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

Inhaltsverzeichnis

1 Einleitung

1.1 Vorwort

1.2 EEROS

EEROS (Easy AAA, Elegant, Reliable, Open and Safe) ist ein open source Software Framework, welches an der NTB entwickelt wurde und auch immer noch weiter Entwicklung wird. Das Ziel von EEROS ist, möglichst einfach, 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 (Control System). 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 inversen Kinematik werden hier erledigt.
3. Der Sequencer steuert den Ablauf des Roboters. Hier werden nicht nur Wegpunkte aufgelistet, sondern auch das allgemeine Verhalten definiert.
4. Im SS (Safety System) werden sicherheitsrelevante Parameter überwacht. Das SS arbeitet unabhängig vom CS und vom Sequencer. Es löst einen Not-Aus aus, wenn der Roboter ausserhalb der zulässigen Parameter operiert. Ein möglicher Grund für einen Not-Aus wäre zum Beispiel, wenn sich der Roboterarm in einem Sicherheitsbereich zu schnell bewegt.

1.3 Klarstellung der Benennungen

Mit den 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 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 auf den Roboter anpassen. Er bereitet auch erste Sequenzen für den Applikationsentwickler vor. Oft wird die Entwicklung der Steuerung und der Applikation von der selben Person übernommen.

Um den Ablauf des Roboters anzupassen, kann der **Applikationsentwickler** bestehende Sequenzen einfach anpassen. 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 zurück gegriffen 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 beide 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 dem einfachen Aufbau soll es auch für einen Applikationsentwickler möglich sein, Sequenzen anzupassen und zu erstellen, auch wenn dieser Entwickler keine vertiefte Kenntnisse von C++ besitzt.

2 EEROS aktueller Stand

2.1 EEROS generell

2.2 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.

2.2.1 Sequencer

Die Grundlage bildet ein Sequencer-Objekt, das in einem Nicht-Realtime-Thread läuft. In so 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.

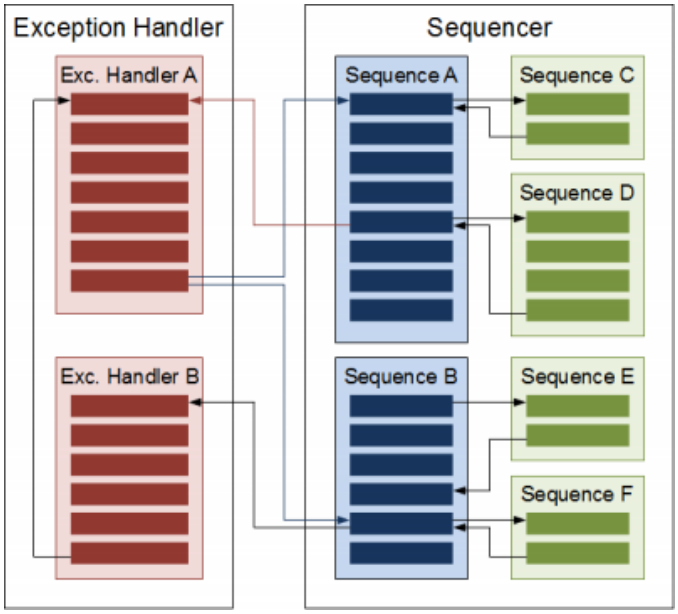


Abbildung 2.1: Schematische Darstellung des bestehenden Sequencers [EEW-17]

2.2.2 Sequence

Eine *Sequence* führt als erstes eine benutzerdefinierte Initialisierungsfunktion aus. Als nächstes wird eine *preCondition* überprüft. Fehlt 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 jeden *yield()* pausiert und wird 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();
```

```
yield();

if(!checkPostCondition())
  return SequenceResult<void>(result::postConditionFailure);
yield();

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

2.2.3 Step

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

2.2.4 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 Parameter 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.

```
class SequenceB : public Sequence<> {
public:
  SequenceB(std::string name, Sequencer* seq, Robot& r) : Sequence<void,double>(name↩
      , seq), robot(r){ }

  void run() {
    robot.moveZ(5);
    sleep(3);
    yield();
    robot.moveZ(0);
  }

private:
  Robot& robot;
};
```

2.2.5 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 zu folge soll er *Exceptions* behandeln. Je nach *Exception* soll er flexibel reagieren, um Probleme zu beheben.

