technische universität
dortmund

Endbericht

PG 582 - Industrial Programming by Example

Agata Berg, Cedric Perez Donfack, Julian Gaedecke,
Eike Ogkler, Steffen Plate, Katharina Schamber,
David Schmidt, Yasin Sönmez, Florian Treinat, Jan
Weckwerth, Patrick Wolf, Philip Zweihoff

27. Mai 2015

Betreuer:
Dipl.-Inf. Michael Lybecait
Dipl.-Inf. Stefan Naujokat
Prof. Dr. Bernhard Steffen

Technische Universität Dortmund
Fakultät für Informatik
LS 5: Lehrstuhl für Programmiersysteme
http://ls5-www.cs.tu-dortmund.de

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation und Problemstellung

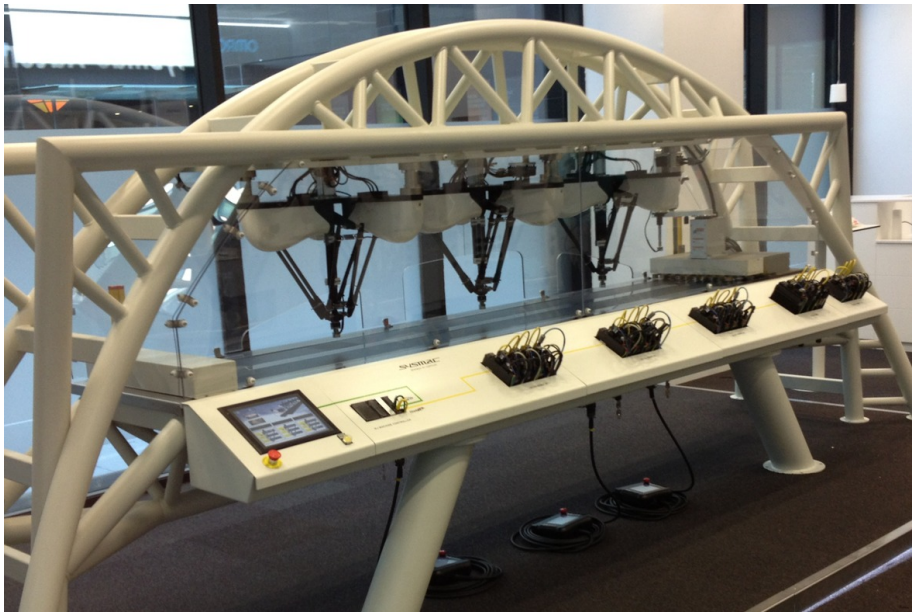


Abbildung 1.1: Omron „Pick and Place“-Anlage mit drei Delta-Robotern [7].

Das Themengebiet der Robotik zielt darauf ab die schweren, eintönigen und gefährlichen Arbeiten nicht über einen Menschen, aber über einen Roboter zu erledigen. Eine Form davon sind Verpackungsautomaten, die schnell diverse Produkte in Verpackungen deponieren können. Diese Projektgruppe konzentriert sich auf einen Delta-Roboter, welcher in Abbildung 1.1 zu sehen ist. Dieser kann „Pick and Place Packaging“-Aufgaben übernehmen, also Produkte wie zum Beispiel Kekse oder Pralinen auf einem Fließband mithilfe einer Vision-Komponente identifizieren, sie greifen und in die vorgesehene Schachtel legen.

Zuverlässige und effiziente Programme für solche Anlagen zu konstruieren, ist nicht nur komplex, sondern auch ein immer wiederkehrender Prozess, der nach jeder Änderung ei-

nes Verpackungsszenarios geändert werden muss. Außerdem besteht das Problem, dass die Sprache zum Erstellen der Programme meist nur von spezialisierten Entwicklern beherrscht wird. Diejenigen Domänenexperten für Verpackungsautomation, welche sich mit dem Erstellen von effizienten Verpackungsalgorithmen befassen, verfügen jedoch meist nicht über die nötigen Fähigkeiten, diese Programme zu schreiben.

Aufgrund dessen ist ein Ziel dieser Projektgruppe eine Modellierungsoberfläche zu implementieren, mit der genau solche Sortierungsszenarien erstellt und daraufhin visualisiert werden können. Mit Hilfe dieser Modellierungsoberfläche wird es den Domänenexperten ermöglicht, ohne vertiefte Kenntnisse über die Sprache der Programme, die Spezifikation von Verpackungsabläufen zu vereinfachen. Aus diesen Modellen wird Code generiert, welcher dann in einer Visualisierungsumgebung dargestellt wird.

