NTB
Interstaatliche Hochschule
für Technik Buchs
FHO Fachhochschule Ostschweiz

Neuer Sequencer für EEROS

Vertiefungsarbeit 1 2016/1017

von

Marcel Gehrig

Advisor:	Dr. Urs Graf
Abgabedatum:	8. Februar 2017

Kurzfassung

In dieser Arbeit wird ein neuer Sequencer für EEROS vorgestellt.

EEROS ist ein echtzeitfähiges Roboter-Framework, welches an der Interstaatlichen Hochschule für Technik Buchs (NTB) entwickelt wurde, und immer noch weiter entwickelt wird. Das Framework ist Open Source und frei erhältlich.

EEROS besteht aus den drei Hauptteilen Sequencer, SafetySystem und ControlSystem. Der Sequencer ist dabei der Teil der Software, der den Ablauf des Roboters steuert.

In dieser Arbeit wurden zuerst der bestehende Sequencer analysiert, um die Schwächen und Stärken zu ermitteln. Im Anschluss wurden die Anforderungen für den neuen Sequencer definiert.

Es wurde ein Set von Testfällen ausgearbeitet, welche alle Anforderungen abdecken. Die Testfälle wurden genutzt um ein Sequencer zu entwickeln, der alle Testfälle abdeckt.

Bei der Entwicklung des Sequencer wurde auf zwei Hauptfokuspunkte geachtet. Zum einen sollte er möglichst flexibel sein, damit EEROS auch weiterhin bei einer Vielzahl von sehr unterschiedlichen Robotern eingesetzt werden kann. Der zweite Fokuspunkt war er möglichst einfache Aufbaue einer Sequenz. Eine fertige Sequenz sollte so einfach sein, dass auch ein Nicht-Fachmann den Ablauf versteht und geringfügige Änderungen machen kann.

Inhaltsverzeichnis

1 Einleitung

1.1 Git-Repositories

Dieses Dokument enthält die gesamte Dokumentation. Zusätzlich wurden auf drei verschiedenen Git-Repositories entwickelt.

Pseudo Sequencer
Der *Pseudo Sequencer* wurde am Anfang der Entwicklung geschrieben, um Zusammenhänge zu verstehen und das ganze Konzept logisch zu verstehen. Er erhält Code, der C++ ähnelt, ist aber nicht lauffähig.

- `https://github.com/MarcelGehrig2/pseudoSequencer`

TestApp Dieses Repository wurde synchron mit der EEROS-Repository mitentwickelt. Wenn beim Sequenzer im EEROS-Repository eine neue Funktion eingebaut wurde, wurde sie vor dem Commit mit dieser Testapplikation getestet.

- `https://github.com/MarcelGehrig2/testAppVt1`

EEROS-Repository Das EEROS-Repository ist ein Fork vom originalen EEROS-Repository (Hash: f4d8c8e32c138fdca5d1eb47d6e821c374c2cf07). In diesem Repository wurde der Sequenzer entwickelt.

- `https://github.com/MarcelGehrig2/eeros-vt1`

1.2 EEROS

EEROS (Easy, Elegant, Reliable, Open and Safe) ist ein echtzeitfähiges, Open Source Software Framework, welches an der NTB entwickelt wurde und auch immer noch weiter entwickelt wird. Das Ziel von EEROS ist, möglichst zuverlässig und einfach in der Bedienung zu sein. Da das Framework besonders auch in industriellen Robotern zum Einsatz kommen soll, ist besonders auch die Zuverlässigkeit der Software ein wichtiger Punkt. Für die Software wird die objektorientierte Programmiersprache C++ verwendet.

EEROS kann in vier Hauptbereiche unterteilt werden.

1. Die HAL (Hardware Abstraction Layer), welche als Schnittstelle zur Hardware dient.
2. Das CS (*ControlSystem*). Im CS wird die Regelung des Roboters aufgebaut. In diesem System wird aber nicht nur die Regelung gerechnet, sondern auch Aufgaben wie die Berechnung der Vorwärts- und Inverskinematik werden hier erledigt.
3. Der Sequencer steuert den Ablauf des Roboters. Hier werden sowohl Wegpunkte aufgelistet, wie auch das allgemeine Verhalten definiert.
4. Im SS (*SafetySystem*) werden sicherheitsrelevante Parameter überwacht. Das SS arbeitet unabhängig vom CS und vom Sequencer. Es löst ein Not-Aus aus, wenn der Roboter ausserhalb der zulässigen Parameter operiert. Ein möglicher Grund für ein Not-Aus wäre zum Beispiel, wenn sich der Roboterarm in einem Sicherheitsbereich zu schnell bewegt.

1.3 Klarstellung der Benennungen

Die meisten Programmiersprachen werden in Englisch codiert. Auch die offizielle Onlinedokumentation [EER-17] von EEROS und die Benennung von Komponenten und Konzepten ist in Englisch. In diesem Dokument wird an vielen Stellen bewusst darauf verzichtet, englische Bezeichnungen auf Deutsch zu übersetzen. Dies kann zu Deutsch - Englischen Mischwörter führen. Solche Mischwörter sind zwar nicht elegant, können aber besser für die Verständlichkeit sein und werden deshalb mit Absicht verwendet. Auch einige Eigennamen, wie z.B. *Sequencer* anstelle von *Sequenzer*, werden in diesem Dokument nicht immer auf Deutsch übersetzt.

In dieser Arbeit wird oft von drei verschiedenen Kategorien von Entwicklern gesprochen. Es wird zwischen EEROS-, Steuerungs- und Applikationsentwickler unterschieden.

Der **EEROS-Entwickler** hat vertiefte Kenntnisse der Programmiersprache C++ und vom EEROS Framework. Seine Hauptaufgabe ist die Weiterentwicklung des Frameworks, welches vom Steuerungsentwickler verwendet wird.

Der **Steuerungsentwickler** hat ebenfalls gute C++ Kompetenzen und nutzt das Framework um eine Steuerung für einen Roboter zu entwickeln. Dafür muss er seine Software speziell an den Roboter anpassen. Er bereitet auch erste Sequenzen für den Applikationsentwickler vor. Oft wird die Entwicklung der Steuerung und der Applikation von derselben Person übernommen.

Um den Ablauf des Roboters anzupassen kann der **Applikationsentwickler** bestehende Sequenzen einfach abändern. Dazu werden nur grundlegende Programmierfähigkeiten benötigt. Mit etwas erweiterten Kenntnissen kann er auch neue Sequenzen erstellen.

1.4 Aufgabenstellung

In der aktuellen Version von EEROS existiert bereits eine erste Version von einem Sequencer. Dieser ist in seiner Funktionalität und Übersichtlichkeit aber stark eingeschränkt. Oft musste auf Tricks zurückgegriffen werden, damit bestimmte Aufgaben mit dem bestehenden Sequencer gelöst werden konnten. Um eine bestehende Sequenz anpassen zu können, auch wenn der Ablauf nur geringfügig geändert werden soll, ist schon viel Fachwissen notwendig.

In dieser Vertiefungsarbeit sollen diese beiden Probleme gelöst werden. Es soll ein neuer Sequencer entwickelt werden, der flexibel für verschiedenste Arten von Robotern eingesetzt werden kann. Die Sequenzen, welche den Ablauf des Roboters beschreiben, sollen dabei möglichst einfach und übersichtlich aufgebaut sein. Dank des einfachen Aufbaus soll es auch für einen Applikationsentwickler möglich sein Sequenzen anzupassen und zu erstellen, auch wenn er keine vertieften Kenntnisse von C++ besitzt.

2 EEROS aktueller Stand

2.1 Aktuelle Implementierung des Sequencers

Für EEROS besteht bereist ein rudimentärer *Sequencer*. Im folgenden Kapitel wird die bestehende Implementierung kurz erklärt. In der Onlinedokumentation [EER-17] wird noch vertiefter in die Details des bestehenden *Sequencers* eingegangen, als in dieser Arbeit.

2.1.1 Sequencer

Die Grundlage bildet ein Sequencer-Objekt, das in einem Nicht-Realtime-Thread läuft. In solch einem *Sequencer* können eine Serie von blockierenden *Sequences* aufgerufen werden. Wenn mehrere parallele *Sequences* aufgerufen werden sollen, muss für jede *Sequence* ein eigener *Sequencer* erstellt werden.

