Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät
Institut für Informatik

Diskrete, prozessbasierte Simulation in Rust

Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Science (B. Sc.)

eingereicht von: Paula Wiesner
geboren am:
geboren in:
Gutachter/innen: Prof. Dr. Jens-Peter Redlich
Prof. Dr. Henning Meyerhenke

eingereicht am: .... verteidigt am: ....

Inhaltsverzeichnis

1 Motivation **4**
- 1.1 Diskrete Simulation . . . 4
- 1.2 Umfang eines Simulatorkerns . . . 4
- 1.3 Ziel der Arbeit . . . 5
- 1.4 Aufbau der Arbeit . . . 5

2 Grundlagen **5**
- 2.1 Besonderheiten von Rust . . . 5
  - 2.1.1 Generische Funktionen und Strukturen . . . 6
  - 2.1.2 Speichersicherheit . . . 7
  - 2.1.3 Koroutinen in Rust . . . 10
- 2.2 Simulation nebenläufiger Prozesse . . . 14

3 Implementation eines Simulatorkerns **14**
- 3.1 Überblick . . . 14
- 3.2 Executor und Waker . . . 14
- 3.3 Konfiguration des Executors . . . 17
- 3.4 Kontrollvariablen . . . 18
  - 3.4.1 Kontrollvariablen in *mk-simulation* . . . 19
  - 3.4.2 Kontrollvariablen in SLX . . . 19
- 3.5 Prozesse und Pucks . . . 19
  - 3.5.1 Prozesse und Pucks in SLX . . . 22
  - 3.5.2 Pucks in *mk-simulation* . . . 22
  - 3.5.3 Prozesse in *mk-simulation* . . . 22
- 3.6 Kanäle . . . 22

4 Benchmarks **24**
- 4.1 Kontextwechsel . . . 25
- 4.2 *Barbershop* . . . 27
- 4.3 Fähre . . . 29

5 Zusammenfassung und Ausblick **34**

Abbildungsverzeichnis

1 Unterschied zwischen Threads und Koroutinen . . . 10
2 Ablauf einer asynchronen Funktion mit einmaligem Ruf von `await` . . . 12
3 Klassendiagramm des Simulatorkerns . . . 15
4 Ablauf der Funktion `wait_until()` . . . 20
5 Klassendiagramm der Kanäle . . . 23
6 Ergebnisse des Kontextwechsel-Benchmarks . . . 27
7 Vorgang zur Erstellung eines Simulationsprogramms . . . 28
8 Ergebnisse des *Barbershop*-Benchmarks . . . 31
9 Skizze des Fähren-Beispiels . . . 31
10 Ergebnisse des Fähren-Benchmarks . . . 34

Programm-Listings

1 Beispielcode für ein Trait . . . 6
2 Beispiel für eine generische Funktion und eine generische Struktur . . . 7
3 Beispiel für Besitz . . . 8
4 Definition des Traits `Future` . . . 11
5 Implementation des Traits `Future` . . . 13
6 Konfigurationstrait . . . 17
7 Nutzung des Kofigurationstraits . . . 18
8 Interne Repräsentation der Kontrollvariablen . . . 18
9 Signatur der Funktion `wait_until()` . . . 18
10 Implementation der Funktion `wait_until()` . . . 21
11 Nutzung des Prozessmakros . . . 23
12 Singnatur von `recv_timeout()` . . . 24
13 Kontextwechsel in Rust . . . 26
14 Kontextwechsel in SLX . . . 26
15 *Barbershop* in SLX . . . 29
16 *Barbershop* in Rust . . . 30
17 Fähre in Rust . . . 32
18 Fähre in SLX . . . 33

Tabellenverzeichnis

1 Ergebnisse des Kontextwechsel-Benchmarks . . . 28
2 Ergebnisse des *Barbershop*-Benchmarks . . . 30
3 Ergebnisse des Fähren-Benchmarks . . . 33

1 Motivation

Mithilfe von Simulation können Einblicke in reale Systeme gewonnen werden um daraus neue Erkenntnisse zu gewinnen, mit denen beispielsweise Optimierungsmöglichkeiten für die Abläufe des Systems erkannt werden können. Dafür wird ein Modell des realen Systems erstellt und die dort ablaufenden Prozesse simuliert. Bei der Computersimulation wird die Simulation mithilfe eines Computerprogramms durchgeführt. Komplexe, externe Faktoren werden hierbei mithilfe zufälliger Komponenten modelliert.

1.1 Diskrete Simulation

Bei der diskreten Simulation treten Ereignisse in dem System nur zu diskreten Zeitpunkten auf [1]. Die Simulationszeit wird hierbei immer auf den Zeitpunkt des nächsten Ereignisses gesetzt. Bei der diskreten Simulation kann zwischen der ereignisorientierten und der prozessbasierten Simulation unterschieden werden. Bei der ereignisorientierten Simulation werden Ereignisse simuliert, die den neuen Zustand des Modells berechnen und im Anschluss an ihre Ausführung gegebenenfalls weitere Ereignisse in der Zukunft planen können. Es wird immer das als nächstes in der Zukunft stattfindende Ereignis zur Simulation ausgewählt und die Modellzeit auf den entsprechenden Zeitpunkt gesetzt. Die Ausführung der Aktionen eines Ereignisses verbraucht keine Modellzeit. Im Gegensatz dazu können bei der prozessbasierten Simulation Prozesse erstellt werden, die während ihrer Ausführung Modellzeit verbrauchen können. Beispielsweise kann eine Fähre, die Autos über einen Fluss transportiert, mit dem Lebenslauf *auf Autos warten - Überfahrt - Autos hinunterfahren lassen* als Prozess beschrieben werden. Jeder der Schritte benötigt eine bestimmte Modellzeit. Es kann eine dynamische Anzahl an Prozessen in der Simulation geben, das heißt die Anzahl der Prozesse ist nicht von Anfang an bekannt.

Weiterhin kann bei der prozessbasierten Simulation zwischen aktiven und passiven Klassen unterschieden werden [6]. Passive Klassen sind Klassen, die nicht von selbst aktiv werden, sondern nur von anderen Klassen benutzt werden. Ein Beispiel für eine passive Klasse ist ein Kanal, der nur von anderen Klassen dazu genutzt wird Nachrichten zu verschicken, aber nicht von selbst aktiv wird. Im Gegensatz dazu werden aktive Klassen von selbst aktiv, so wird zum Beispiel die Fähre, nachdem ihre Überfahrt beendet ist, automatisch aktiv und setzt ihren Lebenslauf fort.

Auf diese Weise können komplexere Systeme, bei denen eine Vielzahl von Ereignissen nötig wären, einfacher beschrieben werden.

1.2 Umfang eines Simulatorkerns

Eine Simulationsbibliothek beziehungsweise eine domänenspezifische Sprache für Simulation bietet die Möglichkeit, Prozesse zu erstellen und ihren Lebenslauf zu definieren. Der Simulatorkern bietet folgende Funktionalitäten um dieses Ziel zu erreichen:

1. Möglichkeit einen Prozess erstmalig zu aktivieren
2. Möglichkeit einen wartenden Prozess zu reaktivieren
3. Möglichkeit eine bestimmte Modellzeit zu warten und im Anschluss reaktiviert zu werden
4. Möglichkeit eine unbestimmte Modellzeit zu warten
5. Möglichkeit auf das Erreichen einer bestimmten Bedingung zu warten

1.3 Ziel der Arbeit

Mit der Sprache SLX (*Simulation Language with Extensibility*) existiert bereits eine domänenspezifische Programmiersprache für diskrete, prozessbasierte Simulation [5]. Sie wurde Mitte der 90er Jahre von der *Wolverine Software Cooperation* entwickelt und ist eine Simulationssprache, die als sehr performant gilt [2]. Ein Nachteil von SLX ist, dass sie nur unter Windows und somit nicht plattformübergreifend läuft, da der SLX-Compiler Windows-spezifischen Maschinencode erzeugt [5].

Ziel dieser Arbeit ist es, einen Simulatorkern in Rust zu entwickeln, der die Funktionalitäten aus Unterabschnitt 1.2 bietet. Da Rust grundsätzlich plattformunabhängig läuft, lässt sich diese Simulationsbibliothek auf vielen verschiedenen Systemen nutzen. Anschließend soll anhand ausgewählter Beispiele ein Benchmark zwischen einer Implementation in Rust und einer in SLX durchgeführt werden, um die Performanz zu vergleichen.

1.4 Aufbau der Arbeit

In Abschnitt 2 werden die Grundlagen von Rust erläutert, die zum Verständnis der folgenden Kapitel notwendig sind. Im dritten Kapitel wird der entwickelte Rust-Simulatorkern näher erläutert, im vierten Kapitel werden die Benchmarks vorgestellt. Im fünften Kapitel werden die Ergebnisse zusammengefasst und es gibt einen Ausblick auf weitergehende Problemstellungen.

2 Grundlagen

2.1 Besonderheiten von Rust

Rust ist eine quelloffene Programmiersprache, deren erste stabile Version es seit 2015 gibt. Sie wurde unter anderem mit dem Ziel entwickelt, speichersicher und performant zu sein. Im Folgenden werden die Besonderheiten von Rust erklärt, die für die Implementation des Simulatorkerns genutzt wurden.

```
trait ExampleTrait {
  fn example(self)->u32;
}

impl ExampleTrait for MyStruct {
  fn example(self)->u32 {return 1;}
}
```

Listing 1: Beispiel für einen Trait und die Implementation des Traits für eine Struktur. Jede Struktur, die `ExampleTrait` implementiert, muss die Funktion `fn example(self)->u32` anbieten.

2.1.1 Generische Funktionen und Strukturen

In diesem Abschnitt werden die Möglichkeiten erklärt, in Rust generische Strukturen und Funktionen zu erstellen und Anforderungen an die generischen Parameter zu stellen.

Traits Traits sind ähnlich zu Interfaces in Java, das heißt sie beschreiben eine gemeinsame Schnittstelle. Ein Trait kann beliebig viele Methoden fordern. Für alle Strukturen, die das Trait implementieren, müssen diese Methoden implementiert werden. Traits können außerdem beliebig viele assoziierte Typen haben. So hat zum Beispiel das Trait `Add` den assoziierten Typ `Output` der festlegt, welchen Typ das Ergebnis der Addition hat. Der konkrete Typ für jeden assoziierten Typen muss bei der Implementation des Traits für eine Struktur angegeben werden. In Listing 1ist ein Beispiel für die Definition eines Traits und die Implementation eines Traits für eine Struktur angegeben.

Generische Funktionen und Strukturen Eine Funktion ist generisch, wenn ihre Ein- oder Ausgabeparameter keinen konkreten Typ haben. An den Typ der Parameter können bestimmte Anforderungen gestellt werden indem gefordert wird, dass sie ein bestimmtes Trait implementieren. Zur Übersetzungszeit wird für jeden Ruf der Funktion mit unterschiedlichen konkreten Typen eine Funktionsdefinition mit konkreten Typen generiert. Das Programm verhält sich also genau so, als ob die Funktion für alle vorkommenden Typen separat definiert worden wäre. Somit entstehen für generische Funktionen keine zusätzlichen Laufzeitkosten. Analog gibt es Strukturen mit generischen Parametern.

Damit können Funktionen wie beispielsweise `example(example: ExampleTrait)` generisch für alle Typen implementiert werden, die das `ExampleTrait` aus dem vorigen Abschnitt implementieren. In Listing 2 ist ein Beispiel für eine generische Struktur und eine generische Funktion angegeben.