Um das Vorgehen realitätsnäher zu gestalten, soll außerdem maschinelles Lernen der Delta-Roboter unterstützt werden. Hierzu wird die Möglichkeit geboten durch ein interaktives Verfahren in der Visualisierungsumgebung den Delta-Robotern ein bestimmtes Verhalten anzueignen. Benutzer befinden sich somit in einer Nachbildung der realen Produktionsumgebung und brauchen daher keine zusätzlichen Kenntnisse. Den Beginn bildet ein Programm wo kein Gegenstand auf dem Fließband gegriffen wird. Daraufhin kann durch ein iteratives Verfahren den Robotern ein bestimmtes Verhalten angelernt werden, bis schließlich das gewünschte Ergebnis geliefert wird. Zur weiteren Optimierung wird der erlernte Algorithmus unmittelbar durch Domänenexperten evaluiert.

Nachfolgend sind die Ziele der Projektgruppe noch einmal zusammenfassend aufgelistet:

- Entwicklung einer Modellierungsoberfläche zur Beschreibung von „Pick and Place Packaging“-Aufgaben und Maschinenkonfigurationen (3PL)
- Realisierung einer Visualisierungsumgebung, die 3PL-Programme ausführen kann
- Entwicklung eines Frameworks zur iterativen 3PL-Inferenz aus Beispielen und Gegenbeispielen

1.2 Konzepte und Ideen

Aus der Motivation bildet sich das zentrale Problem als die Erlernbarkeit von beliebigem Verhalten eines Delta-Roboters mit Hilfe von Beispielen und Bewertungen heraus. Die erste Abstraktion von der grundlegenden Programmierung eines solchen Delta-Roboters ist die Einführung einer Zwischenschicht, welche beherrschbarer ist, als die Programmierung der Roboter in der vorgesehenen Sprache. Diese Zwischenschicht ist eine domänenspezifische Sprache (3PL).

Die Sprache 3PL wurde mit dem Gedanken konzipiert, dass Menschen weniger in mathematischen oder logischen Strukturen, sondern mit den Objekten ihrer Domäne denken.

3PL beinhaltet deshalb Aktionen, welche ein konkreter Delta-Roboter ausführen kann. Die Verbindungen zwischen den einzelnen Aktionen sind wie ein Fließband zu verstehen, auf dem ein Gegenstand entlangläuft. Das passende Äquivalent in der Informatik ist ein Kontrollflussgraph. Das Verhalten, welches wiederum über diesen beschrieben wird, wird in einer Visualisierungsumgebung dargestellt.

Die Visualisierungsumgebung dient zunächst der reinen Visualisierung und zur Überprüfung der in 3PL modellierten Programme. In dieser Umgebung können Objekte zu Szenen arrangiert werden. Es wird eine eingeschränkte Szene entwickelt, welche vom Aufbau her die grundlegenden Elemente einer „Pick and Place Packaging"-Aufgabe beinhaltet. Die Szene wird auf zwei Delta-Roboter, 50 Gegenstände, ein Fließband, eine Kiste pro Delta-Roboter sowie fünf Charakteristika pro Gegenstandstyp beschränkt. In Abbildung 1.2 ist diese Szene zu sehen. Diese Einschränkungen machen den Entscheidungsraum für eine Projektgruppe handhabbar, schränken aber die Gültigkeit der entwickelten Lösung auch auf größere Szenarien nicht ein. So wurden alle Komponenten mit dem Hintergedanken entwickelt, auch mehr als die hier genannten Szenenobjekte zu unterstützen. Neben der reinen Darstellung ermöglicht die Visualisierungsumgebung auch die Modellierung zahlreicher Sortieralgorithmen in realitätsnaher Umgebung.

