Inhaltsverzeichnis

1 Dokumentation für Steuerungsentwickler

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

1.1 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

...
```

1.2 Benutzerdefinierte Klassen

Der Kern des neuen *Sequencers* sind 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 Kapitel wird mit Hilfe von einem Pseudo-System demonstriert, wie diese Klassen verwendet werden können.

1.3 Beispiel für eine mainSequence

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 benutzerdefiniert 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 Namen 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 findet sich im Kapitel 1.6.

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

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

1.4 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 von dieser Sequenz ähnelt stark dem Aufbau von der 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 von der Klasse *SequenceA* blockierend ausgeführt. Der Befehl *seqA1.setIsBlocking();* aus *MainSequence.cpp* ist somit nicht notwendig.

Es ist zwingen 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.

1.5 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, dass die für die Klasse *SequenceB* spezifische Methode *setLastReachedPosition(int pos)* (diese Methode wird in dieser Arbeit nicht definiert) verwendet werden kann.

1.6 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 auch einfach festlegen, was passieren soll, wenn der Roboter blockiert. Das Prinzip ist das gleiche 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 Verhalte *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.

Anhang