Aktive Klassen wurden in dieser Arbeit in Rust mithilfe von Strukturen mit einem zugehörigen Lebenslauf umgesetzt.

```rust
struct GenericStruct<T:Add<Output=T>> {
  item: T
}

fn generic_add<T: Add<Output=T>>(item1: T, item2: T) -> T {
  item1 + item2
}
```

Listing 2: Beispiel für eine generische Struktur und eine generische Funktion. Das `GenericStruct` besitzt einen generischen Parameter, an den die Anforderung gestellt wird, dass er das Trait `Add` implementiert und das Ergebnis der Addition ebenfalls vom Typ `T` ist. Die Funktion `generic_add()` kann mit allen Parametern gerufen werden, die das Trait `Add` implementieren. Der Rückgabewert muss vom selben Typ sein wie die Parameter.

dynamic dispatch Um ein beliebiges Objekt zu beschreiben, das ein bestimmtes Trait implementiert, ohne den Typ des Objekts festlegen zu müssen, bietet Rust das Schlüsselwort `dyn`. Damit kann die Auflösung einer konkreten Funktion zur Laufzeit durchgeführt werden. Dies wird zum Beispiel dafür benötigt, die korrekte Funktion eines Prozesslebenslaufs zur Laufzeit zu ermitteln, da heterogene Listen von Prozessen gespeichert werden müssen. So kann zum Beispiel ein beliebiges Objekt, dass das `ExampleTrait` implementiert, mit `dyn ExampleTrait` beschrieben werden.

2.1.2 Speichersicherheit

Speichersicherheit bedeutet, dass jede Dereferenzierung eines Zeigers immer gültig ist und auf Daten des richtigen Typs zeigt, sofern das Programm erfolgreich übersetzt werden konnte. Daraus folgen folgende Merkmale:

- keine Nullzeiger-Dereferenzierungen
- keine Dereferenzierungen auf bereits freigegebene Daten
- keine Nutzung von Daten nach deren Freigabe
- keine doppelte Freigabe von Daten
- keine Zugriffe außerhalb definierter Speicherbereiche
- kein unsynchronisierter gleichzeitiger Zugriff auf Daten. Das heißt keine Fehler durch Lesen inkonsistenter Daten wenn eine Änderung an den Daten wenn sie gelesen werden, angefangen, aber noch nicht abgeschlossen ist.

Alle Rust-Programme, die erfolgreich übersetzt werden können und in denen kein `unsafe` vorkommt, erfüllen die Bedingungen für Speichersicherheit. In solchen Blöcken ist der Programmierer dafür zuständig, dass die Regeln für Speichersicherheit eingehalten werden, es findet also keine Prüfung der Regeln durch den Compiler statt.
Eine Sprache, die die Eigenschaften von Speichersicherheit erfüllt, eignet sich besonders

```rust
let s1 = String::from("hello");
let s2 = s1;
println!("{}, world!", s1);
```

Listing 3: Das Programm kompiliert nicht, da in Zeile zwei `s2` zum Besitzer des Strings `"hello"` wird und der Bezeichner `s1` somit ungültig wird. In Zeile drei wird auf den ungültigen Bezeichner zugegriffen.

gut für Simulationen, da schwer erkennbare Fehler ausgeschlossen werden können. So ist es möglich, dass der fehlerhafte Zugriff auf bereits freigegebenen Speicher in vielen Fällen trotzdem funktioniert. Also könnte die Simulation in allen Testläufen fehlerfrei laufen, im eigentlichen Simulationslauf dann allerdings fehlerhaft sein. Auch die Simulationssprache SLX erfüllt diese Bedingungen.

Besitz und Referenz Besitz und Referenz sind die Konzepte, die dafür sorgen, dass Rust speichersicher ist. In Rust hat jeder Wert genau einen Besitzer. Die Variable wird freigegeben, sobald der Besitzer seinen Sichtbarkeitsbereich verlässt. Damit jede Variable immer nur genau einen Besitzer hat wird bei Zuweisungen im Allgemeinen die *Move*-Semantik verwendet, das heißt der Besitz wird übertragen und der andere Bezeichner wird ungültig. Diese *Move*-Semantik ist notwendig, damit klar ist, nach wessen Verlassen des Sichtbarkeitsbereiches der Speicherplatz freigegeben wird und er somit nicht mehrmals freigegeben werden kann. Einige Basisdatentypen wie `u32` oder `bool` verwenden die *Copy*-Semantik, das heißt sie werden bei jeder Zuweisung kopiert und keine der Variablen wird ungültig. In Listing 3 ist ein Beispiel angegeben das nicht kompiliert, da die Bedingungen für Besitz verletzt werden. Außerdem sind immer nur beliebig viele lesende oder eine schreibende Referenz auf eine Variable möglich um Fehler wie zum Beispiel ungültige Zeiger nach Vergrößerung eines Vektors, wie sie in C möglich sind, schon zur Übersetzungszeit zu erkennen.

Gültigkeitsbereiche In Rust annotiert der Compiler zur Übersetzungszeit die Gültigkeitsbereiche der Variablen um durch strukturelle Induktion zu beweisen, dass alle Variablen gültig sind. In manchen Fällen kann der Gültigkeitsbereich zur Übersetzungszeit nicht inferiert werden und muss somit explizit angegeben werden. In dieser Arbeit wird nur der explizit angegebene Gültigkeitsbereich `'static` genutzt der Referenzen beschreibt, die das ganze Programm über gültig sind. Alle anderen in dieser Arbeit genutzten Gültigkeitsbereiche können zur Übersetzungszeit inferiert werden.

Ausnahmen von den Regeln für Speichersicherheit Mit `unsafe` Blöcken können Anweisungen annotiert werden, bei denen der Compiler nicht garantieren kann, dass die oben genannten Bedingungen für Speichersicherheit erfüllt sind. Für alle Anweisungen, die nicht in einem `unsafe`-Block sind, garantiert der Compiler, dass der Code speichersicher ist, sofern alle Anweisungen in `unsafe` Blöcken die Bedingungen für Speichersicherheit erfüllen. Beispielsweise ist das Dereferenzieren eines Rohzeigers,

also eines Zeigers wie in C, `unsafe` da in diesem Fall nicht garantiert werden kann, dass gültige Daten dereferenziert werden. Dies wird benötigt, da in manchen Fällen der Compiler Speichersicherheit zwar nicht garantieren kann, der Programmierer aber trotzdem weiß, dass die Bedingungen dafür erfüllt sind.

Geteilter Schreibzugriff Um den Test, ob eine Variable nur maximal eine schreibende oder beliebig viele lesende Referenzen hat, erst zur Laufzeit und nicht schon zur Übersetzungszeit durchführen zu können, kann man `Cell<T>` für alle Typen, die die *Copy*-Semantik verwenden, beziehungsweise `RefCell<T>` für beliebige Typen verwenden. Dies ist unter anderem notwendig, da die Prozesse von verschiedenen Stellen des Programms aus geändert werden müssen und somit nur eine veränderliche Referenz nicht ausreicht. Rust zählt intern mit, wie viele lesende beziehungsweise schreibende Referenzen es auf ein Objekt gibt und es gibt einen Laufzeitfehler, wenn die Regeln verletzt werden, also beispielsweise wenn es noch mindestens eine lesende Referenz gibt und eine weitere schreibende angefordert wird.

Globale Variablen Da der Compiler bei globalen veränderlichen Variablen die Existenz mehrerer Threads nicht ausschließen kann, sind diese in Rust nicht erlaubt. Deswegen werden hier für globale Variablen thread-lokale Variablen genutzt. Das sind Variablen, die für jeden Thread einmal existieren, so dass sicher gestellt ist, dass immer nur ein Thread gleichzeitig auf diese Variable zugreifen kann. Weil bei Zugriff auf die thread-lokale Variable bestimmte Synchronisationen nötig sind die sicherstellen, dass die Bedingungen für Speichersicherheit eingehalten werden, ist ein solcher Zugriff mit gewissen Laufzeitkosten verbunden.

Referenzgezählte Zeiger `Rc<T>` - Zeiger in Rust bezeichnen referenzgezählte lesende Zeiger, das heißt ein solcher Zeiger zählt mit, wie viele Referenzen auf sein Objekt es insgesamt noch gibt und gibt das Objekt frei, sobald keine solche Referenz mehr existiert. Somit gibt es nicht mehr einen einzigen Besitzer dieser Variable, das heißt `Rc` ist dann sinnvoll, wenn zur Übersetzungszeit noch nicht gesagt werden kann, wer die Daten als Letztes noch benötigt. Dies wird beispielsweise für die Prozesse benötigt, da diese in verschiedenen Listen gespeichert werden können und nicht gesagt werden kann, wo sie als letztes benötigt werden.

`Weak<T>`-Zeiger sind ähnlich zu `Rc`-Zeigern, allerdings beschreiben sie nur Kenntnis über das Objekt, nicht Besitz. Das heißt ein Objekt wird freigegeben, sobald keine `Rc`-Zeiger mehr auf es existieren, es können allerdings noch beliebig viele `Weak`-Zeiger auf das Objekt existieren. Um von einem `Weak`-Zeiger aus lesend auf das Objekt zuzugreifen muss dieser erst in einen `Rc`-Zeiger umgewandelt werden. Diese Umwandlung schlägt zur Laufzeit fehl, falls das Objekt bereits freigegeben wurde, so dass kein fehlerhafter Speicherzugriff auf bereits freigegebenen Speicher vorkommt. `Rc<T>` implementiert außerdem das Trait `Deref`, das heißt es kann wie ein normaler Zeiger dereferenziert werden.

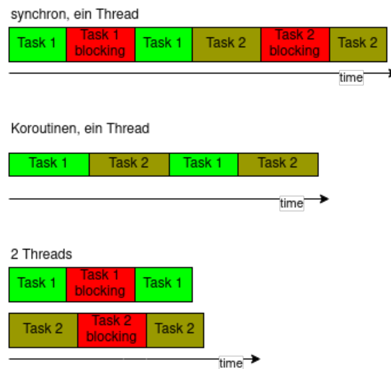


Abbildung 1: Bei einer synchronen Implementation mit einem Thread bleibt die Zeit, wenn einer von beiden blockiert, ungenutzt. Bei einer asynchronen Implementation mit einem Thread kann die Wartezeit einer Aufgabe für die andere Aufgabe genutzt werden. Der Zustand der wartenden Aufgabe muss in dieser Zeit gespeichert werden, damit diese im Anschluss korrekt fortgesetzt werden kann.

2.1.3 Koroutinen in Rust

Mithilfe von Koroutinen können Funktionen ihren Ablauf unterbrechen und zu einem späteren Zeitpunkt an der Stelle wieder aufnehmen, wo sie ihn zuvor unterbrochen hatten, wobei der Zustand der Funktion zum Zeitpunkt der Unterbrechung gespeichert wird. Für die Abarbeitung der Koroutinen wird also eine Zustandsmaschine benötigt, die sich den aktuellen Zustand der Funktionen merkt. In diesem Kapitel werden die Mechanismen in Rust erklärt die es ermöglichen, strukturierten Code zu schreiben, der zu Koroutinen kompiliert. Rust unterstützt seit Version **1.39** mithilfe von `async` und `await` eine Abstraktion für Koroutinen, die es ermöglicht, strukturierten Code zu schreiben, der dann zu einer Zustandsmaschine kompiliert. Es handelt sich bei dieser Abstraktion um eine sogenannte *Zero-Cost-Abstraction*, das heißt, dass der so generierte Code nicht langsamer ist als eine äquivalente von Hand geschriebene Zustandsmaschine. Im folgenden Abschnitt wird zunächst der Unterschied zu einer thread-basierten Herangehensweise und die Funktionsweise von mittels `async` und `await` generierter Koroutinen erklärt.