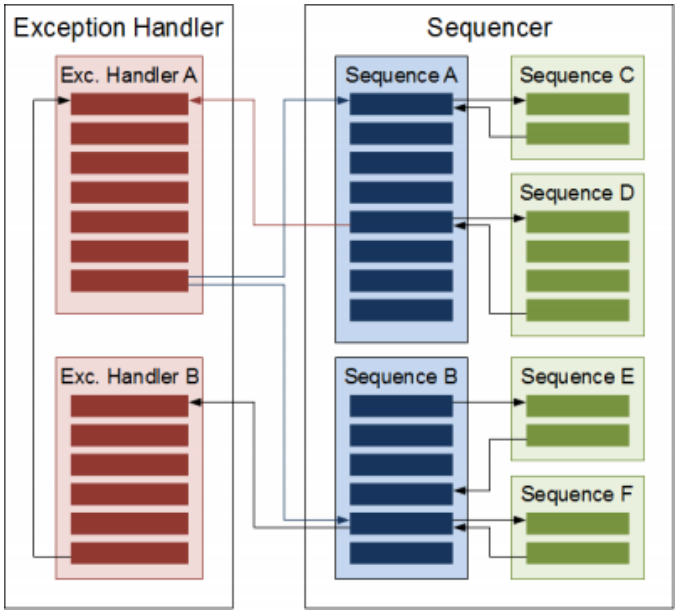


Abbildung 2.1: Schematische Darstellung des bestehenden Sequencers [EEW-17]

2.1.2 Sequence

Eine *Sequence* führt als erstes eine benutzerdefinierte Initialisierungsfunktion aus. Als nächstes wird eine *preCondition* überprüft. Schlägt die Überprüfung fehl, wird die *Sequence* abgebrochen. Bei einem positiven Ergebnis wird der Hauptteil, eine Abfolge von definierten *Steps*, ausgeführt. Dem letzten *Step* folgt noch eine Überprüfung einer *postCondition*. Läuft der *Sequencer* im *stepping-mode* wird die *Sequence* bei jedem *yield()* pausiert und erst fortgeführt, wenn der Befehl dazu gegeben wird. [EEW-17]

```
1  init();
2  yield();
3
4  if(!checkPreCondition())
5    return SequenceResult<void>(result::preConditionFailure);
6  yield();
7
8  run();
9  yield();
10
11 if(!checkPostCondition())
12   return SequenceResult<void>(result::postConditionFailure);
13 yield();
```

```
14
15 exit();
16 return SequenceResult<void>(result::success);
```

2.1.3 Step

Ein *Step* ist eine vom *Steuerungsentwickler* festgelegte Einheit, die von einem *yield()* Befehl unterteilt wird. *Steps* können im *stepping-mode* einzeln abgearbeitet werden.

2.1.4 Sub-Sequence

Subsequences sind sehr ähnlich wie normale *Sequenzen*. Sie können verwendet werden, wenn ein ähnlicher Ablauf mehrmals wiederholt werden soll. Durch eine Übergabe von Parametern an eine solche *Subsequence* kann sie sehr flexibel gestaltet werden. Wenn die *Subsequence* parallel, also nicht-blockierend, aufgerufen werden soll, muss sie einem neuen Sequencer übergeben werden.

```
1  class SequenceB : public Sequence<> {
2  public:
3    SequenceB(std::string name, Sequencer* seq, Robot& r) : Sequence<void,double>(name↩
         , seq), robot(r){ }
4
5    void run() {
6      robot.moveZ(5);
7      sleep(3);
8      yield();
9      robot.moveZ(0);
10   }
11
12 private:
13   Robot& robot;
14 };
```

2.1.5 Error-Handler

Der *Error-Handler* wird in der Onlinedokumentation zwar erwähnt, im Sourcecode von EEROS sind aber keine Spuren von der Implementierung zu finden. Der Dokumentation zufolge soll er *Exceptions* behandeln. Je nach *Exception* soll er flexibel reagieren um Probleme zu beheben.

2.2 Fallbeispiel 'EEDURO Delta Roboter'

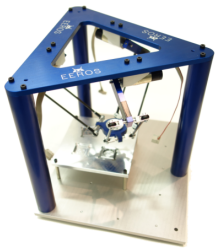


Abbildung 2.2: Der EEDURO Delta Roboter

Der *EEDURO Delta Roboter* ist ein Roboter, dessen Hard- und auch Software an der NTB entwickelt wurde. Die Steuerung ist mit EEROS und dem bestehenden *Sequencer* aufgebaut. Im folgenden Kapitel wird der Quellcode der Steuerung analyisiert um herauszufinden, welche Aspekte des bestehenden *Sequencers* brauchbar sind, und wo er noch Lücken aufweist.

Der Quellcode der Steuerungssoftware ist auf folgendem Git-Repository zu finden:
https://github.com/ClaudiaVisentin/eeduro-platform

Die Software wurde vom Stand 10.10.2016 mit dem Hash *a25bcfa752516723f067f5a5166e8f09e60fc6e8* verwendet.

2.2.1 Betrachtung des bestehenden Sequencers vom Delta Roboter

Öffnet man die Hauptsequenz des Sequenzers wird schnell klar, dass die Sequenz alles andere als übersichtlich ist. Der Zugriff auf das *ControlSystem* und der Zugriff auf des *SafetySystem* sind verwoben mit dem eigentlichen Ablauf des Roboters.

Auch wenn man Erfahrungen hat zum C++ Source Code zu lesen, ist es nicht einfach, den Ablauf zu verstehen. Zusätzlich ist es an manchen Stellen notwendig, dass man weiss, wie das *ControlSystem* aufgebaut ist.

Oft wird auch das *SafetySystem* verwendet um gewisse Abläufe zu programmieren. Die Zeile *safetySys->triggerEvent(doParking);* löst zum Beispiel im *SafetySystem* einen Event aus welcher, über den Umweg von zwei Safety-Level und noch einem zusätzlichen Event, folgenden Code ausführt:

```
1  static unsigned int count = 0;
2
3  if(count == 1) {
4     controlSys->setVoltageForInitializing({q012InitVoltage, q012InitVoltage, ↩
          q012InitVoltage, q3InitVoltage});
5  }
6  else if(count > static_cast<unsigned int>(2.0 / dt)) {
7     if(controlSys->switchToPosControl()) {
8        privateContext->triggerEvent(homeingDone);
9     }
10    count = 0;
11 }
12 count++;
```

Dieser Code-Teil übernimmt das *Homeing* des Roboters. Von der Logik her, wäre er besser im Sequencer untergebracht. Das *SafetySystem* sollte möglichst einfach aufgebaut sein, damit es weniger Angriffspunkte für Fehler hat und zuverlässiger läuft.

3 Anforderungen an den Sequencer

3.1 Formulierung der Anforderungen

Im folgenden Kapitel werden die Anforderungen an den *Sequencer* beschrieben. Die Kapitel *Ziele*, *Nicht Ziele* und *Test Cases* beschreiben die Anforderungen auf verschiedene Arten. In *Ziele* und *Nicht Ziele* wird abstrakt beschrieben, welche Funktionen der neue *Sequencer* beinhalten, beziehungsweise nicht beinhalten muss. Im Kapitel *Test Cases* werden verschiedene Fälle beschrieben die mit dem neuen *Sequencer* möglichst elegant gelöst werden sollen.

3.2 Ziele

3.2.1 Einfaches Interface für den Applikationsentwickler

Mit dem bestehenden *Sequencer* sind vertiefte Programmierkenntnisse notwendig, um eine Sequenz zu erstellen, oder abzuändern. Sequenzen sind nicht intuitiv verständlich. Es werden Kenntnisse vom *Control System* und dem *Safety System* benötigt, um einen neuen Ablauf für den Roboter schreiben zu können. Zurzeit werden Sequenzen von den Steuerungsentwickler geschrieben.

Wenn der Roboter fertig entwickelt worden ist, soll er an einen Kunden übergeben werden können. Der Kunde, oder der Betreiber des Roboters, soll dann Änderungen im Ablauf des *Sequencers* vornehmen können, ohne dass er vertiefte Kenntnisse von der Programmiersprache C++ oder von der inneren Funktionsweise des Roboters haben muss. Dafür ist es notwendig, dass der Steuerungsentwickler den Roboter so abstrahiert, dass die Sequenzen aus logischen und verständlichen Schritten bestehen.