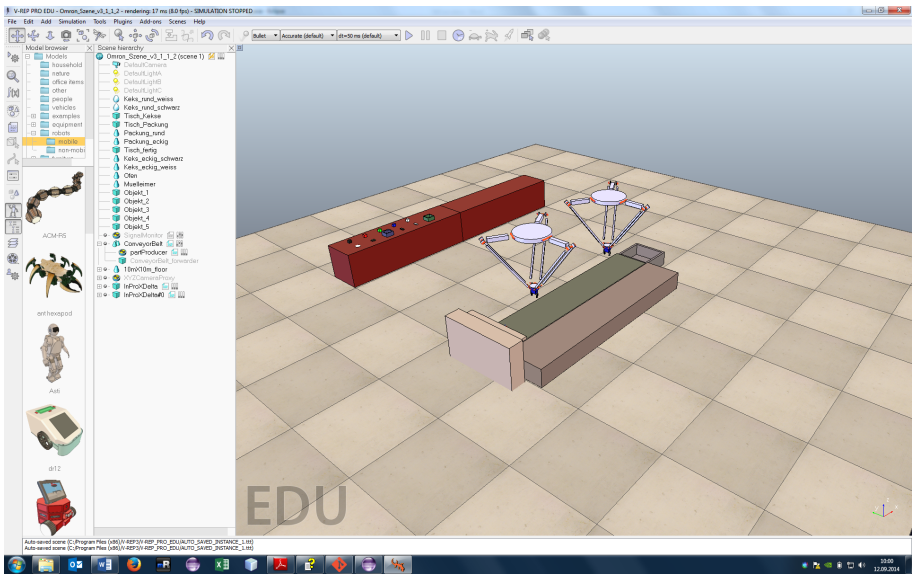


Abbildung 1.2: V-REP Szene.

Der *Lernroundtrip* ist in eine *Entscheidungs-* und in einer *Vetophase* aufgeteilt. Das Konzept dazu entsteht aus der Voraussetzung, dass dem Anwender eine Möglichkeit gegeben werden muss um Beispiele zu generieren (Entscheidung) und diese zu bewerten (Veto). Die beiden Phasen werden über eine zentrale GUI gesteuert. Dabei ist die Idee nicht einen absolut starren Ablauf zu entwickeln, sondern es in der Hand des Domänenexperten zu legen, ob das präsentierte Ergebnis mehr von Vetos oder einer neuen Beispielmenge profitiert.

Die Entscheidung sollte ihm auch dahingehend leicht fallen, da nur gemachte Aktionen bewertet werden können. Wird also ein Gegenstand überhaupt nicht gegriffen, hilft dort nur eine neue Beispielmenge. Die GUI ist entsprechend in die Module zerlegt, welche der Lernroundtrip benötigt. Eine vollständige Übersicht wird in Kapitel 1.3 beschrieben. In diesem Kapitel ist auch das Mapping zwischen den einzelnen Komponenten und den folgenden Kapiteln in diesem Endbericht näher erläutert.

1.3 Aufbau der Arbeit

Der Aufbau der Arbeit orientiert sich an Abbildung 1.3, welche die zentralen Komponenten der Projektgruppe InProX beinhaltet. Das heißt in den folgenden Kapiteln werden alle Bestandteile der Abbildung 1.3 beschrieben.

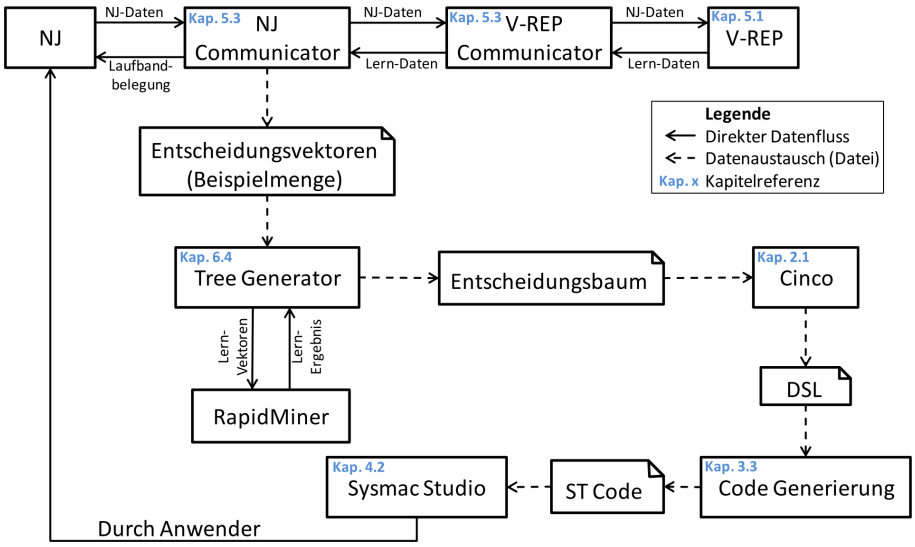


Abbildung 1.3: Überblick über die Gesamtarchitektur.