Unterschied zwischen Koroutinen und Threads Der Unterschied zwischen einer koroutinenbasierten und einer threadbasierten Herangehensweise ist in Abbildung 1 beschrieben.

Bei einer Implementation mit mehreren Threads kann jede Aufgabe in einem Thread erledigt werden, die Aufgaben werden also eventuell gleichzeitig erledigt. Außerdem erfolgt die Abgabe der Kontrolle präemptiv durch den Scheduler des Betriebssystems, ein Thread kann also nicht wissen, zu welchem Zeitpunkt die Kontrolle zu einem anderen Thread wechselt. Koroutinen hingegen sind kooperativ, das heißt eine Koroutine entscheidet selbst, wann sie die Kontrolle an eine andere Koroutine abgibt.

Beispielsweise könnte ein Server, der mithilfe einer Event-Schleife implementiert ist, sich für jeden Client den Zustand merken und immer einen Client, der gerade etwas

```
pub enum Poll<T> {
  Ready(T),
  Pending
}

pub trait Future {
  type Output;

  fn poll(self: Pin<&mut Self>, cx: &mut Context) -> Poll<Self::Output>;
}
```

Listing 4: Definition des Traits `Future` und der Enumeration `Poll<T>`. Das Trait `Future` fordert die Implementation der Funktion `Future::poll()` und die Definition des assoziierten Typs `Output`. Dieser Typ ist der Rückgabetyp der asynchronen Funktion. `Poll<T>` ist eine Enumeration, die anzeigt, ob bereits ein Ergebnis vorliegt. Diese Enumeration ist generisch für alle möglichen Rückgabetypen. Jeder Ruf von `Future::poll()` an einem Objekt, dass `Future` implementiert gibt entweder `Pending` zurück, falls noch kein Ergebnis vorliegt, oder `Ready(Self::Output)` falls ein Ergebnis vorliegt. `Self::Output` ist der Rückgabetyp derjenigen asynchronen Funktion, an der gerade `Future::poll()` gerufen wird. Der Parameter `cx` ist ein Kontext-Objekt, das Zugriff auf den Waker ermöglicht, der im Folgenden noch erklärt wird. Um eine Future aufzulösen, muss nun so lange `Future::poll()` an ihr gerufen werden, bis sie mit `Ready(Self::Output)` zurück kommt.

anfragt, bearbeiten. Im Fall mit mehreren Threads würde der Server für jeden Client einen eigenen Thread erstellen, der auf Anfragen des Clients wartet. In diesem Fall können also Anfragen echt gleichzeitig bearbeitet werden, was bei Verwendung einer Event-Schleife mit nur einem Thread nicht möglich ist.

Async Funktionen oder Blöcke können mit dem Schlüsselwort `async` annotiert werden. Solche Blöcke erstellen beim Ruf ein Objekt einer compilergenerierten Klasse, die das Trait `Future` implementiert. Alle lokalen Variablen der Funktion beziehungsweise des Blocks werden als Elemente in der generierten Struktur gespeichert. In Listing 4 sind die Definition des Traits `Future` und der Enumeration `Poll` angegeben, in Listing 5 wird eine beispielhafte Implementation des Traits angegeben.

Await `await` kann nur innerhalb von mit `async` annotierten Funktionen/ Blöcken gerufen werden und versucht eine Future aufzulösen. Jeder Ruf von `await` führt zu einem weiteren Zustand in der vom Compiler erzeugten Zustandsmaschine. In Abbildung 2 ist der Ablauf bei Ruf einer asynchronen Funktion mit einem `await` erklärt.

Executor Der Executor koordiniert die Wahl des Folgeprozesses und ruft so lange `Future::poll()` an den Futures, bis diese mit `Ready` zurückkehren. Da beim Ruf von `Future::poll()` ein Kontextobjekt übergeben wird, muss der Executor auch dieses Kontextobjekt erstellen. Mithilfe dieses Objekts kann dann im `Future::poll()` auf

```rust
struct SleepProcess {}

impl SleepProcess {
  async fn get_actions(self) {
    Sleep.await;
    //weiterer Lebenslauf
  }
}

fn main() {
  run(|_| async {
    let sleep = SleepProcess{};
    activate(sleep);
    //weitere Simulation
  })
}
```

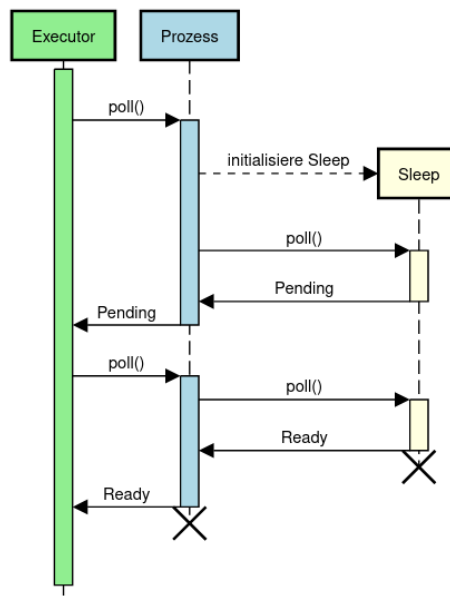


Abbildung 2: Beispiel für den Ablauf einer asynchronen Funktion mit einmaligem Ruf von `await`. Der Executor ruft zunächst `Future::poll()` an `SleepProcess`. Daraufhin wird das `get_actions()` von `SleepProcess` ausgeführt und das bereits bekannte `Sleep` initialisiert. An diesem `Sleep` wird ebenfalls `Future::poll()` gerufen. Beim ersten Ruf von `Future::poll()` kommt `Pending` zurück. Wenn der Executor erneut `Future::poll()` an diesem `SleepProcess` ruft, wird wieder `Future::poll()` am bereits initialisiertem `Sleep` gerufen, das diesmal mit `Ready` zurück kommt. Daraufhin kommt auch der `SleepProcess` mit `Ready` zurück und seine Abarbeitung ist beendet. Der weitere Lebenslauf von `SleepProcess` kann nun abgearbeitet werden.

```rust
struct Sleep {
  called: bool
}

impl Future for Sleep {
  type Output = ();

  fn poll(mut self: Pin<&mut Self>, _: &mut Context) -> Poll<Self::Output> {
    if self.called {
      Poll::Ready(())
    } else {
      self.called = true;
      Poll::Pending
    }
  }
}
```

Listing 5: Für die Struktur `Sleep` wird hier das Trait `Future` implementiert, wofür die Funktion `Future::poll()` implementiert werden muss. In diesem Fall kommt beim ersten Ruf von `Future::poll()` `Pending` zurück und das `Sleep` merkt sich in seinem Attribut `called`, dass an ihm schon einmal `Future::poll()` gerufen wurde. Beim zweiten Ruf kommt `Ready(())` zurück. Der assoziierte Typ `Output` ist somit das Einheitstupel.

den Waker zugegriffen werden, dessen Funktionsweise im folgenden Abschnitt erklärt wird. Da die Existenz eines Executors mit mehreren Threads nicht ausgeschlossen werden kann, müssen alle Futures sicher zwischen Threads versendet werden können. Dieser kann dann parallel an mehreren Futures `Future::poll()` rufen. Für die Simulation wird nur ein Executor mit einem Thread verwendet, da es ansonsten zu Synchronisationsproblemen in der Simulation wie in Unterabschnitt 2.2 beschrieben kommen könnte.

Waker Der Waker verbindet den Executor mit den Futures und wird vor allem dazu genutzt, dem Executor zu signalisieren, dass eine Future durch erneutes Rufen von `Future::poll()` an ihr Fortschritte machen könnte. Auf den eigenen Waker kann über das Kontextobjekt, das beim Ruf von `Future::poll()` übergeben wird, zugegriffen werden. Dieser kann dann an denjenigen weitergegeben werden der weiß wann die zugehörige Future Fortschritte gemacht hat und dann `wake()` an dem Waker ruft. Da dieses Objekt vom Executor erzeugt wird, können dort spezifische Informationen über den Executor übermittelt werden, über die die Kommunikation mit dem Executor funktioniert. Waker sind explizit kopierbar, da eine Future auf verschiedene Ressourcen zur gleichen Zeit warten kann.

2.2 Simulation nebenläufiger Prozesse

In dieser Lösung werden die verschiedenen Prozesse asynchron mithilfe von Koroutinen, die im vorigen Abschnitt erklärt wurden, umgesetzt. Im Folgenden wird erklärt, warum diese Herangehensweise für prozessbasierte Simulation sinnvoll ist.

Eine andere Möglichkeit wäre, die Prozesse mithilfe einer Event-Schleife zu implementieren. Allerdings speichert eine Event-Schleife keine Zustände, so dass die Prozesse nicht an der Stelle fortgesetzt werden können, wo sie zuletzt ihre Ausführung unterbrochen hatten. Deswegen eignet sich eine Event-Schleife nur für die in Unterabschnitt 1.1 erklärte ereignisorientierte Simulation, da die Ereignisse dort ihre Abarbeitung nicht unterbrechen können, also keine Simulationszeit verbrauchen können.

Eine weitere Möglichkeit nebenläufige Prozesse umzusetzen wäre, für jeden Prozess einen eigenen Thread zu erstellen. Mehrere Threads eignen sich allerdings nicht für Simulation, da die Simulationszeit zwischen den Threads sorgfältig synchronisiert werden müsste, damit kein Prozess versucht, sich oder einen anderen Prozess in der Vergangenheit zu terminieren. Durch die notwendigen Synchronisationen könnte meist nur ein Thread aktiv sein, da nur Prozesse die zur gleichen Zeit aktiv sind parallel abgearbeitet werden könnten. Außerdem entscheidet bei mehreren Threads der Scheduler des Betriebssystems, welcher Thread als nächstes aktiv ist. In einer Simulation soll allerdings immer der Prozess als nächstes aktiviert werden, dessen Aktivierung zum nächsten Zeitpunkt geplant ist. Diese Entscheidungslogik kennt der Scheduler des Betriebssystems nicht, so dass er sie nicht anwenden kann.

Koroutinen haben die Nachteile von Event-Schleifen und Programmen mit mehreren Threads nicht, so dass sie sich gut zur Umsetzung eines Simulatorkerns für diskrete Simulation eignen. Im folgenden Kapitel wird die Implementation des Simulatorkerns erklärt.

3 Implementation eines Simulatorkerns

In diesem Kapitel werden einige interessante Implementationsdetails der im Rahmen dieser Arbeit entwickelten Simulationsbibliothek *mk-simulation* vorgestellt und, falls vorhanden, mit den äquivalenten Funktionalitäten in SLX verglichen. Der gesamte Quellcode von *mk-simulation* befindet sich auf dem beigelegten USB-Stick.

3.1 Überblick

In Abbildung 3 ist ein überblicksartiges Klassendiagramm des Simulatorkerns dargestellt. Im Folgenden werden einige Aspekte des Simulatorkerns näher erläutert.

