Inhaltsverzeichnis

1 Aufbau des Sequencers

1.1 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 eine 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.

1.2 Grundsätzlicher Ablauf

Im folgenden Abschnitt wird der grundsätzliche Ablauf des *Sequencers* erklärt. Detaillierte Erklärungen zu den einzelnen Komponenten und deren Funktionen befinden sich in den nachfolgenden Kapiteln.

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, dann wird der *RunningState* der aktuellen Seqeunz entsprechend gesetzt.

5. Wenn der *RunningState* = *restarting*, dann wird die Sequenz wiederholt.

1.3 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.

1.3.1 Membervariablen

```
1 BaseSequence* callerSequence
```

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

```
1 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* ein Pointer auf die Hauptsequenz und *callerStack[1]* enthält einen Pointer auf *SeqA1*. Der *callerStack* von *SeqA2* enthält aber keinen Pointer auf sich selbst.

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

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

Wird im vorherigen Beispiel *SeqA1* nicht-blockieren 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.

```
1 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 1.7 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 1.7.

```
MonitorTimeout monitorTimeout
```

Der *monitorTimeout* ist für die Überwachung des Timeouts zuständig. Mehr dazu im Kapitel 1.7.4.

```
ConditionTimeout conditionTimeout
```

Die zum *monitorTimeout* gehörende *Condition*. Mehr dazu im Kapitel 1.7.4.

```
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.

1.3.2 Virtuelle Methoden

```
1 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.

```
1 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 *Sequence* sofort nach der *action()*-Methode, die bei einem *Step* nur ein kurzer Befehl ans *ControSystem* sein soll, beendet.

checkExitCondition() kann bei einer benutzerdefinierten *Sequence* überschrieben werden. Eine mögliche Implementierung wäre bei einem benutzerdefinierten *Step* "*moveTo(x, y)*", dass der *Step* erst dann abgeschlossen ist, wenn der Roboter die 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 Methode *action()* 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 Methode *action()* von einer *Sequenz* kann mehrere blockierende *Steps* und *Sequenzen* beinhalten, aber sie darf sonst keine blockierende Befehle beinhalten. Wenn eine blockierende Funktion benötigt wird, muss ein benutzerspezifischer *Step* mit einer entsprechenden *excitCondition* benutzt werden.

1.3.3 Sonstige Methoden

```
int actionFramework()
```

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

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

1.5 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 gestarted.

1.6 Condition

Condition ist eine einfache Klasse, die als Basis-Klasse für benutzerdefinierte *Condition*-Klassen dient. *Condition*-Objekte werden für *Monitoren*, siehe Kapitel 1.7, gebraucht.

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

1.7 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

1.7.1 Condition

Der 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.

1.7.2 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* gestarted. Es kann auch keine *Exception Sequence* übergeben werden.

1.7.3 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, wird abgebrochen.
3. **restartOwner**: Die Besitzer-Sequenz wird abgebrochen und neu gestartet.

4. **abortCallerofOwner**: Die *callerSequence*, und damit natürlich auch die Besitzer-Sequenz, wird abgebrochen.

5. **restartCallerofOwner**: Die *callerSequence* wird abgebrochen und neu gestartet.

1.7.4 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.

Anhang