3.2.2 Flexibel einsetzbar für verschiedenste Arten von Roboter

Auch wenn die Sequenzen möglichst einfach aufgebaut werden sollen, muss der *Steuerungsentwickler* für alle möglichen Kategorien von Robotern Sequenzen bauen können. Verschiedene Arten von Roboter haben verschiedene Anforderungen. Das *Control System* von einem Roboterarm mit sechs Freiheitsgraden unterscheidet sich stark von einer Fertigungsstrassen mit mehreren Förderbändern. Trotz diesen Unterschieden soll es möglich sein, für beide Kategorien von Robotern sinnvolle Sequenzen zu erstellen. Das bedeutet, dass der *Steuerungsentwickler* möglichst viele Freiheiten beibehält, ohne dass er durch das *Framework* unnötig begrenzt wird.

3.2.3 Parallele und blockierende Sequenzen

Wie bereits im bestehenden Sequenzer verwirklicht wurde, sollen Sequenzen blockierend und nichtblockierend aufgerufen werden können. Diese Funktion von blockierenden und parallel ausgeführten Sequenzen soll auch im neuen *Sequencer* beibehalten werden.

3.2.4 Exception Handling

Eine *Exception* ist ein Ereignis, dass nicht immer auftritt, aber auftreten kann. Zu solchen *Exception*, oder Ausnahmen, gehören zum Beispiel:

1. Ein blockiertes Förderband
2. Ein Timeout
3. Der Roboter soll ein Paket abholen, dass nicht vorhanden ist

Solche Ausnahmen sollen im *Sequencer* erkennt werden können und flexibel darauf reagiert werden können. Eine solche Reaktion könnte eine spezielle Sequenz sein, die versucht, eine solche Ausnahme zu behandeln. Alternativ soll aber auch die aktuelle Sequenz abgebrochen, oder neu gestartet werden können.

3.2.5 Zugriff auf Control System

Der aktuelle *Sequencer* nutzt ein Pointer auf das *Control System* um Blöcke direkt auslesen und schreiben zu können. Wenn das *Control System* betrachtet wird, kann nicht festgestellt werden, welche Blöcke vom *Sequencer* ausgelesen oder geschrieben werden. Ein klares Interface zum *Sequencer* wäre wünschenswert, um das *Control System* übersichtlicher zu machen.

3.2.6 Safety System entlasten

Eine Analyse von bestehenden Implementationen des alten *Sequencers* hat gezeigt, dass das *Safety System* viele Aufgaben übernimmt, welche besser vom *Sequencer* übernommen werden sollten. Das *Safety System* sollte möglichst nur eingesetzt werden, um das System zu überwachen. Alle anderen Aufgaben sollen vom *Sequencer* oder vom *Control System* übernommen werden.

3.3 Nicht Ziele

3.3.1 Echtzeit

Das *Control System* und das *Safety System* laufen beide in einem Echtzeit-Task. Der *Sequencer* soll aber bewusst nur mit normaler Priorität laufen und besitzt keine Echtzeit Fähigkeit.

Da der *Sequencer* keine Regelung berechnet, benötigt er keine Echtzeit Fähigkeit. Eine niedrigere Priorität als das *Control System* und das *Safety System* ist notwendig, dass die beiden Systeme nicht vom *Sequencer* beeinträchtigt werden.

3.3.2 Pfadplanung

Die Pfadplanung ist nicht Teil des *Sequencers*, da sie den Rahmen dieser Arbeit sprengen würde.

3.4 Test Cases

3.4.1 Einleitung

Die Testfälle sind so aufgebaut, dass sie möglichst einfach und elementar sind. Jeder *Test Case* beschreibt eine andere Anforderung oder Spezialfall an den *Sequencer*. Alle in der Realität vorkommenden Situation sollte durch einen, oder einer Kombination von mehreren, *Test Cases* beschrieben werden können.

Eine Ausnahme dazu bildet *Test Case 8*. In diesem Testfall wurden möglichst viele verschiedene Situationen vereint.

Mit dem neuen *Sequencer* sollen alle Testfälle sauber und elegant umgesetzt werden werden können.

3.4.2 Test Case 1: Achse einfach

System

- Eine Achse die sich nach links und rechts bewegen kann
- An beiden Enden befindet sich ein Endschalter
- Ein Taster *Taster links*, der die Achse nach links laufen lässt
- Ein Taster *Taster rechts*, der die Achse nach rechts laufen lässt

Aufgabe

- Solange *Taster links* gedrückt bleibt, fährt die Achse nach links
- Solange *Taster rechts* gedrückt bleibt, fährt die Achse nach rechts
- Die Achse hält an, wenn einer der beiden Endschalter erreicht wird

Herausforderungen

- Der *Sequencer* muss die Eingänge *Taster links*, *Taster rechts* und die beiden Endschalter permanent überwachen und auf eine Änderung reagieren.

3.4.3 Test Case 2: EEDURO Delta Roboter Maus

System

- EEDURO Delta Roboter mit Maus

Aufgabe

- Während dem Idle Zustand wird auf einen Input von der Maus gewartet
- Wenn sich die Maus für fünf Sekunden nicht bewegt, wird eine blockierende *Autor-Sort Sequenz* gestartet
- Bewegt sich die Maus innerhalb von fünf Sekunden, dann bewegen sich die Achsen entsprechend der Mausbewegung und der 5-Sekunden-Timer wird neu gestartet

Herausforderungen

- Timeout
- Blockierende Sequenz

3.4.4 Test Case 3: Rendezvous

System

- Greifer Zubringer
- Greifer Abholer
- Paket: Gegenstand, der übergeben wird
- Förderband Zubringer: Hält ständig ein neues Paket bereit für *Greifer Zubringer*
- Förderband Abholer: Transportiert ständig alle Pakete weg, welche vom *Greifer Abholer* abgelegt werden

Ablauf (Siehe Abbildung 3.1)

1. Der *Greifer Zubringer* holt ein neues Paket vom *Förderband Zubringer*
2. Der *Greifer Zubringer* bringt das Paket in Rendezvous-Position
3. Der *Greifer Abholer* übernimmt das Paket vom *Greifer Zubringer*
4. Der *Greifer Abholer* legt das Paket auf das *Förderband Abholer*

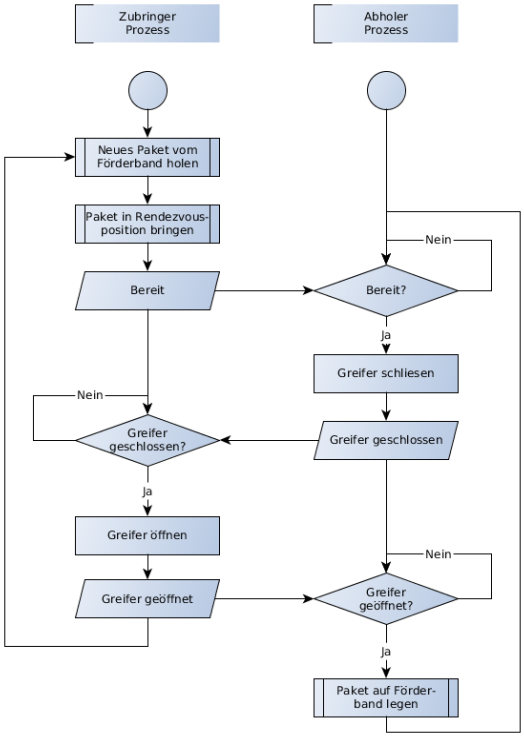


Abbildung 3.1: *Test Case 3*: Ablaufdiagramm vom

5. Die Sequenz beginnt wieder von Anfang an

Herausforderungen

- Synchronisation von zwei parallellaufenden Sequenzen
- Kommunikation zwischen zwei Sequenzen

3.4.5 Test Case 4: Sequenz pausieren

System

- Roboterarm
- Ein Taster *Taster Pause*, der die Sequenz pausiert

Ablauf

- Der Roboterarm führt eine sich wiederholende Sequenz endlos aus
- Wird *Taster Pause* gedrückt, pausiert die Sequenz
- Wird *Taster Pause* erneut gedrückt, wird die Sequenz fortgeführt