3.2 Executor und Waker

Der Executor verwaltet die Prozesse und ist somit verantwortlich für die Wahl des Folgeprozesses.
Da die Simulation in nur einem Thread abläuft kann aber immer nur die Aktionen

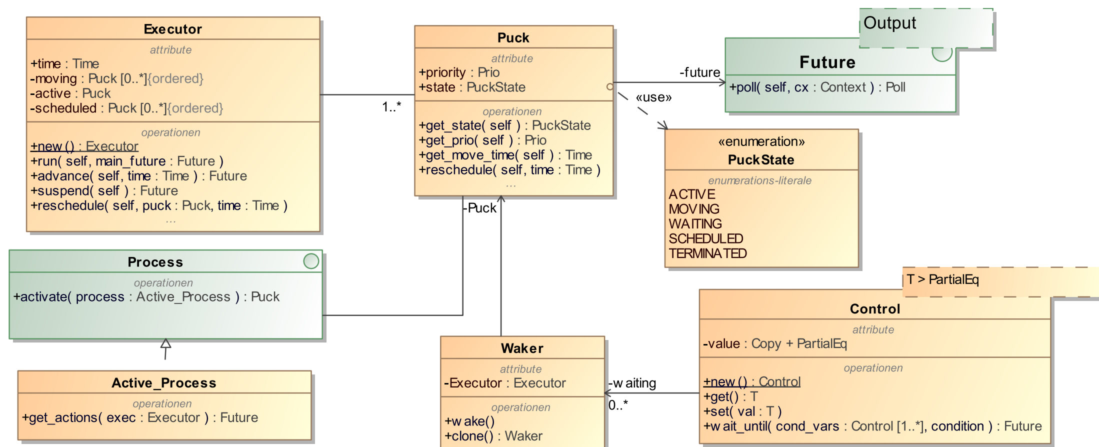


Abbildung 3: Klassendiagramm des Simulatorkerns. Die Datentypen für Zeit, Priorität und globale Daten sind generisch. Ein Puck [5] ist eine Klasse zum Speichern eines Prozesses, so werden dort der Lebenslauf des Prozesses sowie Verwaltungsinformationen wie beispielsweise die Priorität gespeichert. Jedem Prozess wird genau ein Puck zugeordnet. Jeder Puck ist immer in genau einem Zustand: `ACTIVE`, `MOVING`, `SCHEDULED`, `WAITING` oder `TERMINATED`. Jede aktive Klasse muss das Trait `Process` implementieren um aktiviert werden zu können. In *mk-simulation* kann jeder aktiven Klasse nur ein Puck zugeordnet werden, es liegt also eine 1:1-Beziehung vor. Jedem Puck können beliebig viele Waker zugeordnet werden. Der Waker eines Pucks terminiert diesen Puck beim Ruf von `waker.wake()` zur jetzigen Simulationszeit. Diese Waker werden beispielsweise von den Kontrollvariablen,also Variablen bei denen auf das Erreichen einer bestimmten Bedingung gewartet werden kann, genutzt um auf Änderung einer Kontrollvariable wartende Pucks zu reaktivieren.

eines Prozesses auf einmal ausgeführt werden. Dieser Puck wird im Attribut `active` des Executors gespeichert. Alle Prozesse, die zu einem bestimmten Zeitpunkt in der Zukunft reaktiviert werden sollen sind im Attribut `scheduled` gespeichert. Diese Prozesse werden in einer Prioritätswarteschlange gespeichert, mit dem Zeitpunkt zu dem sie reaktiviert werden sollen als Priorität. Als Datenstruktur für diese Warteschlange wurde ein Radix-Heap gewählt, da dieser besonders effizientes Einfügen von Elementen und Herausnehmen des Minimums ermöglicht, wenn jedes eingefügte Element größer ist als das zuletzt entnommene. Da sich ein Prozess nicht in der Vergangenheit terminieren kann, ist der Zeitpunkt eines neu terminierten Prozesses immer mindestens so groß wie der Zeitpunkt des zuletzt entnommenen Prozesses und diese Bedingung ist somit erfüllt. Außerdem ist der Radix-Heap ordnungserhaltend, was die Nachvollziehbarkeit der Simulation erleichtert. In dem Attribut `moving` des Executors werden alle Pucks gespeichert, die zur jetzigen Simulationszeit aktiv sein könnten. Die Prozesse werden ebenfalls in einem Radix-Heap gespeichert, da die Prozesse hier nicht monoton eingefügt werden, ist dieser Radix-Heap ineffizienter als der vorige, bei dem diese Eigenschaft erfüllt ist. Allerdings wurde trotzdem ein Radix-Heap gewählt, da dieser ordnungserhaltend ist. Sobald diese Warteschlange leer ist oder der Hauptprozess beendet ist, ist die Simulation beendet. Aus dem Zustand eines Pucks kann geschlussfolgert werden, wo im Executor er gespeichert ist. Im Executor können Daten gespeichert werden, auf die alle Prozesse durch den Ruf von `exec.get_global()` lesenden Zugriff erhalten können.

Der Executor kennt außerdem den Hauptprozess, da bei dessen Beendigung die Simulation ebenfalls beendet ist. Für Zeit, Priorität und globale Daten können beliebige Datentypen gewählt werden, die bestimmte Eigenschaften erfüllen. In Unterabschnitt 3.3 sind die geforderten Eigenschaften für diese Datentypen erklärt.
Um eine Simulation durchzuführen muss am Executor die Methode `run()` gerufen werden. Diese nimmt als Parameter den Hauptprozess, die dann beispielsweise alle nötigen Prozesse aktivieren kann. Dieses `run()` ist äquivalent zur `procedure main()` in SLX.

Mithilfe von `GetWaker` kann der Waker für den gerade aktiven Prozess des aktuellen Executors herausgefunden werden. Da Waker prinzipiell unabhängig vom Executor sind muss um sie zu speichern keine Kenntnis über den Executor vorliegen. `GetWaker` implementiert das Trait `Future` und greift in seinem `Future::poll()` über das übergebene Kontextobjekt auf den Waker des Executors zu und gibt diesen zurück. Durch den Ruf von `waker.wake()` kann der zum Waker gehörende Prozess in die Liste `moving` des Executors verschoben werden. Dafür speichert der Waker in seinen Attributen den zu aktivierenden Puck und eine Referenz auf den Executor. Da das Konzept von Futures in Rust für asynchrone Ein- und Ausgabeoperationen eingeführt wurde, müssen alle Attribute von Wakern eigentlich *thread-safe* sein, da sie bei Nutzung eines Executors mit mehreren Threads geschickt werden könnten. Hier speichert ein Waker beispielsweise einen Puck und kann somit nicht sicher zwischen Threads versendet werden, das heißt bei Verwendung mehrerer Threads kann es zu Fehlern kommen. Dies könnte bei einer Weiterentwicklung der Simulationsbibliothek noch verbessert werden.

```rust
pub struct Executor<C: Config> {...}

pub trait Config {
  type Time: Default + Copy + PartialOrd + Radix +
  Add<Output = Self::Time> + Debug;
  type Prio: Default + Copy + PartialOrd + Radix + Debug;
  type Data: Default;

  fn exec() -> &'static Executor<Self> where Self: std::marker::Sized;
}
```

Listing 6: Definition des Traits `Config`. Es fordert die Funktion `fn exec() -> &'static Executor<Self>`, die keine Parameter nimmt und eine Referenz mit der Lebenszeit `'static` auf den Executor zurück gibt. Dies wird durch eine thread-lokale Variable umgesetzt, in der der Executor gespeichert ist. Außerdem definiert es die Anforderungen an die Typen für Zeit, Priorität und globale Daten. Die hier genutzten Traits fordern folgende Eigenschaften:

- `Default`: Es existiert ein Standardwert für die Initialisierung
- `Copy`: Es darf keine Move-Semantik wie in Absatz 2.1.2 verwendet werden
- `PartialOrd`: Es existiert eine partielle Ordnung
- `Debug`: Es existiert eine Repräsentation für die Debug-Ausgabe
- `Radix`: Er kann als Schlüssel im Radix-Heap verwendet werden
- `Add`: Addierbarkeit, wobei das Ergebis der Addition vom Typ `Self::Time` ist.
- `Sized`: Die Größe des Typs ist zur Übersetzungszeit bekannt

3.3 Konfiguration des Executors

Der Executor ist generisch über dem Trait `Config`, damit eine Simulation mit verschiedenen Datentypen für Priorität, Zeit und die globalen Daten möglich ist. Um nicht alle Anforderungen an den Zeit-, Prioritäts- und globalen Datentyp immer explizit aufschreiben zu müssen, wurden diese in diesem Trait zusammengefasst. In Listing 6 sind die Definition des Konfigurationstraits sowie Erklärungen der dafür genutzten Traits angegeben. Dadurch haben alle Strukturen, die generisch über das Konfigurationstrait sind, Zugriff auf den Executor. Dies wird beispielsweise genutzt um einen Puck zu reaktivieren, da dafür auf die Liste der `moving` Pucks zugegriffen werden muss, die im Executor gespeichert ist. In Listing 7 ist ein Beispiel für die Nutzung des Konfigurationstraits angegeben.

```rust
pub fn reactivate(&self) {
  if self.get_state() != PuckState::WAITING {
    panic!("only waiting pucks can be reactivated");
  } else {
    self.set_moving();
    C::exec().get_moving_mut().push(self.get_prio(), self.clone());
  }
}
```

Listing 7: Da alle Pucks generisch über das Trait `Config` sind, kennen alle Pucks den Executor über die vom Trait geforderte Funktion `exec()`. Mithilfe von `C::exec()` kann auf den Executor zugegriffen werden.

```rust
struct RawControl<T: Copy + Debug + PartialEq> {
  control: Cell<T>,
  waiting: RefCell<Vec<Weak<Waker>>>,
}

pub struct Control<T: Copy + Debug + PartialEq>(Rc<RawControl<T>>);
```

Listing 8: Interne Repräsentation der Kontrollvariablen. `Control` besteht aus einem `Rc`-Zeiger auf `RawControl`, damit Kontrollvariablen kopierbar sind.

3.4 Kontrollvariablen

mk-simulation bietet Kontrollvariablen und die Funktion `async fn wait_until()` um auf das Eintreten einer bestimmten Bedingung zu warten. Im Folgenden werden erst die Kontrollvariablen und im Anschluss die Funktion `wait_until()` beschrieben.

```rust
pub async fn wait_until<F, C>(controls: C, condition: F)
  where F: Fn() -> bool, C: Controlled
```

Listing 9: Die Signatur der Funktion `wait_until()`, die für das Warten auf eine bestimmte Bedingung genutzt wird. Die Bedingung wird im Parameter `condition` mithilfe einer Funktion, die keine Parameter hat und einen Rückgabewert vom Typ `bool`, hat übergeben. Um alle in ihr vorkommenden Kontrollvariablen zu kennen, müssen diese im Parameter `controls` als Tupel von Kontrollvariablen oder Zeigern auf Kontrollvariablen angegeben werden. Das ist notwendig, da diese ansonsten nicht erkannt werden könnten. Das Trait `Controlled` wird von allen, auch heterogenen, Tupeln von Kontrollvariablen sowie von Tupeln von Referenzen auf Kontrollvariablen implementiert. Es bietet eine Methode, um einen Waker bei all diesen Kontrollvariablen zu registrieren. Bei Änderung des Wertes der Kontrollvariable werden alle wartenden Prozesse geweckt.

3.4.1 Kontrollvariablen in *mk-simulation*

Kontrollvariablen sind generisch für alle Datentypen implementiert, die vergleichbar sind, eine Debug-Ausgabe besitzen und die *Copy*-Semantik implementieren. Es werden also die Traits `Copy`, `Debug` und `PartialEq` gefordert. Das heißt, es können beispielsweise Kontrollvariablen vom Typ `bool` oder `i32` erstellt werden. Da Kontrollvariablen mithilfe von Wakern implementiert sind, benötigen sie keine Kenntnis über den Executor. Jede Kontrollvariable kennt die Waker aller auf ihre Änderung wartenden Prozesse, um diese im Falle einer Änderung reaktivieren zu können. Allerdings hat sie nur Kenntnis von den Wakern und keinen Besitz (umgesetzt durch einen `Weak`-Zeiger), da ein Prozess auf eine von mehreren Kontrollvariablen warten kann und bei Änderung der Kontrollvariable schon das `wait_until()` verlassen haben könnte. Damit dieser Prozess nicht fälschlicherweise erneut geweckt wird, hat die Kontrollvariable keinen Besitz von dem Waker. Falls der Prozess bereits reaktiviert wurde, schlägt das Umwandeln des `Weak`-Zeigers fehl, der Waker dieses Prozesses kann beim Reaktivieren der Prozesse also ignoriert werden. Damit Kontrollvariablen kopierbar sind, sind sie mithilfe eines `Rc`-Zeigers implementiert, so dass auf eine Kontrollvariable von verschiedenen Orten in der Simulation aus zugegriffen werden kann.