2.3 Fallbeispiel 'EEDURO Delta Roboter'

Der *EEDURO Delta Roboter* ist ein Roboter, dessen Hardware 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.

2.3.1 asdf

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. Zur Zeit 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 arten von Robotern Sequenzen bauen können. Alle möglichen 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 Arten von Robotern sinnvolle Sequenzen zu erstellen. Das bedeutet, dass der *Steuerungsentwickler* möglichst viele Freiheiten behält, 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 Ereigniss, 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 *Sequencer* sollen alle Testfälle sauber und verwirklicht 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 Enschalter 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

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*

Herausforderungen

- Synchronisation von zwei parallel laufenden Sequenzen
- Kommunikation zwischen zwei Sequenzen

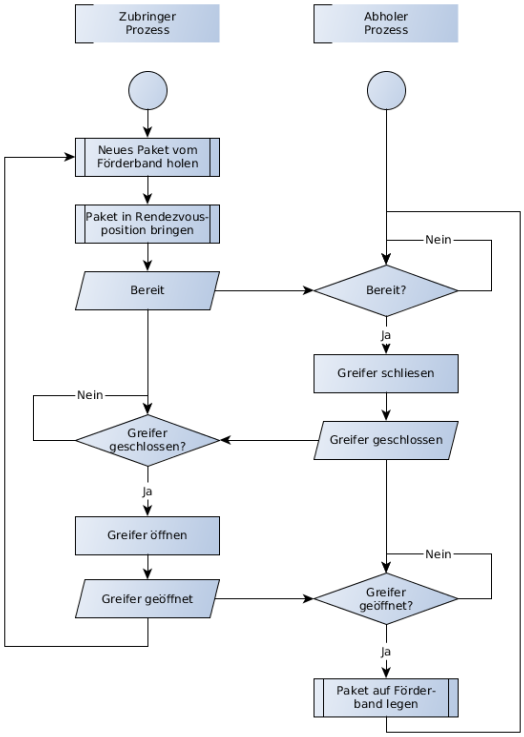


Abbildung 3.1: Ablaufdiagramm vom *Test Case 3*

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

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

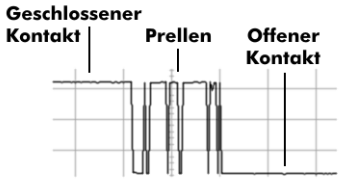


Abbildung 3.2: Signalverlauf bei einem prellenden Schalter

Herausforderungen

- Zwei parallel laufende 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*.

asdf A

asdf A

4 Aufbau des Sequencers

4.1 Caller Stack

Das unterste (älteste) Element ist die ID-Nummer der Hauptsequenz. Als nächstes folgt