Herausforderungen

- Jede Sequenz muss jederzeit pausiert werden können
- Der Taster gibt nur einen Impuls, nicht eine bleibende Pegeländerung an einem Eingang
- Timeouts müssen pausiert werden

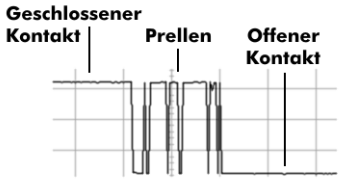


Abbildung 3.2: *Test Case 6*: Signalverlauf bei einem prellenden Schalter

3.4.6 Test Case 5: Zwei Roboterarme

System

- Roboterarm A
- Roboterarm B
- Ein Taster *Taster Pause*, der beide Sequenzen pausiert

Ablauf

- Beide Roboterarme führen sich wiederholende Sequenz endlos aus
- Wird *Taster Pause* gedrückt, pausieren beide Sequenzen

Herausforderungen

- Zwei parallellaufende Sequenzen müssen mit einem Taster pausiert werden können

3.4.7 Test Case 6: Prellender Taster

System

- Ein prellender Taster *Taster A*

Aufgabe

- Ein prellender Taster soll sauber eingelesen werden

Herausforderungen

- Der Taster muss entprellt werden
- Mehrmaliges drücken soll sauber registriert werden, auch wenn der Taster in schneller Abfolge gedrückt wird

3.4.8 Test Case 7: Menü

System

- Ein beliebige Anzahl Taster *Taster A*, *Taster B*, *Taster C*, ...

Aufgabe

- Wenn sich das System in einem *Idle* Zustand befindet, soll jeder Taster eine andere Sequenz starten

Herausforderungen

- Dieses *Menü* lässt sich nicht im klassischen Sinne als eine Sequenz, also eine Abfolge von Schritten, beschrieben werden. Trotzdem soll eine einfache Implementierung im *Sequencer* möglich sein

3.4.9 Test Case 8: Detailliertes Rendezvous

Dieser *Test Case* basiert auf auf dem *Test Case 3*, ist aber detaillierter ausgearbeitet. Im Anhang A ist das Ablaufdiagramm der Hauptsequenz *Sequenz Rendezvous* und der blockierenden Subsequenz *Sequenz PickUp* angehängt. Das Ziel von diesem *Test Case* war es nicht einen realistischen Fall abzubilden. Viel mehr dient er dazu, möglichst viele Fälle und Situationen abzubilden, die in einer Sequenz vorkommen können.

Das Ablaufdiagramm besteht aus folgenden Blöcken:

- Rechtecke: einzelne Steps
- Rechtecke mit doppelten, vertikalen Linien: Blockierende Subsequenzen
- Blaue Rauten: Entscheidungen
- Grüne Rauten: Blockierungen, bis Entscheidung wahr wird
- Parallelogramm: Statusvariable wird geändert
- Lange senkrechte Rechtecke: Bedingung, welche über längere Zeit überwacht wird

4 Dokumentation für Steuerungsentwickler

Dieses Kapitel ermöglicht einen leichten Einstieg in den neuen Sequencer. Es deckt aber nicht alle Details von allen Funktionen des *Sequenzers* vollständig ab. Eine detailliertere Beschreibung des *Sequenzers* und von dessen Funktionen findet sich im Kapitel 5.

4.1 Den Sequencer erstellen

Im Hauptprogramm der Applikation muss der Sequenzer erst erstellt werden. Die Hauptsequenz, in diesem Beispiel *mainSequence* genannt, wird in einem Thread gestartet, sobald sie dem Sequencer hinzugefügt wird. Alle anderen Sequenzen werden von der Hauptsequenz aus gestartet.

main.cpp:

```
#include "sequences/MainSequence.hpp"

...

eeros::sequencer::Sequencer S;
MainSequence mainSequence(S, &controlSystem, "mainSequence");
S.addMainSequence(&mainSequence);

executor.run();
mainSequence.join(); //The application only stops, when the mainSequence is finished

...
```

4.2 Benutzerdefinierte Klassen

Den Kern des neuen *Sequencers* bilden benutzerdefinierte Klassen, welche von vordefinierten EEROS-Klassen abgeleitet werden. Dieses System erlaubt dem *Steuerungsentwickler* die spezifische Implementierung von Funktionen hinter Klassen mit sinnvollen Namen zu verstecken.

Weil die Implementierung in den benutzerdefinierten Klassen versteckt wird, können einfach verständliche Sequenzen aufgebaut werden.

Die wichtigsten EEROS-Klassen für diesen Zweck sind:

- Sequence
- Step
- Condition

In den folgenden Kapiteln wird mit Hilfe von einem Pseudo-System demonstriert, wie diese Klassen verwendet werden können.

4.3 Beispiel für eine Hauptsequenz

MainSequence.hpp:

```
1  #include "SequenceA.hpp"
2  #include "SequenceB.hpp"
3  #include "SequenceExceptionA.hpp"
4
5  namespace testappsequencer {
6
7     class TestAppCS;
8
9     class MainSequence : public eeros::sequencer::Sequence {
10    public:
11       MainSequence(Sequencer& S, TestAppCS* CS, std::__cxx11::string name);
12
13       int action();
14
15       SequenceA seqA1;
16       SequenceB seqB1;
17       SequenceB seqB2;
18       SequenceB seqB3;
19
20       SequenceExceptionA seqEA1;
21
22       TestAppCS* CS;
23    };
24 };
```

Jede Sequenz muss von der Klasse *eeros::sequencer::Sequence* abgeleitet werden.

Zeile 7 und **Zeile 22** sind notwendig, um einen Pointer auf das *ControlSystem* zu erhalten. Dafür muss zusätzlich der *Constructor* angepasst werden.

int action(); ist die wichtigste Methode, denn sie beinhaltet den Ablauf der Sequenz.

Zeile 15 bis **Zeile 20** beschreiben benutzerdefinierte Sequenzen, die von der Hauptsequenz aus gestartet werden.

MainSequence.cpp:

```
1  #include "MainSequence.hpp"
2
3  using namespace testappsequencer;
4  using namespace eeros;
5  using namespace eeros::sequencer;
6
7
8  MainSequence::MainSequence(Sequencer& S, TestAppCS* CS, std::__cxx11::string name) :
9  Sequence(S, name), CS(CS),
10
11 seqA1(S, CS, this, "seqA1"),
12 seqB1(S, CS, this, "seqB1"),
13 seqB2(S, CS, this, "seqB2"),
14 seqB3(S, CS, this, "seqB3"),
15 seqEA1(S, CS, this, "seqEA1")    //Exception Sequence
16 {
17    setIsNonBlocking();
18
19    seqA1.setTimeoutTime(5.1);
20    seqA1.setTimeoutExceptionSequence(&seqEA1);
21    seqA1.setTimeoutBehavior(restartOwner);
22
23    seqA1.setIsBlocking();
24 }
25
26 int MainSequence::action()
27 {
28    seqB1();
29    seqB2();
30    seqA1(10, 3);
31    seqB3();
32
33    seqB1.join();
34    seqB2.join();
35    seqB3.join();
```

```
36
37     log.info() << "MainSequence ended";
38 }
```

In **Zeile 9** wird die Basis-Sequenz und der Pointer für das *ControlSystem* initialisiert. Zusätzlich müssen auch alle Sequenzen, die in dieser Sequenz genutzt werden, initialisiert werden. Die ersten drei Parameter sind dabei immer gleich. Der vierte Parameter ist der Name der Sequenz.

Zeile 17 definiert diese Sequenz als nicht-blockierend. Die Hauptsequenz von einer Applikation muss immer nicht-blockierend sein. Andere Sequenzen können blockierend sein.

Die **Zeile 19** aktiviert für die Sequenz *seqA1* den Timeout und setzt die Zeit auf 5.1 Sekunden. Wenn *seqA1* nach 5.1 Sekunden Laufzeit noch nicht fertig abgearbeitet wurde, wird *seqEA1* ausgeführt. Nachdem die Exception-Sequenz *seqEA1* beendet wurde, wird *seqA1* neu gestartet, da dieses Verhalten in **Zeile 21** definiert wurde. Mehr Informationen zum Verhalten finden sich im Kapitel 4.6.

Durch den Befehl in **Zeile 23** wird *seqA1*, unabhängig vom vordefinierten Standard von *SequenceA*, blockierend ausgeführt.