In Listing 8 wird die interne Repräsentation der Kontrollvariablen gezeigt. In Listing 9 ist die Signatur des `wait_until()` angegeben, mit dessen Hilfe auf das Erreichen einer bestimmten Bedingung gewartet werden kann, in Listing 10 ist die konkrete Implementation erklärt. In Abbildung 4 wird der Ablauf bei Ruf von `wait_until()` anhand eines Sequenzdiagramms erklärt.

3.4.2 Kontrollvariablen in SLX

In SLX müssen Kontrollvariablen mit dem Schlüsselwort `control` deklariert werden, also beispielsweise `control int j;`. Die Funktion `wait until(condition)` benötigt als Eingabeparameter dann noch eine boolesche Bedingung, die mindestens eine Kontrollvariable enthält. Anders als in *mk-simulation* genügt es hier im `wait until` nur die Bedingung anzugeben und es müssen nicht davor alle in der Bedingung genutzten Kontrollvariablen angegeben werden. Dies ist möglich da der SLX-Compiler weiß, welche Variablen Kontrollvariablen sind.

3.5 Prozesse und Pucks

Da sich sowohl SLX als auch *mk-simulation* für die prozessbasierte Simulation eignen, unterstützen beide das Erstellen von Prozessen. Prozesse sind aktive Klassen, die einen Lebenslauf besitzen, dessen Abarbeitung startet, sobald sie aktiviert werden. Der Zusammenhang zwischen Pucks und Prozessen wurde bereits in Unterabschnitt 3.1 erklärt.

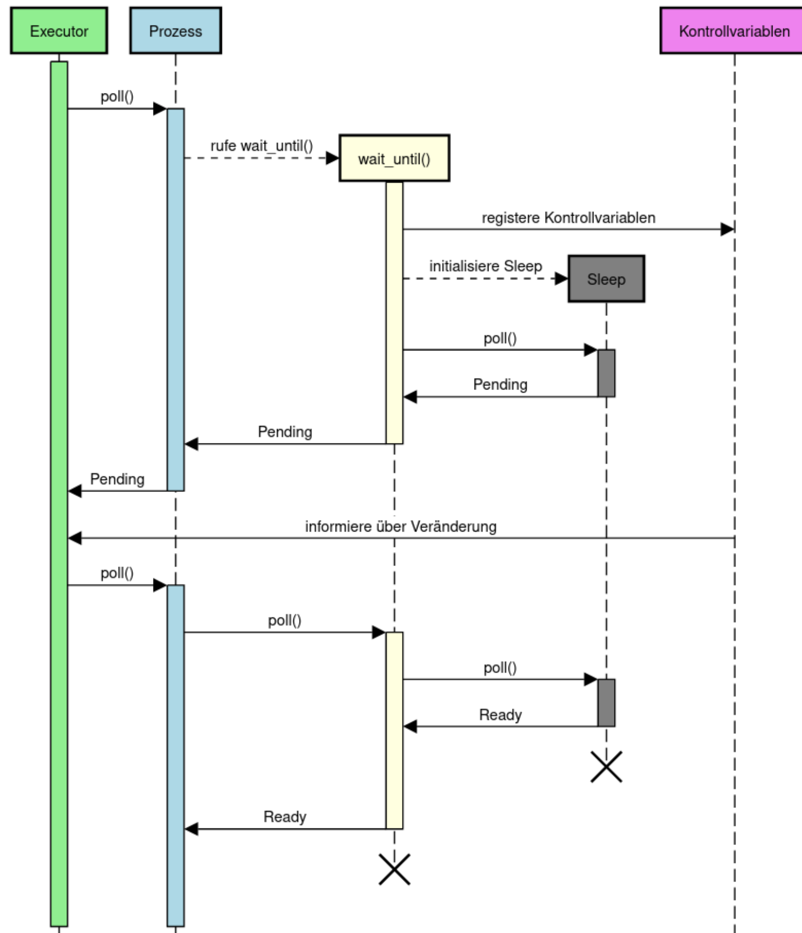


Abbildung 4: In diesem Sequenzdiagramm ist der Ablauf beschrieben, wenn ein Prozess `wait_until()` ruft und die Bedingung nach der ersten Änderung der Kontrollvariablen erfüllt ist. Bei Ruf des `wait_until()` wird der Waker des Prozesses bei den in der Bedingung vorkommenden Kontrollvariablen registriert. Sobald eine Kontrollvariable sich ändert informiert diese den Executor, der daraufhin `Future::poll()` am Prozess ruft. Da die Bedingung in diesem Beispiel nach der ersten Änderung erfüllt ist, kommt das `wait_until()` mit `Ready` zurück und der Prozess kann mit der Abarbeitung seines Lebenslaufes fortfahren.

```
1  pub async fn wait_until<F, C>(controls: C, condition: F)
2    where F: Fn() -> bool, C: Controlled
3  {
4    if condition() {
5      return;
6    }
7    let my_waker = Rc::new(GetWaker.await);
8
9    while !condition() {
10     controls.register(Rc::downgrade(&my_waker));
11     sleep().await;
12   }
13 }
```

Listing 10: Implementation der Funktion `wait_until()`. Mithilfe von `GetWaker.await` in Zeile sieben wird auf den eigenen Waker zugegriffen und dieser wird in die Wartelisten aller Kontrollvariablen eingetragen, auf deren Änderung gewartet wird. Jede Kontrollvariable kennt also den Waker aller Prozesse, die auf eine Bedingung warten, bei der sie beteiligt ist. Bei Änderung ihres Wertes weckt sie alle diese Prozesse, die daraufhin überprüfen, ob ihre Bedingung bereits eingetreten ist. Falls das nicht der Fall ist, rufen die Prozesse erneut `sleep().await`, um auf eine erneute Änderung zu warten. Nach Erreichen der Bedingung muss der Waker nicht aus diesen Listen entfernt werden, da nach Freigabe von `my_waker` das Umwandeln der `Weak`-Zeiger in den Kontrollvariablen fehlschlagen wird, weil der zugehörige besitzende Zeiger dann freigegeben wurde.

3.5.1 Prozesse und Pucks in SLX

In SLX können aktive und passive Klassen definiert werden. Jede aktive Klasse hat die Methode `actions()`, welche ihren Lebenslauf beschreibt. Alle aktiven Klassen können mithilfe von `activate &active_class` aktiviert werden. Jeder aktiven Klasse ist mindestens ein Puck zugeordnet. Durch Nutzung der Methode `fork` können jedoch mehrere Pucks einer aktiven Klasse zugeordnet werden, anders als in *mk-simulation* ist in SLX also ein 1:N Verhältnis zwischen Prozessen und Pucks möglich [5]. Dieses 1:N Verhältnis zwischen Prozessen und Pucks wird auch als Intra-Prozess-Parallelität bezeichnet.

3.5.2 Pucks in *mk-simulation*

Anders als in SLX ist in *mk-simulation* die Zuordnung von Pucks zu Prozessen eine 1:1-Zuordnung, das heißt jedem Prozess ist nur ein Puck zugeordnet, Intra-Prozess-Parallelität ist dementsprechend nicht möglich. Jeder Puck kennt die ihm zugeordnete Future, die seinen Lebenslauf beschreibt, sowie Verwaltungsinformationen wie zum Beispiel seine Priorität. Um die Pucks in einer homogenen Liste speichern zu können, wird der konkrete Typ des jeweiligen Lebenslaufs mittels *dynamic dispatch* ermittelt. Da Pucks intern mithilfe eines `Rc`-Zeigers implementiert sind implementieren sie `Clone`, sie können also explizit kopiert werden.

3.5.3 Prozesse in *mk-simulation*

Um einen Prozess in *mk-simulation* zu erstellen, muss zuerst eine aktive Klasse erstellt werden und diese dann aktiviert werden. Alle aktiven Klassen müssen das Trait `Process` implementieren, das eine Funktion fordert, die die Klasse aktiviert und den erstellten Puck zurück gibt. Zur einfacheren Implementation dieses Traits bietet die Rust-Bibliothek ein Makro, das `Process` automatisch für die angegebene Klasse implementiert, vorausgesetzt sie besitzt eine Funktion, die ihren Lebenslauf beschreibt. Um die Ausführung des Lebenslaufs einer aktiven Klasse zu starten, muss `activate()` an ihr gerufen werden. Damit der Executor bekannt ist, muss der konkrete Typ des Rückgabewerts angegeben werden, also beispielsweise `let _:Puck<Sim> = Active.activate()`. In Listing 11 wird eine beispielhafte Nutzung des Makros gezeigt.

3.6 Kanäle

Zur Kommunikation zwischen Prozessen wurden Kanäle implementiert. Da die Kanäle mithilfe von Kontrollvariablen implementiert wurden, wird für den Ruf ihrer Methoden, bis auf eine in diesem Abschnitt erklärte Ausnahme, keine Kenntnis des Executors benötigt. Die Nutzung dieser Kanäle ist analog zur Nutzung der MPMC (*Multiple-Producer-Multiple-Consumer*)-Kanäle für die Kommunikation zwischen unterschiedlichen Threads aus der Rust-Bibliothek **crossbeam** [3].
In Abbildung 5 ist ein Klassendiagramm für die Kanäle dargestellt. Um einen Kanal zu erstellen,kann `let (rcv, send) = Channel::new()` für einen unbeschränkten Kanal

```rust
#[derive(Process)]
struct Active {}

impl Active {
  async fn get_actions(self) {
    //Lebenslauf von Active hier
  }
}
```

Listing 11: Beispielhafte Nutzung des Prozessmakros. `Active` ist die aktive Klasse, deren Lebenslauf in `get_actions()` beschrieben wird.