Kapitel 2 geht auf die Entwicklung einer domänenspezifischen Sprache (DSL) zur Beschreibung von „Pick and Place Packaging“-Aufgaben, sowie auf die Entwicklung eines Editors für die DSL, welche zur Modellierung von Sortieralgorithmen als Ablaufdiagramme verwendet werden kann. Dieser Editor wurde mit Hilfe des Frameworks Cinco (siehe Abbildung 1.3) erstellt, welches eine Softwarelösung zur Erstellung einer graphischen DSL darstellt. Im anschließenden Kapitel 3 wird die Erstellung des Codegenerators, welcher die domänenspezifische Sprache in Quellcode der Zielsprache Structured Text (ST) übersetzt, erläutert. Kapitel 4 beschreibt die Entwicklung eines Rahmenprogramms in Sysmac Studio, in welches das in der DSL entworfene und in ST übersetzte Programm ohne weitere Anpassungen eingefügt werden kann. In Kapitel 5 wird auf die Visualisierungskomponente eingegangen, mit deren Hilfe es möglich ist die Delta-Roboter, bestehend aus dem in der DSL entworfenen und in ST übersetzten Programmen zur Steuerung derselbigen, wel-

che in der NJ simuliert werden, darzustellen. Die Visualisierungskomponente setzt sich aus dem Bestandteilen *V-REP*, *V-REP Communicator*, *NJCom* und *NJ* zusammen (siehe auch Abbildung 1.3). Das Kapitel 6 beschreibt die Lernkomponente, welche es dem Anwendungsentwickler für Verpackungsszenarios durch ein iteratives Verfahren in der Visualisierungsumgebung ermöglicht, den Delta-Robotern ein gewünschtes Verhalten durch sogenanntes *„Programming by Example“* anzueignen. Dieses gelernte Verhalten wird mittels der DSL beschrieben und lässt sich mit der Visualisierungskompomente darstellen sowie bewerten. Die Lernkomponente ist quer schneidend in allen in Abbildung 1.3 dargestellten Bestandteilen implementiert. Weiterhin wird in Kapitel 6 auch auf die Bestandteile *Tree Generator*, *RapidMiner*, Entscheidungsvektoren und Entscheidungsbäume eingegangen wie sie in der Abbildung 1.3 zusehen sind. Das Kapitel endet mit einer Evaluation der Lernkomponente. Die schließenden Kapitel 7 und 8 fassen alle zentralen Punkte dieses Endberichts zusammen und geben einen Ausblick.

Kapitel 2

Pick and Place Packaging Language (3PL)

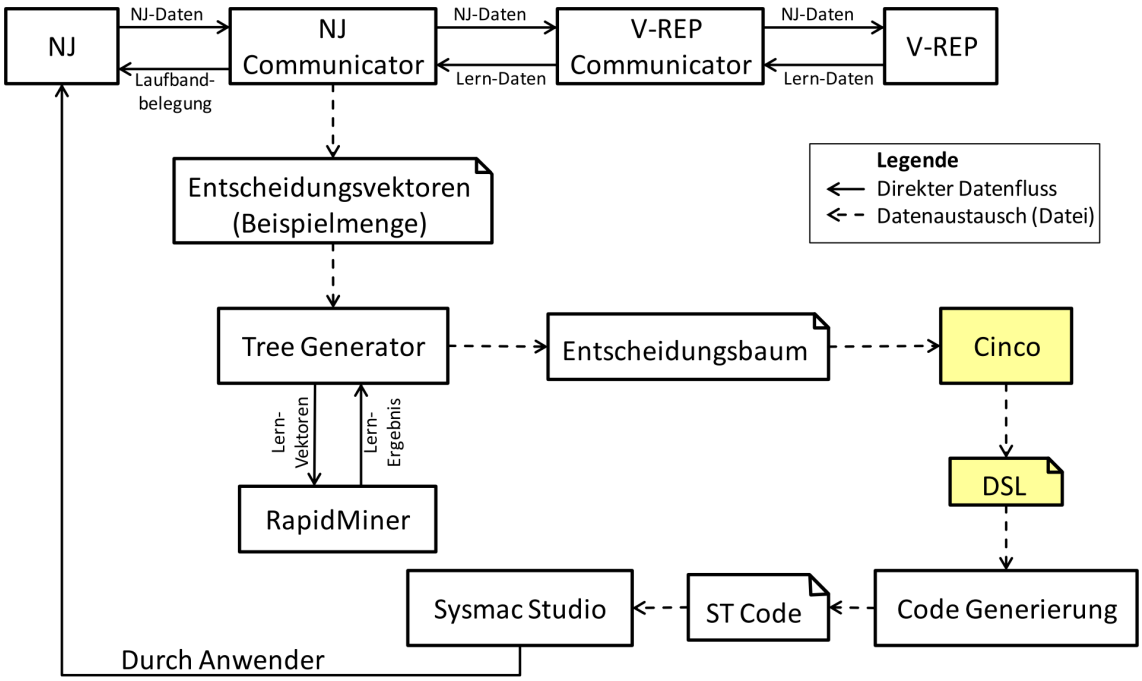


Abbildung 2.1: Einordnung der DSL in der Gesamtarchitektur.