In der Methode *action()* wird der eigentliche Ablauf der Sequenz definiert. Als erstes wird *seqB1* gestartet. Da diese, und auch *seqB2*, nicht-blockierend sind, werden sofort auch *seqB2* und *seqA1* gestartet. Weil *seqA1* den weiteren Ablauf blockiert, bis sie fertig gestellt ist, wird *seqB3* erst ausgeführt, wenn *seqA1* beendet wurde.

Mit den *join()*-Befehlen kann sichergestellt werden, dass die Hauptsequenz erst dann beendet wird, wenn die entsprechenden, parallel laufenden Sequenzen fertig abgearbeitet wurden.

4.4 Beispiel für eine benutzerdefinierte Sequenz 'SequenceA'

SequenceA.hpp:

```
1  #include "SequenceExceptionA.hpp"
2
3  namespace testappsequencer {
4
5     using namespace eeros::sequencer;
6
7     class TestAppCS;
8
9     class SequenceA : public Sequence {
10    public:
11       SequenceA(Sequencer& S, TestAppCS* CS, BaseSequence* caller, std::__cxx11::↩
             string name);
12
13       int operator()(int a, int b);
14       //int operator()(std::string str);
15       //int operator()();
16       int action();
17
18       SequenceExceptionA seqEA2;
19
20       TestAppCS* CS;
21       int posA;
22       int posB;
23    };
24 };
```

Der Aufbau dieser Sequenz ähnelt stark dem Aufbau von Hauptsequenz. Der einzige Unterschied ist die Methode *int operator()(int a, int b);*. Diese Methode wird aufgerufen, bevor die Sequenz gestartet wird und kann genutzt werden, um Parameter zu übergeben. Je nach Bedarf können die Typen und Anzahl der Parameter frei gewählt werden, oder mit *int operator()();* ganz weggelassen werden.

SequenceA.cpp:

```
#include "SequenceA.hpp"
#include "../steps/GoTo.hpp"

using namespace testappsequencer;


SequenceA::SequenceA(Sequencer& S, TestAppCS* CS, BaseSequence* caller, std::__cxx11::string name)
: Sequence(S, caller, name), CS(CS),

seqEA2(S, CS, this, "seqEA2Step")
{
   setIsBlocking();
}


int SequenceA::operator()(int a, int b)
{
   posA = a;
   posB = b;
   return Sequence::start();  //this code is mandatory for every derived Step- and Sequence-Class
}


int SequenceA::action()
{
   //initialisation of the step 'goTo'
   GoTo goTo = GoTo(S, CS, this);
   goTo.setTimeoutTime(5);
   goTo.setTimeoutBehavior(abortOwner);
   goTo.setTimeoutExceptionSequence(&seqEA2);

   //start of the sequence
   goTo(0);
   goTo(posA);
   goTo(posB);
}
```

Mit dem Befehl *setIsBlocking()* im Konstruktor werden standardmässig alle Sequenzen der Klasse *SequenceA* blockierend ausgeführt. Der Befehl *seqA1.setIsBlocking()* aus *MainSequence.cpp* ist somit nicht notwendig.

Es ist zwingend notwendig, dass die Methode *operator()* implementiert wird. Ebenfalls erforderlich ist, dass der letzte Befehl von dieser Methode *return Sequence::start();* lautet. Wenn der Sequenz Parameter übergeben werden, dann können hier die Variablen gespeichert werden.

Am Anfang der *action()*-Methode wird ein neuer *Step* initialisiert. *Steps* verhalten sich sehr ähnlich wie Sequenzen. Sie können aber nur blockierend aufgerufen werden. Da *Steps* keinen eigenen Thread starten, brauchen sie weniger Ressourcen und können einfach mehrmals hintereinander aufgerufen werden.

4.5 Beispiel für einen benutzerdefinierten Step 'GoTo'

GoTo.hpp:

```
#include <eeros/sequencer/Step.hpp>

namespace testappsequencer {

   using namespace eeros::sequencer;

   class  TestAppCS;

   class GoTo : public Step {
   public:
      GoTo(Sequencer& S, TestAppCS* CS, BaseSequence* caller);

```

```
13        int operator()(int position);
14        int action();
15        bool checkExitCondition();
16
17        int pos;
18        TestAppCS* CS;
19    };
20 };
```

Das Interface zu einem *Step* ist gleich wie bei einer Sequenz. Neu ist hier die Methode *checkExitCondition()*, welche aber auch bei Sequenzen implementiert werden kann. Der *Step* wird erst beendet, wenn *checkExitCondition()* ein *true* zurück gibt.

GoTo.cpp:

```
1  #include "GoTo.hpp"
2  #include "../control/TestAppCS.hpp"
3  #include <eeros/sequencer/Sequencer.hpp>
4
5  using namespace testappsequencer;
6
7
8  GoTo::GoTo(Sequencer& S, TestAppCS* CS, BaseSequence* caller)
9  : Step(S, caller), CS(CS)
10 {
11
12 }
13
14
15 int GoTo::operator()(int position)
16 {
17    pos = position;
18    return Step::start();    //this code is mandatory for every derived Step- and ↩
          Sequence-Class
19 }
20
21
22 int GoTo::action()
23 {
24    CS->pathPlaner.setTarget(position);
25 }
26
27
28 bool GoTo::checkExitCondition()
29 {
30    int actPos = CS->pathPlaner.getActPos;
31    if ( abs(actPos - pos) > 10 ) return false;  //target position not yet reached
32
33    SequenceB* seqB1 = (SequenceB*)(S.getSeqenceByName("seqB1"));
34    seqB1->setLastReachedPosition(actPos);
35
36    return true;
37 }
```

Die *action()*-Methode hat in diesem Fall nur eine einfache Anweisung an das *ControlSystem*. Diese Methode darf nie, weder bei *Steps* noch bei Sequenzen, blockierend sein.

Weil *checkExitCondition()* in dieser Klasse überschrieben wurde, beendet der *GoTo-Step* erst, wenn die Position erreicht wurde. Im *SequenceA.cpp* wurde aber noch der Timeout aktiviert. Das bedeutet, dass *seqEA2* gestartet wird, wenn die Zielposition nicht innerhalb von 5 Sekunden erreicht wird.

In Zeile **33** wird demonstriert, wie ein Pointer auf eine parallel laufende Sequenz geholt werden kann. Der *Typecast* ist notwendig, damit die für die Klasse *SequenceB* spezifische Methode *setLastReachedPosition(int pos)*, (diese Methode wird in dieser Arbeit nicht definiert) verwendet werden kann.

4.6 Beispiel für einen benutzerdefinierten Monitor und Condition

Wenn ein Zustand eines Roboters überwacht werden soll, dann sind die Klassen *Monitor* und *Condition* nützliche Werkzeuge. Eine *Condition* überprüft eine oder mehrere zusammenhängende Zustände des Roboters. In unserem Beispiel wollen wir überprüfen, ob der Roboter blockiert ist. Dazu erstellen wir folgende Condition:

IsBlocked.hpp:

```
namespace testappsequencer {

   using namespace eeros::sequencer;

   class  TestAppCS;

   class IsBlocked : public Condition {
   public:
      ConditionTimeout(Sequencer& S, TestAppCS* CS);

      bool validate();

      int lastActPos;
      TestAppCS* CS;
   };
};
```

Die Methode *bool validate();* muss bei jeder *Condition* überschrieben werden.

IsBlocked.cpp:

```
#include <IsBlocked.hpp>

using namespace eeros;
using namespace eeros::sequencer;

IsBlocked::IsBlocked(Sequencer& S, TestAppCS* CS)
: Condition(seq), CS(CS)
{ }

bool IsBlocked::validate()
{
   int actPos = CS->pathPlaner.getActPos();
   if( abs(actPos - CS->pathPlaner.getSetPos()) > 10 ) { //target position not yet ↩
       reached
      if( abs(actPos - lastActPos) <= 1 ) return true;   //roboter is blocked
   }
   lastActPos = actPos;
   return false;
}
```

Mit der *Condition IsBlocked* kann jetzt einfach überprüft werden, ob der Roboter blockiert ist.

Mit einem *Monitor* können wir diese Überprüfung automatisieren. Mit dem *Monitor* lässt sich auch einfach festlegen, was passieren soll, wenn der Roboter blockiert. Das Prinzip ist das selbe wie bei einem Timeout.