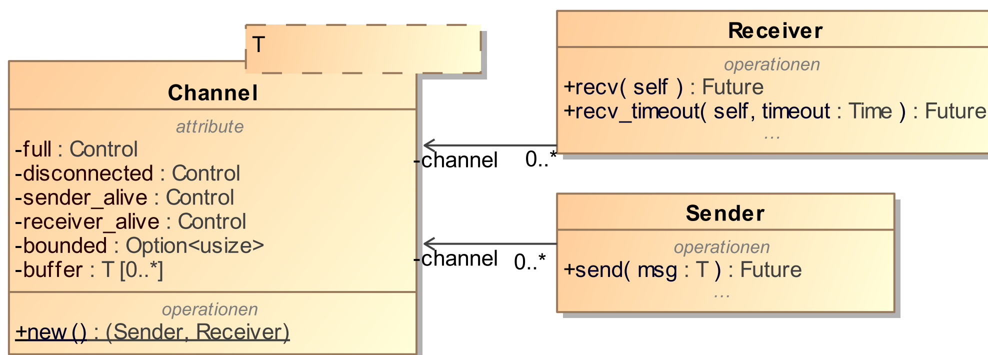


Abbildung 5: Klassendiagramm der Kanäle. Jeder Kanal hat ein Ende zum Empfangen von Nachrichten (`Receiver<T>`) und eines zum Senden von Nachrichten (`Sender<T>`). Es können Kanäle für Daten beliebigen Typs erstellt werden.

```rust
async fn recv_timeout<C: Config + 'static>(&self, timeout: C::Time)
    -> Result<T, ReceiveError>
```

Listing 12: Signatur der Funktion `recv_timeout()`. Weil ein neuer Prozess aktiviert werden muss, muss die Konfiguration bekannt sein.

gerufen werden. Da es sowohl mehrere Empfänger als auch mehrere Sender geben kann, implementieren sowohl das sendende als auch das empfangende Ende eines Kanals das Trait `Clone`. Mit diesen beiden Enden kann analog wie bei der Kommunikation zwischen Threads gearbeitet werden. Sobald kein Sender oder kein Empfänger mehr existiert, ist der Kanal unterbrochen. Der Versuch Daten zu senden kommt dann mit einem Fehler zurück. Wenn noch Nachrichten in dem Kanal sind können diese allerdings weiterhin empfangen werden. Der Versuch, Daten aus einem Kanal, der unterbrochen und leer ist, zu empfangen, schlägt fehl, da nie wieder Daten an diesem Kanal anliegen können. Intern sind die Kanäle mithilfe von Kontrollvariablen implementiert. Beim Empfangen einer Nachricht wird darauf gewartet, dass entweder alle sendenden Enden freigegeben wurden, und somit nie wieder jemand eine Nachricht in den Kanal senden kann, oder dass eine Nachricht anliegt. Für das Senden einer Nachricht werden die Kontrollvariablen nur im Falle eines beschränkten Kanals benötigt, da bei einem unbeschränkten Kanal die Nachricht entweder direkt gesendet werden kann oder der Kanal unterbrochen ist und das Senden direkt mit einem Fehler zurück kommt. Das Senden in einen beschränkten Kanal wartet darauf, dass die Kontrollvariable, die angibt ob der Kanal voll ist, den Wert `False` hat.

Für die Implementation des Fähren-Beispiels wurde ein `recv_timeout(timeout)` benötigt, das ist eine Funktion, die auf Erhalt einer Nachricht wartet, aber spätestens nach einer bestimmten Zeit mit einer Fehlermeldung zurück kommt. In Listing 12 ist die Signatur dieser Funktion angegeben. Hierfür wird ein neuer Prozess initialisiert und aktiviert, dessen einzige Aufgabe es ist, nach `timeout` Zeit eine Kontrollvariable von `false` auf `true` zu setzen. Nun wird mithilfe von `wait_until()` entweder auf die Bedingungen zum Erhalten einer Nachricht oder auf das Umschalten der Kontrollvariable gewartet. Falls innerhalb der vorgegebenen Zeit eine Nachricht gelesen werden kann kommt `Ok(msg)` zurück, ansonsten `ReceiveError::Timeout`. Da zur Aktivierung eines neuen Prozesses der Executor bekannt sein muss, ist diese Methode als einzige des Kanals abhängig von der Konfiguration des Executors.

4 Benchmarks

Um *mk-simulation* und SLX zu vergleichen wurden drei Benchmarks mit bereits aus der Literatur bekannten Szenarien ([2], [7]) durchgeführt. Einmal anhand eines einfachen Kontextwechsel-Beispiels bei dem sich zwei Prozesse immer gegenseitig die Kontrolle geben. Die anderen beiden an etwas komplexeren Beispielen, um die Performanz bei Nutzung von Synchronisationsstrukturen zu vergleichen. Im zweiten Beispiel wird ein *Barbershop* simuliert, bei dem zu zufälligen Zeiten Kunden ankommen, im anderen eine

Fähre, die zwischen zwei Häfen pendelt. In den beiden komplexeren Szenarien wurden die Benchmarks mit Zufallszahlen durchgeführt, da in realen Simulationen fast immer Zufallszahlen verwendet werden. Alle Benchmarks wurden jeweils mit unterschiedlichen Endzeitpunkten durchgeführt.
Sowohl für Rust als auch für SLX wurden die Benchmarks mithilfe der Benchmarking-Bibliothek **criterion** [4] durchgeführt. Diese Bibliothek bietet einen ausgereiften Benchmarking-Mechanismus, inklusive der Berechnung statistischer Werte und dem Erstellen von Diagrammen. In den Tabellen wurde der Median der Laufzeit angegeben, da dieser stabiler gegen Ausreißer als das arithmetische Mittel ist. Jedoch ist der Unterschied zur durchschnittlichen Laufzeit klein. Die Diagramme in dieser Arbeit wurden aus dem **criterion** Report entnommen.

Alle Benchmarks wurden auf einem Rechner mit Windows 10, einem Intel Pentium 5405U Prozessor mit 2 Kernen und 8 GB RAM ausgeführt. Alle Beispiele wurde in Rust mit rustc in Version 1.49.0, mit aktivierter Link-Time-Optimierung und dem höchsten Optimierungslevel 3 kompiliert. Für SLX wurde die Version 2.3 der 32-bit Studentenversion genutzt mit Patchlevel AN027 genutzt.

Folgende Einschränkungen könnten die Aussagekraft der Benchmarks beeinflussen:

- Für SLX konnte nur die Studentenversion genutzt werden, die etwa 30% langsamer ist als die Vollversion. Außerdem sind in der Studentenversion keine beliebig langen Benchmarks möglich, da nur eine gewisse Anzahl Pucks erstellt werden dürfen.
- Für das Fähren- und *Barbershop*-Beispiel wurden Pseudozufallszahlen verwendet. Der Algorithmus zum Berechnen dieser Zahlen unterscheidet sich in Rust und SLX.
- Leider stand kein schneller Rechner für die Benchmarks zur Verfügung.

Im Folgenden werden die drei Benchmark-Szenarien näher erläutert.

4.1 Kontextwechsel

In diesem Beispiel wurde die Performance bei Kontextwechseln gemessen. Hierfür wurde eine Simulation mit zwei Prozessen verwendet, die sich gegenseitig die Kontrolle geben.

Rust-Programm

In Rust wurden zwei Instanzen der aktiven Klasse `Counter` erstellt, deren Lebenslauf in Listing 13 beschrieben ist.

SLX-Programm

In Listing 14 ist der Lebenslauf der aktiven Klasse `Counter` in SLX dargestellt.

```
impl Counter {
  async fn get_actions(mut self) {
    let exec = counter::exec();
    while self.count > 0 {
      self.count -= 1;
      exec.suspend().await;
    }
  }
}
```

Listing 13: Der Lebenslauf eines `Counter`-Objektes. Der Executor wird einmalig in Zeile drei erfragt und gespeichert, um in der Schleife nicht bei jedem Schleifendurchlauf auf die thread-lokale Variable zugreifen zu müssen. Für einen Simulationslauf werden zwei Counter mit dem selben Wert für ihr Element `count` erstellt. Beide verringern den Wert von `count` um eines und rufen im Anschluss `exec.suspend().await`, was den eigenen Prozess zur jetzigen Simulationszeit aber hinter allen anderen Prozessen, die zu dieser Zeit terminiert sind, terminiert. Die Simulation ist beendet, sobald die Variable `count` in beiden Countern den Wert 0 erreicht.

```
class Counter(int _count) {
  int count = _count;
  actions {
    while (count > 0) {
      count--;
      yield;
    }
  }
}
```

Listing 14: Der Lebenslauf der aktiven Klassen `Counter` in SLX. Die Variable `count` wird in jedem Schleifendurchlauf um eins verringert und im Anschluss wird die Kontrolle an das andere `Counter` Objekt übergeben. Die Simulation ist beendet, sobald `count` den Wert 0 erreicht.

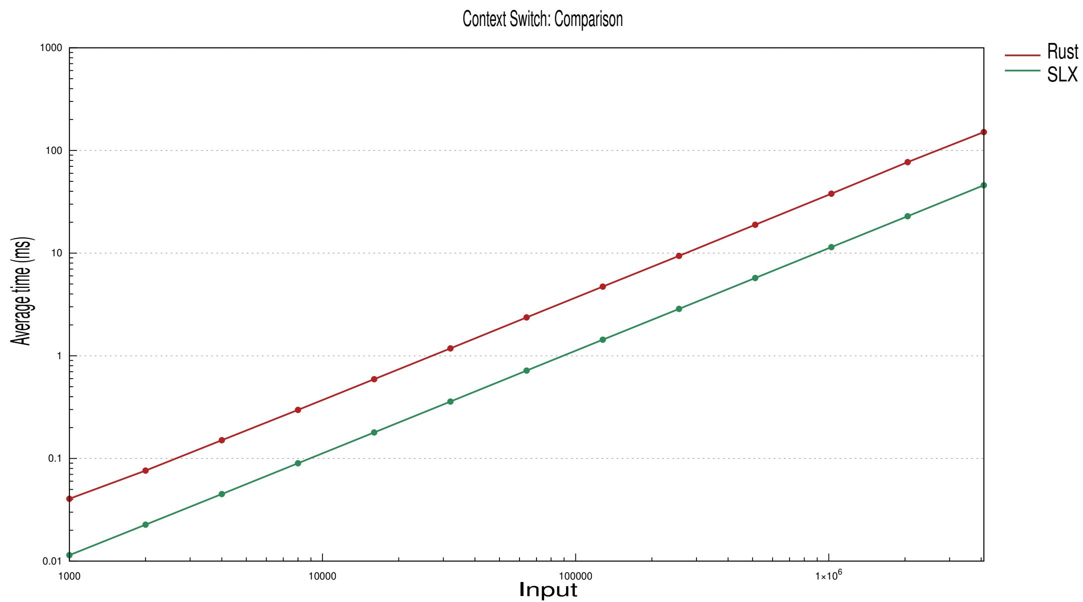


Abbildung 6: Grafische Repräsentation der Ergebnisse des Benchmarks.

Ergebnisse

In Abbildung 6 ist eine grafische Repräsentation der Ergebnisse des Benchmarks zu sehen und in Tabelle 1 eine tabellarische Repräsentation der Ergebnisse. Wie aus den Ergebnissen ablesbar ist, ist die Rust-Implementation mit 99% Wahrscheinlichkeit etwa um den Faktor $3,3$ langsamer als die SLX-Implementation.

4.2 *Barbershop*

Als zweites Beispiel wurde das *Barbershop*-Beispiel implementiert [7], in Abbildung 7 sind das Beispiel sowie der generische Ablauf zur Erstellung eines Simulationsprogramm erklärt.

Zur Implementation dieses Beispiels wurde eine *Facility* verwendet, eine Synchronisationsstruktur, die zwei Operationen bietet.

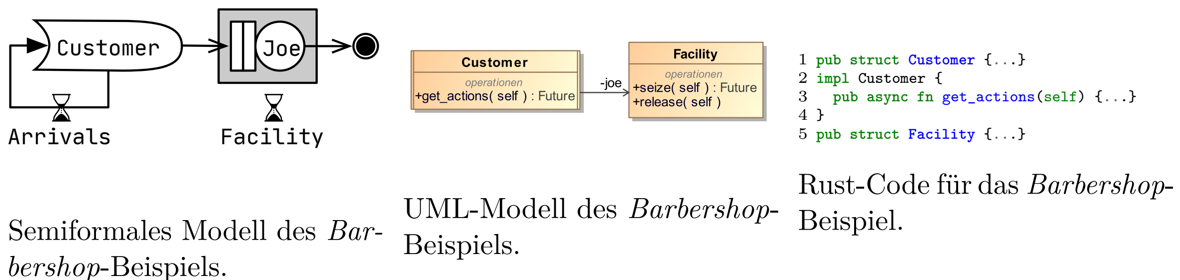
1. Man kann Zugriff auf die *Facility* anfordern. Dieser Ruf kehrt zurück, sobald der Zugriff erhalten wurde.

