!*, *<!!*, *thread*, sowie alle weiteren auf Threads basierenden Funktionen (alle mit der Endung *!!*) nicht zur Verfügung.

3.3 Beispiele von Hoare zu CSP

Tony Hoare hat in seinem Paper[2] diverse Beispiele erläutert, in denen *Communicating Sequential Processes* (CSP) angewendet wird. Einige der einfacheren Beispiele sind in

[1] Rich Hickey, *ClojureScript*, 2012.
[2] C.A.R. Hoare, "Communication Sequential Processes", 1978.

core.async implementiert worden und sind auf Github[1] zu finden.

Hoare teilt seine Beispiele in vier Bereiche.

Coroutines

Koroutinen sind Prozesse, die Daten von einem oder mehreren Channels entgegennehmen und diese dann in meistens veränderter Form ausgeben.

In *core.async* sind alle Koroutinen umgesetzt worden und befinden sich in der Datei *coroutines.clj*.

Subroutines and Data Representation

Subroutinen sind Prozesse, die aus Koroutinen zusammengesetzt sind und eine bestimmte Aufgabe erfüllen, wie z. B. das rekursives Berechnen der Fakultät.

Sechs Beispiele definiert Hoare in seinem Paper. In *core.async* umgesetzt wurden lediglich die rekursive Berechnung der Fakultät und einen Divisionsprozess, der aus einem Dividenden und einem Divisor den Quotienten und den Rest errechnet. Diese beiden Beispiele sind in der Datei *subroutines.clj* zu finden.

Monitores and Scheduling

In diesen Beispielen werden Prozesse definiert, die die Aufgabe eines Monitors übernehmen, um den gleichzeitigen Zugriff auf eine oder mehrere Ressourcen zu koordinieren.

Von den Monitorbeispielen sind der Integer-Monitor und das Beispiel der Dining Philosopers umgesetzt worden. Die Monitore in diesem Beispiel sind die Gabeln und der Raum. Die Integer-Monitore ist in der Datei *semaphore.clj* zu finden und das Philosophen-Beispiel in der Datei *philosophers.clj*.

Miscellaneous

Die Beispiele dieses Kapitel ließen sich in keine der oberen drei Kapitel einteilen. Von diesen wurde keines in *core.async* umgesetzt.

[1] `https://github.com/serofax/CSPHoareExamplesCoreAsync`

4 Zusammenfassung und Fazit

Zu Beginn wurde erklärt, was asynchrone Programmierung bedeutet und weshalb ein nicht blockierendes I/O-Modell hierzu benötigt wird. Diese beiden Begriffe wurden mittels JavaScript- und Clojure-Code anschaulich dargestellt. Zudem wurde das Programmierparadigma *Continuation Passing Style* (CPS), das häufig in JavaScript zum Einsatz kommt, erläutert.
Zur Einführung in die Design Prinzipien von *core.async* wurde das Channel- und Prozess-Prinzip aus Tony Hoare's Prozess-Algebra *Communicating Sequential Processes* (CSP) zur Modellierung nebenläufiger Prozesse erklärt. Es wurde gezeigt, dass CSP eine sinnvolle Methode ist, um effizient, unter Ausschluss von Race-Conditions, nebenläufige Modelle zu konstruieren, die von modernen Multi-Core-Prozessoren parallel ausgeführt werden können.
Zur Darstellung der Vor- und Nachteile von CSP, wurde ein Vergleich zwischen diesem und dem Aktorenmodell angestellt. Das Aktorenmodell ist wie CSP ein Prinzip zur Modellierung von Nebenläufigkeit. Es wurde festgestellt, dass keines der Modelle gegenüber dem Anderen präferiert werden kann. Der Entwickler muss nach den bestehenden Anforderungen entscheiden, welches Modell den fachlichen Anforderungen besser entspricht, so auch beim Einsatz von *core.async*. Jedoch sollten auch die Unterschiede der Implementierungen in Betracht gezogen werden, die eventuelle Nachteile des theoretischen Modells aufwiegen.
Das darauffolgende Kapitel dieser Ausarbeitung, befassten sich mit der Darstellung der Basis-Funktionalität der API von *core.async*, die mit Code-Beispielen demonstriert wurde. Die Bibliothek wirkt sehr stabil. Der Umfang der API ist für eine Version im Alpha-Stadium beachtlich und kann bereits jetzt produktiv in kleineren Projekten eingesetzt werden. Es ist hervorzuheben, dass - ausgenommen Thread-basierter Konstrukte - *core.async* kompatibel zu ClojureScript ist und der Clojure-Code demnach zu JavaScript kompiliert werden kann. Dies erleichtert die Modellierung von Quasi-Parallelen (single threaded) Prozessen in JavaScript erheblich und schafft neue Möglichkeiten.
Dieses Projekt hat uns tiefe Einblicke in diverse hochinteressante Programmierparadigmen zur Modellierung von Nebenläufigkeit und Parallelität gewährt und uns persönlich weiter gebracht.

Abkürzungsverzeichnis

API	Application Programming Interface
CLJS	ClojureScript
CLR	Common Language Runtime
CPS	Continuation Passing Style
CSP	Communicating Sequential Processes
I/O	Input/Output
JVM	Java Virtual Machine
MVCC	Multiversion Concurrency Control
STM	Software Transactional Memory

Abbildungsverzeichnis

2.1 Event-Loop 3

Listingverzeichnis

2.1 JavaScript Callback Beispiel 8
2.2 JavaScript setTimeout Beispiel 9
2.3 Das asynchrone Ausführen von Befehlen mit der Funktion *future* 10

3.1 *read-chan* Hilfsfunktion 13
3.2 *Go*-Blocks 13
3.3 Thread 14
3.4 Timeout 14
3.5 Filter 15
3.6 Buffer 15

Literaturverzeichnis

Burkhardt Renz. *Programmieren in Clojure - Identität, Zustand und Synchronisationskontrolle*. Version WS 2013/2014 `http://homepages.thm.de/~hg11260/mat/clj-state-bh.pdf` zuletzt besucht am 31.01.2014. 2013.

C.A.R. Hoare. "Communication Sequential Processes". In: *Communications of the ACM* 21.8 (1978). `http://www.cs.ucf.edu/courses/cop4020/sum2009/CSP-hoare.pdf` zuletzt besucht am 25.01.2014.

— *Communication Sequential Processes*. Electronic version `http://www.usingcsp.com/cspbook.pdf` zuletzt besucht am 27.01.2014. Juni 2004.

Carl Hewitt. *Actor Model of Computation: Scalable Robust Information Systems*. Paper `https://docs.google.com/file/d/0Bykigp0x1j92M0p6b0ZWWE9SS3Frb3loV3NKX2sxdw/edit` zuletzt besucht am 07.03.2014. 2011.

Carl Hewitt, Peter Bishop und Richard Steiger. "A Universal Modular ACTOR Formalism for Artificial Intelligence". In: *Proceeding IJCAI'73 Proceedings of the 3rd international joint conference on Artificial intelligence* (1973). `http://worrydream.com/refs/Hewitt-ActorModel.pdf` zuletzt besucht am 06.03.2014.

Carl Hewitt, Clemens Szyperski und Erik Mejier. *Carl Hewitt explains the essence of the Actor Model of computation*. Video `http://letitcrash.com/post/20964174345/carl-hewitt-explains-the-essence-of-the-actor` zuletzt besucht am 05.03.2014.

Rich Hickey. *ClojureScript*. `http://clojure.org/clojurescript` zuletzt besucht am 16.03.2014. 2012.