5 Test des Sequencer

5.1 section

6 Fazit

6.1 section

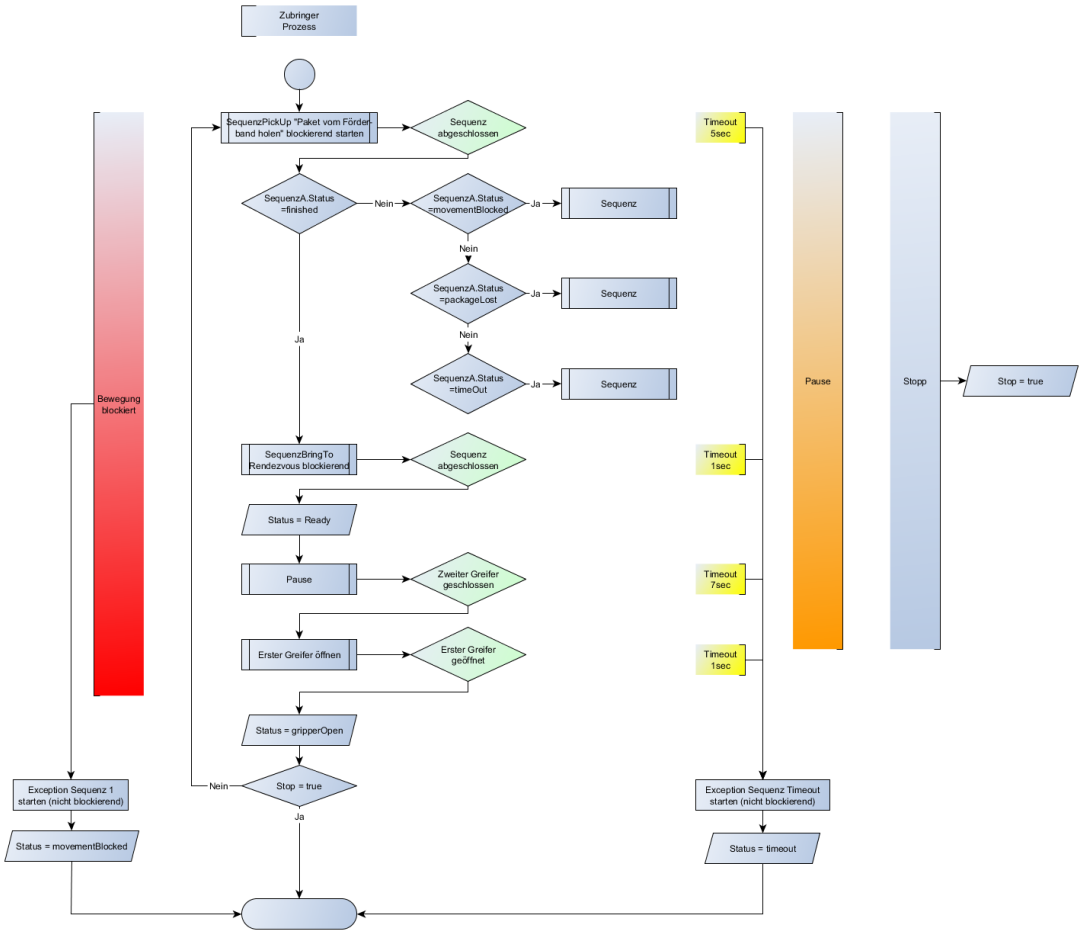
Quellenverzeichnis

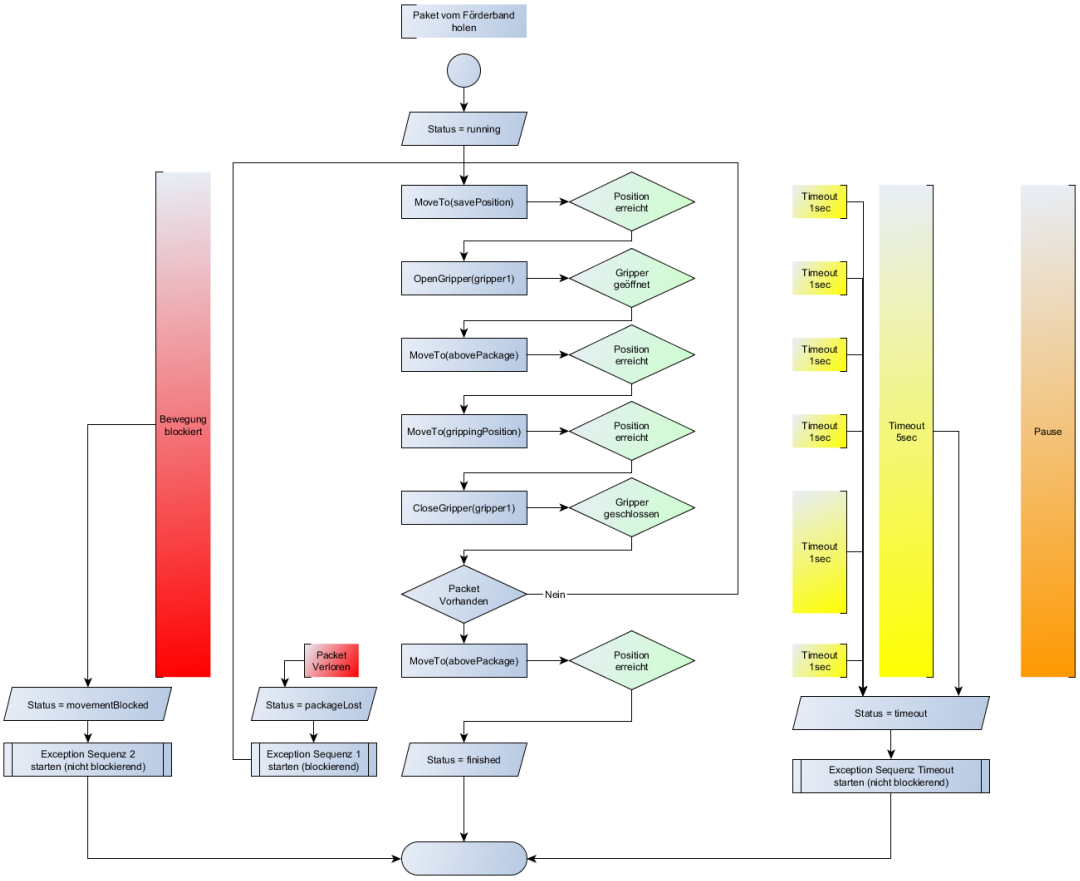
[EER-17] *Homepage: EEROS*
http://eeros.org
Stand vom 27.01.2017

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

Anhang

A Test Case 8


Zubringer Prozess
SequenzPickUp "Paket vom Förderband holen" blockierend starten
Sequenz abgeschlossen
SequenzA.Status =finished
Nein
SequenzA.Status =movementBlocked
Ja
Sequenz
Nein
SequenzA.Status =packageLost
Ja
Sequenz
Nein
SequenzA.Status =timeOut
Ja
Sequenz
Ja
SequenzBringTo Rendezvous blockierend
Sequenz abgeschlossen
Status = Ready
Pause
Zweiter Greifer geschlossen
Erster Greifer öffnen
Erster Greifer geöffnet
Status = gripperOpen
Nein
Stop = true
Ja
Exception Sequenz 1 starten (nicht blockierend)
Status = movementBlocked
Bewegung blockiert
Timeout 5sec
Timeout 1sec
Timeout 7sec
Timeout 1sec
Exception Sequenz Timeout starten (nicht blockierend)
Status = timeout
Pause
Stopp
Stop = true


Paket vom Förderband holen
Status = running
MoveTo(savePosition)
Position erreicht
OpenGripper(gripper1)
Gripper geöffnet
MoveTo(abovePackage)
Position erreicht
MoveTo(grippingPosition)
Position erreicht
CloseGripper(gripper1)
Gripper geschlossen
Packet Vorhanden
Nein
MoveTo(abovePackage)
Position erreicht
Status = finished
Bewegung blockiert
Status = movementBlocked
Exception Sequenz 2 starten (nicht blockierend)
Packet Verloren
Status = packageLost
Exception Sequenz 1 starten (blockierend)
Timeout 1sec
Timeout 1sec
Timeout 1sec
Timeout 1sec
Timeout 1sec
Timeout 1sec
Timeout 5sec
Pause
Status = timeout
Exception Sequenz Timeout starten (nicht blockierend)