2. Man kann seinen Zugriff zurück geben. Daraufhin kann der nächste Zugriff auf die *Facility* erhalten.

Die *Facility* gibt den auf Zugriff wartenden Prozessen nacheinander Zugriff auf sich. Der Barber wurde als *Facility* implementiert, da immer nur ein Kunde gleichzeitig bedient werden kann (also Zugriff auf die *Facility* hat) und im Anschluss seinen Zugriff zurück gibt. Einzig die Kunden sind in diesem Beispiel durch aktive Klassen dargestellt, da die *Facility* eine passive Klasse ist.

Eingabe	Rust (Median)	SLX (Median)
1000	[39,643 , 39,663 , 39,742] us	[11,407 , 11,415 , 11,425] us
2000	[76,081 , 76,137 , 76,211] us	[22,616 , 22,632 , 22,647] us
4000	[150,20 , 150,26 , 150,40] us	[44,899 , 44,952 , 44,994] us
8000	[296,84 , 297,10 , 297,22] us	[89,618 , 89,697 , 89,792] us
16000	[589,15 , 589,52 , 590,06] us	[178,76 , 178,91 , 179,14] us
32000	[1,1790 , 1,1796 , 1,1803] ms	[357,37 , 357,76 , 358,07] us
64000	[2,3631 , 2,3659 , 2,3668] ms	[714,65 , 715,29 , 715,82] us
128000	[4,7052 , 4,7090 , 4,7135] ms	[1,4289 , 1,4300 , 1,4308] ms
256000	[9.3830 , 9.3908 , 9.4001] ms	[2.8565 , 2.8586 , 2.8607] ms
512000	[18.866 , 18.888 , 18.903] ms	[5.7106 , 5.7146 , 5.7195] ms
1024000	[37.802 , 37.864 , 37.921] ms	[11.427 , 11.435 , 11.445] ms
2048000	[75.811 , 76.668 , 77.066] ms	[22.855 , 22.875 , 22.889] ms
4096000	[149.96 , 150.17 , 150.83] ms	[45.712 , 45.749 , 45.788] ms

Tabelle 1: Ergebnisse des Kontextwechsel-Benchmarks inklusive des 99% Konfidenzintervalls für jeden Versuch.



Semiformales Modell des *Barbershop*-Beispiels.

UML-Modell des *Barbershop*-Beispiels.

Rust-Code für das *Barbershop*-Beispiel.

Abbildung 7: Bei der Durchführung einer Simulation wird mit einem semiformalen Beispiel begonnen, etwa einer Skizze des Vorgangs. Anschließend wird dieses Beispiel in ein formales Modell überführt, welches in Rust beziehungsweise SLX-Code überführt werden kann. Hier wird ein *Barbershop* simuliert, an dem zu zufälligen Zeiten Kunden ankommen. Der Barber, *Joe*, bedient die Kunden in der Reihenfolge ihrer Ankunft und benötigt pro Kunden zufällig viel Zeit. Nachdem die Simulationszeit einen bestimmten Wert überschritten hat kommen keine weiteren Kunden an, es werden aber die noch Wartenden bedient. Folgende Parameter für die Simulation wurden in dieser Arbeit verwendet:

- Ankunftszeiten der Kunden: gleichverteilt zwischen 12 und 24 Zeiteinheiten
- Dauer zum Bedienen eines Kunden: gleichverteilt zwischen 12 und 18 Zeiteinheiten

SLX-Programm

Es werde,n bis die Simulationszeit einen gewissen Wert erreicht hat, neue Kunden erstellt. Diese Kunden warten, bis sie Zugriff auf die *Facility* erhalten können, behalten diesen Zugrif für eine bestimmte Zeit und geben ihn danach zurück. In Listing 15 ist die Hauptschleife des SLX-Programms angegeben.

```
procedure run(double stop_time) {
  fork {
    forever {
      advance rv_uniform(rng_a, 12.0, 24.0);
      if (time >= stop_time) terminate;
      activate new Customer;
    }
  }
  advance stop_time;
   wait until FNU(joe);
}
```

Listing 15: Der Lebenslauf der Hauptschleife des *Barbershop*-Beispiels in SLX. `fork` erstellt einen neuen Puck, so dass die `forever`-Schleife in einem eigenen Puck ausgeführt wird.

Rust-Programm

Wie beim SLX-Programm werden auch hier bis zu einer bestimmten Simulationszeit neue Kunden erstellt. Da *mk-simulation* ein `fork` bietet wurde ein weiterer aktiver Prozess `CustomerGenerator` erstellt, der die neuen Kunden erstellt und aktiviert. In Listing 16 wird der `CustomerGenerator` erklärt.

Ergebnisse

In Abbildung 8 ist eine grafische Repräsentation der Ergebnisse des Benchmarks zu sehen und in Tabelle 2 eine tabellarische Repräsentation der Ergebnisse. Wie aus den Ergebnissen ablesbar ist, ist die Rust-Implementation mit 99% Wahrscheinlichkeit bei großen Eingaben etwa um den Faktor $3,5$ langsamer als die SLX-Implementation, jedoch ist der Faktor bei kleinen Eingaben geringer. Dies könnte daran liegen, dass die Initialisierung des SLX-Programms länger dauert als die des Rust-Programms, dieser Unterschied aber bei großen Eingaben keine Bedeutung mehr spielt.

4.3 Fähre

Das Fähren-Beispiel ist ein komplexeres Beispiel zum Vergleichen der Performance zwischen *mk-simulation* und SLX. In Abbildung 9 ist eine Skizze dieses Beispiels dargestellt sowie der Ablauf der Simulation erklärt. Damit das Rust- und das SLX-Programm vergleichbar sind ist die Struktur beider Implementationen gleich.

Rust-Programm

In Listing 17 ist die Hauptschleife des Fähren-Beispiels in Rust zu sehen. Die Häfen wurden durch Kanäle implementiert, in die vom Produzenten zu zufälligen Zeiten Autos gesendet werden. Die Fähre nutzt die Funktion `recv_timeout()` um entweder ein Auto auffahren zu lassen oder nach `timeout` Zeit abzulegen.

```rust
impl CustomerGenerator {
  async fn get_actions(self) {
    let exec = barbershop::exec();
    let mut rng_a = SmallRng::seed_from_u64(SEED_A);
    let mut rng_s = SmallRng::seed_from_u64(SEED_S);
    let dist = Uniform::new(12.0, 24.0);
    loop {
      exec.advance(rng_a.sample(dist)).await;
      if exec.now() >= self.stop_time { return; }
       let customer = Customer {
         joe: self.joe.clone(),
         rv: self.rv.clone(),
         rng: SmallRng::from_seed(rng_s.gen())
       };
       let _ : Puck<Simulator> = customer.activate();
     }
   }
 }
```

Listing 16: Der Lebenslauf des `CustomerGenerator`. Die `loop`-Schleife hier ist äquivalent zur `forever`-Schleife im SLX-Programm. In Zeile 3 wird der Executor in einer Variablen gespeichert, damit nicht in jedem Schleifendurchlauf auf die thread-lokale Variable zugegriffen werden muss. Da *mk-simulation* kein `fork` bietet werden alle Kunden-Prozesse in Zeile 10 erstellt und in Zeile 15 aktiviert. Da beim Aktivieren eines Prozesses der Executor bekannt sein muss, wird in Zeile 15 angegeben, dass die Konfiguration `Simulator` genutzt wird. Diese kennt wie in Abschnitt Unterabschnitt 3.3 beschrieben den Executor.

Eingabe	Rust (Median)	SLX (Median)
1000	[51,008 , 51,066 , 51,117] us	[43,116 , 43,183 , 43,291] us
2000	[99,453 , 99,575 , 99,687] us	[55,719 , 55,816 , 55,891] us
4000	[189,53 , 189,76 , 190,08] us	[81,671 , 81,808 , 81,899] us
8000	[372,65 , 373,00 , 373,53] us	[133,07 , 133,26 , 133,43] us
16000	[719,05 , 721,41 , 723,82] us	[235,24 , 235,60 , 236,09] us
32000	[1,4212 , 1,4222 , 1,4235] ms	[439,54 , 439,99 , 440,85] us
64000	[2,8633 , 2,8728 , 2,8833] ms	[849,21 , 849,85 , 851,08] us
128000	[5,6529 , 5,6613 , 5,6833] ms	[1,6684 , 1,6699 , 1,6732] ms
256000	[11,254 , 11,277 , 11,302] ms	[3,2888 , 3,2919 , 3,2964] ms
512000	[22,530 , 22,578 , 22,626] ms	[6,5406 , 6,5488 , 6,5572] ms
1024000	[44,997 , 45,111 , 45,194] ms	[13,061 , 13,068 , 13,081] ms

Tabelle 2: Ergebnisse des Benchmarks des *Barbershop*-Beispiels inklusive des 99% Konfidenzintervalls für jeden Versuch.

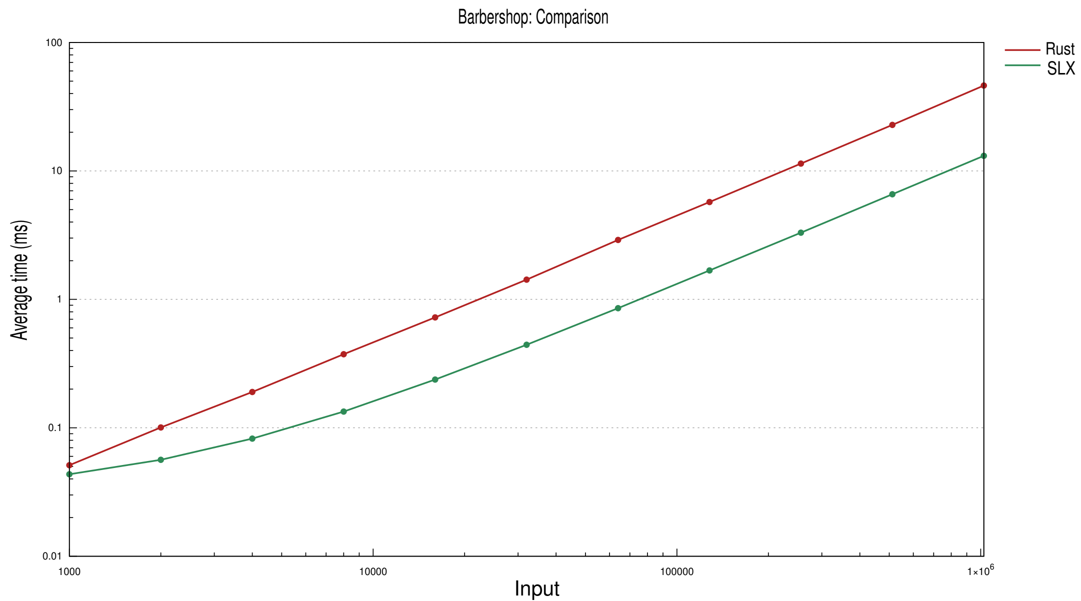


Abbildung 8: Grafische Repräsentation der Ergebnisse des Benchmarks des *Barbershop*-Beispiels.