Dafür ergänzen wir *MainSequence.hpp* und *MainSequence.cpp*.

MainSequence.hpp:

```
...

#include "SequenceExceptionA.hpp"
#include "IsBlocked.hpp"
#include <eeros/sequencer/Monitor.hpp>

```

```
7  ...
8        IsBlocked isBlockedCondition;
9        Moinitor isBlockedMonitor;
10
11       SequenceExceptionA seqEA1;
12       SequenceExceptionA seqEA1isBlocked; //Exception sequence, if roboter is ↩
             blocked
13
14 ...
```

Die **Zeilen 4, 5, 8, 9** und **12** wurden hinzugefügt.

MainSequence.cpp:

```
1  ...
2
3  seqEA1(S, CS, this, "seqEA1"),                    //Exception Sequence
4  seqEA1isBlocked(S, CS, this, "seqEA1isBlockt"),    //Exception Sequence, if roboter ↩
       is blocked
5  isBlockedCondition(S, CS),
6  isBlockedMonitor(this, &isBlockedCondition, repeteOwner, &seqEA1isBlocked)
7  // isBlockedMonitor(this, &isBlockedCondition, restartOwner)   //Exception Sequence ↩
       can be omitted
8  {
9     setIsNonBlocking();
10
11    seqA1.setTimeoutTime(5.1);
12    seqA1.setTimeoutExceptionSequence(&seqEA1);
13    seqA1.setTimeoutBehavior(restartOwner);
14
15    seqA1.setIsBlocking();
16
17    seqA1.addMonitor( &isBlockedMonitor );
18 }
19
20 ...
```

Die **Zeilen 4, 5, 6** und **16** wurden hinzugefügt.

Solange *seqA1* läuft, wird mit der *Condition IsBlocked* überprüft, ob der Roboter blockiert. Sobald der Roboter blockiert, wird die Sequenz *seqEA1isBlockt* ausgeführt. Die Sequenz kann genutzt werden, um die Blockierung zu lösen. Nachdem die Exception Sequenz abgearbeitet wurde, wird *seqA1*, wegen dem Verhalten *restartOwner*, wiederholt.

Folgende Verhalten sind bei einer *Exception* möglich:

- **nothing**: Die Sequenz wird ganz normal weitergeführt.
- **abortOwner**: Die Besitzer-Sequenz, hier *seqA1*, wird abgebrochen.
- **restartOwner**: Die Besitzer-Sequenz wird abgebrochen und neu gestartet.
- **abortCallerofOwner**: Die *callerSequence*, hier *mainSequence*, wird abgebrochen.
- **restartCallerofOwner**: Die *callerSequence* wird abgebrochen und neu gestartet.

5 Aufbau des Sequencers

5.1 Sequencer

Das Sequencer-Objekt bildet die Basis für den ganzen *Sequencer*. Für jede Applikation wird nur ein Sequencer-Objekt erstellt. Der Sequencer speichert automatisch einen Pointer zu jeder erstellten Sequenz. Da jeder Sequenz eine Referenz auf das Sequencer-Objekt mitgegeben wird, kann von jeder Sequenz aus auf das Sequencer-Objekt zugegriffen werden. Mit den Methoden *getSequenceByID(int ID)* und *getSeqenceByName(std::__cxx11::string name)* können Pointer auf andere Sequenzen, auch parallel laufende, geholt werden. Die Methode *getSafetySystem()* gibt einen Pointer auf das *SafetySystem* zurück.

Jede Applikation hat eine andere *ControlSystem*-Klasse. Aus diesem Grund kann kein Pointer auf das *ControlSystem* im *Sequencer* gespeichert werden, da die spezifische *ControlSystem*-Klasse zur Kompilierzeit von EEROS bekannt sein müsste.

5.2 Grundsätzlicher Ablauf

Im folgenden Kapitel wird der grundsätzliche Ablauf des *Sequencers* erklärt. Detaillierte Erklärungen zu den einzelnen Komponenten und deren Funktionen befinden sich in den weiter unten im Kapitel.

Steps und *Sequencen* basieren beide auf der Basis-Klasse *BaseSequence*. Der grundsätzliche Ablauf zur Laufzeit ist bei beiden Klassen gleich und läuft in folgender Reihenfolge ab:

1. Überprüfung von aktiven *Exceptions* der aktuellen Sequenz und von Sequenzen, die von der aktuellen Sequenz blockiert werden.
2. Die *PreCondition* wird überprüft, wenn sie vorhanden ist.
3. *action()*: Die eigentliche Aktion der Sequenz wird ausgeführt. Bei einer Sequenz können dies eine Abfolge von *Steps* und / oder Sequenzen sein. Bei einem *Step* kann es nur ein einzelner Befehl an das *ControlSystem* oder an das *SafetySystem* sein.
4. Folgende Punkte werden periodisch, typischerweise mit einer Periodendauer von 100 Millisekunden, überprüft:
   - *checkExitCondition()*: Sobald diese *Condition* zutrifft, wird die Sequenz oder der *Step* beendet. Ist diese *Condition* nicht definiert, wir die Sequenz oder der *Step* sofort beendet.
   - *checkMonitorsOfThisSequence()*: Überprüft alle *Monitore* von der Sequenz und setzt eine *activeException*, wenn notwendig.
   - *checkMonitorsOfThisSequence()*: Überprüft alle *Monitore* von den Sequenzen, die durch die aktuelle blockiert werden, und setzt eine *activeException*, wenn notwendig.
   - *checkActiveException()*: Überprüft, ob die aktuelle, oder eine blockierte, Sequenz eine *activeException* hat. Ist dies der Fall, wird der *RunningState* der aktuellen Seqeunz entsprechend gesetzt.
5. Wenn der *RunningState* = *restarting*, dann wird die Sequenz wiederholt.

5.3 Base Sequence

Die *BaseSequence* bildet die Basis-Klasse für die Klassen *Sequence* und *Step*. Wenn in diesem Kapitel eine *Sequence* erwähnt wird, dann gilt das Geschriebene auch für einen *Step*, ausser es wird explizit etwas anderes erwähnt. Diese Klasse beinhaltet den grössten Teil der Intelligenz vom *Sequencer*. Die wichtigsten Methoden und Membervariablen, welche nicht selbsterklärend sind, werden in den folgenden Abschnitten genauer erklärt.

5.3.1 Membervariablen

```
BaseSequence* callerSequence
```

callerSequence ist ein Pointer auf die Sequenz, welche die aktuelle Sequenz erzeugt hat. Die *callerSequence* von der *MainSequence* ist ein NULL-Pointer.

```
std::vector< BaseSequence* > callerStack
```

Innerhalb von Sequenzen können neue Sequenzen erstellt werden. Es kann zum Beispiel in der Hauptsequenz eine Sequenz *SeqA1* erzeugt werden. In der *SeqA1* kann noch eine Sequenz *SeqA2* erzeugt werden.

In diesem Beispiel enthält *callerStack[0]* von *SeqA2* einen Pointer auf die Hauptsequenz und *callerStack[1]* enthält einen Pointer auf *SeqA1*. Der *callerStack* von *SeqA2* enthält aber keinen Pointer auf sich selbst.

```
std::vector< BaseSequence* > callerStackBlocking
```

Der *callerStackBlocking* ist ein Vektor mit Pointer auf alle Sequenzen, die von der aktuellen Sequenz blockiert werden.

Wird, wie im vorherigen Beispiel *SeqA1* nicht-blockierend gestartet, *SeqA2* aber blockierend, dann läuft die Hauptsequenz unabhängig von *SeqA1* und *SeqA2* weiter. *SeqA1* wird aber von *SeqA2* blockiert und läuft erst weiter, wenn *SeqA2* beendet ist.