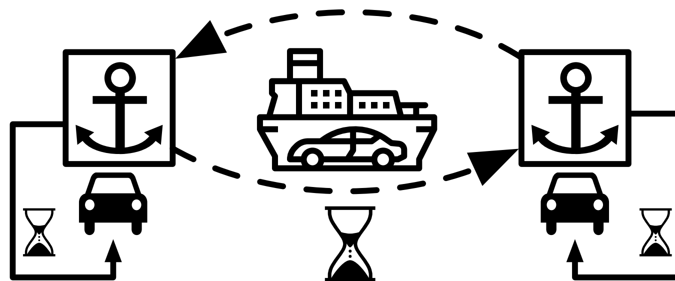


Abbildung 9: Skizze der Simulation des Fähren-Beispiels. Hierbei wird eine Autofähre simuliert, die zwischen zwei Häfen pendelt, an denen zu zufälligen Zeiten Autos ankommen. Sie legt ab, wenn sie entweder voll beladen ist oder eine gewisse Zeit lang kein weiteres Auto ankommt. Die Auf- und Abfahrt der Autos benötigt zufällig viel Zeit. Für die Überfahrt wird eine konstante Zeit benötigt. Folgende Parameter wurden für die Simulation verwendet:

- Dauer der Überfahrt: $7,5$ Zeiteinheiten
- Timeout der Fähre zur Abfahrt: 5 Zeiteinheiten
- Kapazität der Fähre: 5 Autos
- Dauer der Auffahrt eines Autos auf die Fähre: gleichverteilt zwischen $0,25$ und $0,5$ Zeiteinheiten
- Zeit zwischen der Ankunft von zwei Autos an der Fähre: gleichverteilt zwischen 11 und 15 Zeiteinheiten

```rust
loop {
  if let Ok(car) = self.ports[self.cur_port]
    .recv_timeout::<Simulator>(self.timeout).await {
    payload.push(car.clone());
    exec.advance(car.time).await;
    car.ctrl.set(true);
    if arrived_cars.len() == self.capacity {
      self.departure(&mut payload).await;
    }
  } else {
     self.departure(&mut payload).await;
  }
}
```

Listing 17: Hauptschleife der Fähre. In dem Vektor `payload` werden die auf der Fähre wartenden Autos gespeichert. `car.time` bezeichnet die Zeit die ein Auto zur Auf- und Abfahrt von der Fähre benötigt. `self.departure()` führt die Überfahrt der Fähre zum anderen Hafen und das Abfahren der Autos von der Fähre ab. Es wird gerufen, wenn die Fähre voll ist (also `capacity` Autos im Vektor `arrived_cars` gespeichert sind), oder wenn `recv_timeout()` mit einem Timeout zurück kam.

SLX-Programm

In der SLX-Version des Fähren-Beispiels wird eine passive Klasse für den Hafen benötigt, da SLX keine Kanäle bietet. Diese Häfen bieten die Möglichkeit, Autos in eine FIFO-sortierte Liste einzufügen und herauszunehmen. Somit können die Häfen wie eine Art Kanal benutzt werden. In Listing 18 wird die Hauptschleife des Fähren-Beispiels in SLX erklärt.

Ergebnisse

In Abbildung 10 ist eine grafische Repräsentation der Ergebnisse des Benchmarks zu sehen und in Tabelle 3 eine tabellarische Repräsentation der Ergebnisse. Wie aus den Ergebnissen ablesbar ist, ist die Rust-Implementation mit 99% Wahrscheinlichkeit etwa um den Faktor $6,3$ langsamer als die SLX-Implementation. Außerdem ist der Unterschied zwischen Rust und SLX in diesem Beispiel bei großen Eingabewerten fast doppelt so groß wie im vorigen Kontextwechsel-Beispiel. Dies könnte daran liegen, dass die Umsetzung des Wartens der Fähre auf die Ankunft eines Autos oder das Erreichen eines Timeouts in SLX effizienter implementiert sind als in *mk-simulation*. In *mk-simulation* muss dafür ein neuer Prozess initialisiert, was eventuell langsamer ist als das äquivalente `wait until` Statement in SLX.

```
forever {
  while (load < capacity) {
    cutoff = time + timeout;
    wait until harbors[h]->peer.size > 0 || time >= cutoff;

    if (harbors[h]->peer.size == 0) {
      break;
    }
    car = harbors[h]->take();
     advance car->drive_time;
     car->control_car = TRUE;
     load += 1;
     payload[load] = car;
   }
   //Abfahrt der Fähre
   ...
}
```

Listing 18: Hauptschleife der Fähre. Solange die Fähre noch nicht voll ist, wird über das `wait until` Statement gewartet, bis entweder ein neues Auto ankommt oder der Timeout abgelaufen ist. Falls kein Auto da ist ist der Timeout abgelaufen, ansonsten fährt das angekommene Auto in `car->drive_time` Zeit auf die Fähre. Das Verhalten ist analog zum `recv_timeout()` aus dem Rust-Programm.

Eingabe	Rust (Median)	SLX (Median)
1000	[469,64 , 470,00 , 470,36] us	[74,167 , 74,540 , 74,947] us
2000	[917,99 , 919,29 , 920,74] us	[147,89 , 148,12 , 148,57] us
4000	[1,8336 , 1,8364 , 1,8391] ms	[294,75 , 295,50 , 296,36] us
8000	[3,6454 , 3,6485 , 3,6541] ms	[587,07 , 588,43 , 589,44] us
16000	[7,3274 , 7,3380 , 7,3486] ms	[1,1701 , 1,1725 , 1,1742] ms
32000	[14,658 , 14,775 , 15,163] ms	[2,3369 , 2,3409 , 2,3452] ms
64000	[29,221 , 29,262 , 29,346] ms	[4,6637 , 4,6695 , 4,6854] ms
128000	[58,284 , 58,412 , 58,633] ms	[9,3213 , 9,3344 , 9,3489] ms
256000	[116,10 , 116,57 , 116,77] ms	[18,618 , 18,653 , 18,696] ms
512000	[231,38 , 232,00 , 233,00] ms	[37,219 , 37,278 , 37,355] ms

Tabelle 3: Ergebnisse des Fähren-Benchmarks inklusive des 99% Konfidenzintervalls für jeden Versuch.

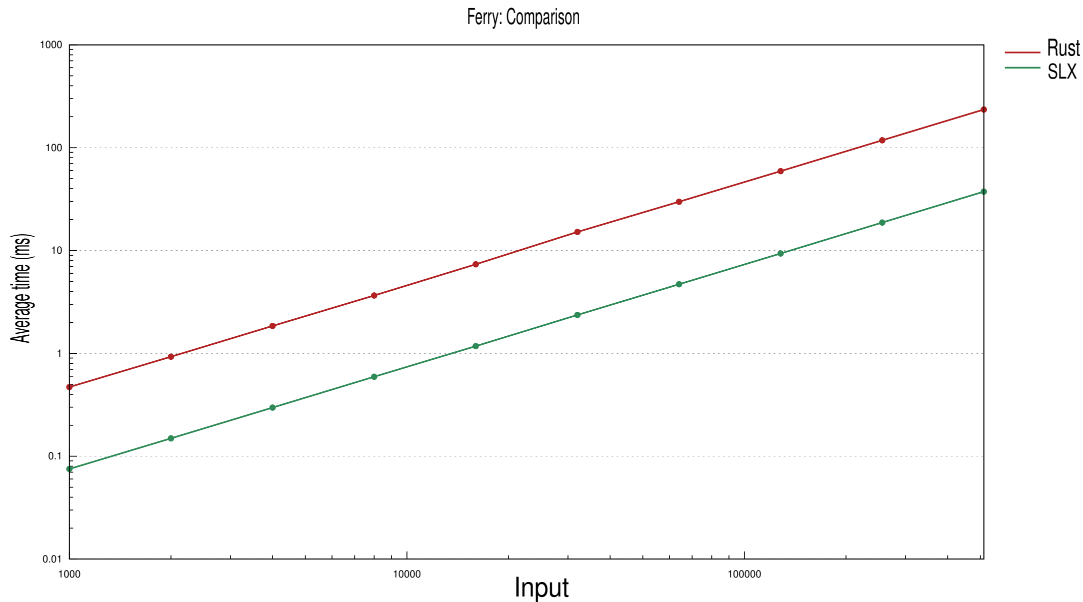


Abbildung 10: Grafische Repräsentation der Ergebnisse des Fähren-Benchmarks.

5 Zusammenfassung und Ausblick

In dieser Arbeit konnte gezeigt werden, dass es möglich ist, einen Simulatorkern in Rust zu implementieren.
Bei den Benchmarks konnte beobachtet werden, dass *mk-simulation* langsamer ist als SLX. Für eine statistisch relevante Aussage über den Geschwindigkeitsunterschied kann wegen der geringen Anzahl an Beispielen und der großen Schwankung der Ergebnisse keine Aussage getroffen werden. So ist *mk-simulation* in den drei durchgeführten Beispielen zweimal etwa um den Faktor 3 und einmal um den Faktor 6 langsamer als SLX. Allerdings können in Rust einfach mehrere voneinander unabhängige Simulationsläufe parallel ausgeführt werden, was die Performanz erhöht [7]. In SLX ist das nicht einfach möglich, da SLX nur Programme mit einem Thread unterstützt. Weiterhin sind in Rust Simulationen mit unterschiedlichen Datentypen für Zeit und Priorität möglich, in SLX müssen hierfür immer Fließkommazahlen verwendet werden.

In zukünftigen Arbeiten könnten folgende Bereiche weiter untersuchen:

- *mk-simulation* ist keine vollwertige Alternative zu SLX. So ist beispielsweise Intra-Prozess-Parallelität in SLX möglich, in *mk-simulation* jedoch nicht. In zukünftigen Arbeiten könnte die Funktionalität von *mk-simulation* so erweitert werden, dass es sich danach um eine vollwertige Alternative zu SLX handelt.
- Es könnten weitere Benchmarks durchgeführt werden, um eine sicherere Aussage über den Laufzeitunterschied tätigen zu können.
- Es könnte untersucht werden, warum *mk-simulation* beim Fähren-Beispiel langsamer im Vergleich zu SLX ist als in den beiden anderen Beispielen und die Laufzeit von *mk-simulation* könnte optimiert werden.

Literatur

[1] J. Banks u. a. *Discrete-Event System Simulation.* Hrsg. von Holly Stark. Pearson, 2010.

[2] Andreas Blunk und Joachim Fischer. “A Highly Efficient Simulation Core in C++”. In: *Proceedings of the Symposium on Theory of Modeling & Simulation - DEVS Integrative.* DEVS ’14. Tampa, Florida: Society for Computer Simulation International, 2014.

[3] *crossbeam.rs.* Zugriff 31.05.2021. URL: `https://github.com/crossbeam-rs/crossbeam`.

[4] Brook Heisler. *Criterion.rs.* Zugriff: 31.05.2021. URL: `https://github.com/bheisler/criterion.rs`.

[5] J.O. Henriksen. “SLX: the X is for extensibility [simulation software]”. In: *2000 Winter Simulation Conference Proceedings (Cat. No.00CH37165).* Bd. 1. 2000, S. 183–190. DOI: `10.1109/WSC.2000.899715`.

[6] Greg Lomow und Dirk Baezner. “A Tutorial Introduction to Object-Oriented Simulation and Sim++”. In: *Proceedings of the 23rd Conference on Winter Simulation.* WSC ’91. Phoenix, Arizona, USA: IEEE Computer Society, 1991, S. 157–163. ISBN: 0780301811.

[7] Dorian Weber, Paula Wiesner und Joachim Fischer. “A Closer Look at Process-Based Simulation with Stackless Coroutines”. In: *Information and Software Technology (eingereicht, noch nicht veröffentlicht)* (2021).

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und noch nicht für andere Prüfungen eingereicht habe. Sämtliche Quellen einschließlich Internetquellen, die unverändert oder abgewandelt wiedergegeben werden, insbesondere Quellen für Texte, Grafiken, Tabellen und Bilder, sind als solche kenntlich gemacht. Mir ist bekannt, dass bei Verstößen gegen diese Grundsätze ein Verfahren wegen Täuschungsversuchs bzw. Täuschung eingeleitet wird.

Karlsruhe, den 7. November 2021 ....