Unter diesen Umständen hat der *callerStackBlocking* von *SeqA2* nur einen Eintrag mit einem Pointer auf *SeqA1*, da nur diese Sequenz von *SeqA2* blockiert wird.

```
bool exceptionIsActive
```

Die Membervariable *lstlisting* wird *true* gesetzt, wenn die Sequenz wegen einer *Exception* abgebrochen oder neu gestartet wird. Eine solche Exception wird von einem Monitor ausgelöst. Die Funktion eines Monitors und der Zusammenhang mit einer *Exception* wird im Kapitel 5.7 genauer beschrieben.

```
Monitor* activeException
```

Ein Pointer auf den *Monitor* der aktiven *Exception*.

```
std::vector< Monitor* > monitors
```

Alle *Monitors*, die zur Sequenz gehören. Mehr dazu im Kapitel 5.7.

```
MonitorTimeout monitorTimeout
```

Der *monitorTimeout* ist für die Überwachung des Timeouts zuständig. Mehr dazu im Kapitel 5.7.4.

```
ConditionTimeout conditionTimeout
```

Die zum *monitorTimeout* gehörende *Condition*. Mehr dazu im Kapitel 5.7.4.

```
int pollingTime
```

Nach dem *action()*-Teil werden in regelmässigen Abständen die relevanten *Monitore* überprüft. Die *pollingTime* beschreibt in Millisekunden, wie lange gewartet wird, bevor alle *Monitore* erneut überprüft werden.

```
runningStateEnum runningState
```

Der *runningState* beschreibt den aktuellen Zustand der Sequenz. Folgende Zustände sind möglich:

- idle
- running
- paused
- aborting
- aborted
- terminated
- restarting

Besonders in der Methode *actionFramework()* spielt der *runningState* eine grosse Rolle.

5.3.2 Virtuelle Methoden

```
virtual int start() = 0
```

Diese Methode wird aufgerufen, wenn die Sequenz gestartet wird. Sie ruft die Methode *actionFramework()* auf. Die Methode wird von den abgeleiteten Klassen *Step* und *Sequence* überschrieben.

```
virtual bool checkPreCondition()
```

Es kann sein, dass eine Sequenz erst gestartet werden darf, wenn bestimmte Bedingungen erfüllt sind. Werden die definierten Bedingungen nicht erfüllt, wird die Sequenz übersprungen. Wenn die Methode nicht überschrieben wird, wird die Sequenz ohne Überprüfung ausgeführt.

In einer benutzerdefinierten Sequenz kann die Methode überschrieben werden. Die Sequenz wird dann nur ausgeführt, wenn der Rückgabewert einem booleschen *true* entspricht. Bei einem Rückgabewert *false* wird die Sequenz übersprungen.

```
virtual bool checkExitCondition();
```

Ein *Step*, oder eine *Sequenz*, wird erst dann beendet, wenn diese Methode den Wert *true* zurück gibt, oder wenn der *Step* durch eine *Exception* abgebrochen wird. Wird *checkExitCondition()* nicht überschrieben, dann wird eine *Sequenz* sofort nach der *action()*-Methode, die bei einem *Step* nur ein kurzer Befehl ans *ControSystem* sein sollte, beendet.

checkExitCondition() kann bei einer benutzerdefinierten *Sequenz* überschrieben werden. Eine mögliche Implementierung wäre bei einem benutzerdefinierten *Step* "*moveTo(x, y)*", dass der *Step* erst dann abgeschlossen ist, wenn der Roboter seine Position erreicht hat.

```
virtual int action() = 0
```

Diese Methode wird von allen benutzerdefinierten Sequenzen überschrieben und enthält den Hauptteil vom Code, den der *Steuerungsentwickler* schreibt.

Ein *Step* soll in einer Anwendung eine ganz spezifische Aufgabe übernehmen. Wenn der Roboter zu einer bestimmten Koordinate fahren soll, dann wäre *moveTo(x, y)* ein möglicher, benutzerdefinierter *Step*. In der benutzerdefinierten Klasse *MoveTo* würde man dann die *action()*-Methode überschreiben

und die notwendigen Befehle an das *ControlSystem* eingefügt. Die *action()*-Methode selbst darf nicht blockierend sein, da ansonsten die *Monitore* während der Blockierung nicht überprüft werden.

Die *action()*-Methode einer *Sequenz* kann mehrere blockierende *Steps* und *Sequenzen* beinhalten, aber sie darf sonst keine blockierenden Befehle beinhalten. Wenn eine blockierende Funktion benötigt wird, muss ein benutzerspezifischer *Step* mit einer entsprechenden *excitCondition* benutzt werden.

5.3.3 Sonstige Methoden

```
1 int actionFramework()
```

Die *actionFramework()*-Methode wird von der *run()*-Methode aufgerufen. In dieser Methode werden alle *Monitore* und *Conditions* überprüft. Die *action()*-Methode wird von hier aus aufgerufen, wenn keine *Exception* vorliegt und die Überprüfung der *PreCondition* erfolgrich war. Wenn erforderlich, wird *action()* mehrmals aufgerufen.

5.4 Sequence

Die Klasse *Sequence* wird von den Klassen *BaseSequence* und *ThreadSequence* abgeleitet.

Wenn eine *Sequence* erstellt wird, wird automatisch ein Thread gestartet, in dem die *run()*-Methode läuft. Der Thread wird dann sofort schlafen gelegt.

Wird die *Sequence* von der Hauptsequenz, oder einer anderen Sequenz aus nicht blockierend gestartet, dann wird die *actionFramework()*-Methode im Thread ausgeführt. Wird sie aber blockierend gestartet, dann wird die *actionFramwork()*-Methode in der *start()*-Methode aufgerufen, welche die *callerSequence* blockiert.

5.5 Step

Im Gegensatz zur *Sequence* wird die Klasse *Step* nur von der *BaseSequence* und nicht von einem *Thread* abgeleitet.

Ein *Step* erzeugt nie einen Thread und wird mit *start()* immer blockierend gestartet.

Ein *Step* hat keinen Namen und wird auch nicht im *Sequencer* registriert. Andere *Steps* oder Sequenzen können also nicht auf einen *Step* zugreifen.

5.6 Condition

Condition ist eine einfache Klasse, die als Basis-Klasse für benutzerdefinierte *Condition*-Klassen dient. *Condition*-Objekte werden für *Monitoren*, siehe Kapitel 5.7, gebraucht.

In einer solchen Klasse soll ein beliebig komplexer oder einfacher Zustand überprüft werden. Beispiele für solche Zustände sind:

- Endanschlag erreicht.
- Taster mit Toggle-Funktion. Die *Condition* ist *true*, wenn der Taster einmal gedrückt wird. Wird der Taster erneut gedrückt, ist sie *false*.
- Der Roboter hat sich in den letzten 5 Sekunden nicht bewegt.

Solche Klassen erlauben es, einfache und komplexe Zustände einheitlich in einer Klasse zu abstrahieren.

```
1 virtual bool validate() = 0
```

Diese Methode wird vom *Steuerungsentwickler* überschrieben. Alle Überprüfungen von Zuständen im *ControlSystem*, oder von anderen Sequenzen, finden hier statt. In dieser Methode können auch Zustände oder Variablen gespeichert werden und Zeitmessungen durchgeführt werden.

```
1 bool isTrue()
```

Überprüft, ob eine *Condition* wahr ist, indem sie die *validate()*-Methode aufruft.

5.7 Monitor

Monitore, oder deutsch *Beobachter*, sind ein zentraler Bestandteil vom Sequencer. Sie erlauben es, gewisse Zustände des Roboters permanent zu überwachen und zu reagieren, wenn eine Veränderung eintritt. *Monitore* eignen sich gut, um Ausnahmefälle, sogenannte *Exceptions*, abzudecken.

Ein *Monitor* wird einer *Sequence* oder einem *Step* zugewiesen. Diese *Sequence* ist dann der Besitzer, oder *owner*, des *Monitors*. Der *Monitor* wird anschliessend von allen Sequenzen und *Steps*, welche die Besitzer-Sequenz blockieren, überprüft.

Ein *Monitor* besteht aus folgenden Hauptkomponenten:

1. Condition
2. Exception Sequence
3. Behavior

5.7.1 Condition

Die *Condition* ist Zustand, der überwacht wird. Ein *Condition*-Objekt muss erzeugt werden und einem *Monitor* als Pointer übergeben werden. Die *Condition* wird regelmässig, abhängig von der *pollingTime* der *Sequence*, überprüft.

5.7.2 Exception Sequence

Eine *Exception Sequence* muss ausserhalb des Monitors erzeugt werden und als Pointer dem Monitor übergeben werden. Sobald die *Condition* wahr wird, wird die *Exception Sequence* gestartet. Es kann auch keine *Exception Sequence* übergeben werden.

5.7.3 Behavior

Das *Behavior* definiert das Verhalten nachdem die *Exception Sequence* gestartet wurde.

Folgende *Behavior* sind möglich:

1. **nothing**: Die Sequenz wird ganz normal weitergeführt.
2. **abortOwner**: Die Besitzer-Sequenz des Monitors und alle Unter-Sequenzen der Besitzer-Sequenz, werden abgebrochen.
3. **restartOwner**: Die Besitzer-Sequenz wird abgebrochen und neu gestartet.
4. **abortCallerofOwner**: Die *callerSequence*, und damit natürlich auch die Besitzer-Sequenz, werden abgebrochen.
5. **restartCallerofOwner**: Die *callerSequence* wird abgebrochen und neu gestartet.

5.7.4 Timeout Monitor

Jede Sequenz besitzt standartmässig bereits einen *Monitor*. Der *Timeout Monitor* kann verwendet werden, um bei einer Sequenz, oder bei einem *Step*, einen Timeout zu überwachen. Der *Monitor* besitzt bereits eine spezialisierte *Condition*. Die *Exception Sequence* und das *Behavior* können aber wie bei jedem anderen *Monitor* gesetzt werden.

6 Ergebnis, Fazit und Ausblick

6.1 Ergebnis

Der entwickelte Sequenzer löst viele Probleme des alten Sequenzer. Es ist nun möglich Sequenzen so zu bauen, dass sie auch für Nicht-Experten einfach verständlich sind.

Die Klasse *Condition* erlaubt es nun, komplexe zusammenhängende Zustände in einer Klasse zu Abstrahieren. Der *Steuerungsentwickler* kann eine benutzerdefinierte Klasse ableiten, die der *Applikationsentwickler* nutzen kann, ohne dass er die implementierte Funktionalität verstehen muss. Der selbe Vorteil besteht auch für *Sequenzen* und *Steps*.

Mit den neuen *Monitore* existiert eine einfache Möglichkeit, bestimmte Zustände permanent und automatisch zu überwachen. Die *Monitore* können zusammen mit den *Conditions* als Exception-Handler verwendet werden.

Eine Synchronisation und Datenaustausch zwischen mehreren parallel laufenden Sequenzen ist einfach möglich, da jetzt nur noch der Namen der gesuchten Sequenz bekannt sein muss, um einen Pointer auf die Sequenz zu erhalten.

6.2 Fazit

Ich habe sehr viel Zeit für den Pseudo-Sequenzer aufgewendet. Der Plan, den Sequenzer mit Hilfe des Pseudo-Sequnzers genau durchzuplanen und ihn dann in das Framework zu integrieren, ist aber nicht aufgegangen. Bei der Integration ins EEROS habe ich gemerkt, dass viele Konzepte nicht so funktionierten, wie ich es geplant hatte. Einige Teile konnte ich mit aufwändig in kleinen Schritten integrieren, andere Teile musste ich verwerfen und neu durchdenken, weil sie nicht möglich waren.

Die vielen unvorhergesehenen Komplikationen haben meinen Zeitplan durcheinander gebracht. Aus diesem Grund konnte ich die Software nicht ausgiebig testen.

In dieser Arbeit habe ich nicht nur vieles neues Wissen bezüglich der Programmiersprache C++ aneignen, ich habe auch eine Lektion in Software-Management gelernt. Für Software eignet sich der Ansatz, erst alles durchplanen, dann alles integrieren und am Schluss die komplette Software durchtesten, nicht gut. In meinem nächsten Software-Projekt werde ich viel mehr auf kleine, aber dafür viele Iterationen planen-implementieren-testen setzen. Diese Strategie habe ich bei dieser Arbeit leider erst am Schluss eingesetzt. Besonders auf das Testen werde ich bei der nächsten Arbeit grösseren Wert legen.

6.3 Ausblick

Der neue Sequenzer bildet eine gute Grundlage für einfach verständliche Abläufe mit sehr hoher Flexibilität. Allerdings wurde er noch nicht ausgiebig getestet, so dass keine Aussage über die Zuverlässigkeit gemacht werden kann.

Des weiteren gibt es noch einige offene Punkte:

- Genau überprüfen, ob alle Ziele erreicht wurden.
- Race-Condition. Zum Beispiel wenn mehrere parallel laufende Sequenzen auf das *SafetySystem* oder das *ControlSystem* zugreifen wollen.
- Geregelter Zugriff auf das *ControlSystem*. Evt. wird von einer Sequenz exklusiven Zugriff auf Teile des *ControlSystem* verlangt, so dass sie nicht von einer anderen Sequenz gestört wird.
- Onlinedokumentation auf der EEROS-Homepage nachführen.

Quellenverzeichnis

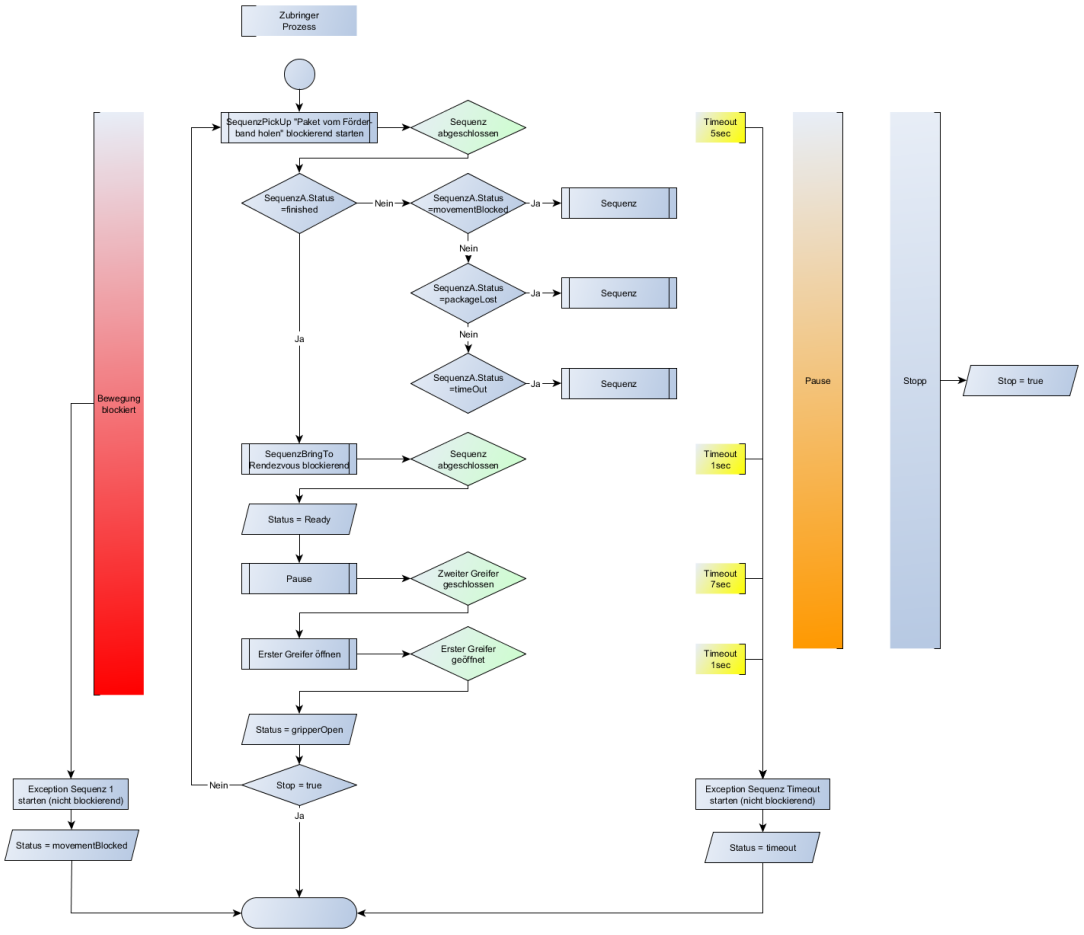
[EER-17] Homepage: EEROS
http://eeros.org
Stand vom 27.01.2017

[EEW-17] *Homepage: EEROS Wiki*
http://wiki.eeros.org
Stand vom 27.01.2017

Anhang

A Test Case 8

A.1 Sequenz Rendezvous



A.2 Sequenz PickUp