In diesem Kapitel wird der graphische Editor zur Modellierung von Ablaufdiagrammen zur Erstellung von Sortieralgorithmen beschrieben. Dieser Editor wurde mit Hilfe des Frameworks Cinco erstellt, welches eine Softwarelösung zur Erstellung einer graphischen DSL darstellt. Die DSL ist dabei ein möglicher Startpunkt bei der Umsetzung, woraus anschließend der Code zur Simulation generiert wird. Eine Ausnahme bildet das Lernen eines Verpackungsszenarios, dort ist Cinco in den Prozess eingebettet, wie auf Abbildung 2.1 zu

sehen. Da wird die DSL aus dem Entscheidungsbaum vom Tree Generator automatisch generiert und anschließend bei der Codegenerierung aufgegriffen.

Es werden zunächst die nötigen Grundlagen, sowie das Framework Cinco zur Erstellung der DSL erläutert. Anschließend wird die generelle Herangehensweise zur Umsetzung des Editors beschrieben. Danach wird die Realisierung, sowie die verschiedenen Knoten und Kanten des resultierenden Editors visualisiert und geschildert. Abgeschlossen wird dieses Kapitel mit einem Beispiel eines Ablaufdiagramms, welches mit dem Editor erstellt wurde und zur Visualisierung von Delta-Roboters dient.

2.1 Grundlagen

In diesem Abschnitt werden die Grundlagen für die Entwicklung einer Modellierungsoberfläche geschaffen. Zum besseren Verständnis wird zunächst der Begriff Domain Specific Language (DSL) erläutert. Anschließend wird das Framework Cinco, welches zur Generierung des Editors verwendet wurde, vorgestellt.

2.1.1 Domain Specific Language (DSL)

Eine DSL ist eine (Programmier-)Sprache, die mit dem Ziel entworfen wurde, Probleme innerhalb einer festgelegten Domäne zu lösen. Diese Sprache ist genau auf diese zugeschnitten und im Idealfall lassen sich mit ihr genau die Probleme der Domäne lösen. Eine DSL ist oftmals sinnvoll, da sie anders als eine generische Programmiersprache mit Begriffen arbeitet, mit denen die Zielgruppe bereits vertraut ist. Die Ersteller denken und sprechen über Probleme meist anders. Durch die Entwicklung einer DSL können so Missverständnisse und Komplikationen verhindert werden und die Sprache kann sich an dem Anwender orientieren und erleichtert so die Einarbeitung, sowie den Umgang mit der Sprache.

Bei den graphischen DSL handelt es sich um eine Untergruppe der DSL. Durch die graphische Darstellung einer DSL wird oftmals die Nutzung und Handhabung erleichtert. Durch die Visualisierung der Daten als graphisches Modell werden viele Probleme für Anwender verständlicher abgebildet.

2.1.2 Cinco

Cinco [8], [6] ist ein aus einem Lehrstuhlprojekt entstandenes Framework für die Generierung von Tools aus high-level Spezifikationen. Es kombiniert die Einfachheit von jABC [12], die Fähigkeiten zur Metamodellierung von EMF und des Graphiti Diagramm Editor Frameworks. jABC wurde über 15 Jahre entwickelt und ermöglicht einfaches modellieren und organisieren von diversen Workflows. Cinco greift diese Eigenschaft auf und kombiniert diese mit den starken Fähigkeiten des Metamodellierungsframeworks EMF.

Durch die Verwendung von Cinco muss der Nutzer sich nicht mit internen Eclipse Strukturen und APIs befassen. Ziel von Cinco ist es domänenspezifische Werkzeuge zu generieren, die präziser zugeschnitten sind, als es mit jABC, BPMN oder UML möglich wäre und dabei gleichzeitig die komplexen technischen Details zu verstecken.

Dabei verwendet Cinco ein Vorgehen, dass sich in zwei Schritte gliedert. Zunächst wird ein Meta Graph Language Model (MGL-Modell) definiert. Dies legt fest welche Art von Modellkomponenten später im Tool verfügbar sein sollen, welche sowohl Attributdeklarationen wie auch Annotationen enthalten können. Die MGL, dessen Metamodell (MGL.ecore) und der textuelle Editor sind dabei mit EMF realisiert. Für die Generierung des Editors aus dem Metamodell wird Xtext [16] verwendet.

Nach der Definition des MGL-Modells wird ein Style-Modell entworfen. Dort wird das Aussehen, der in der MGL definierten graphischen Elemente des Editors, wie Knoten und Kanten, festgelegt. Dies wurde wiederum durch EMF und Xtext realisiert.

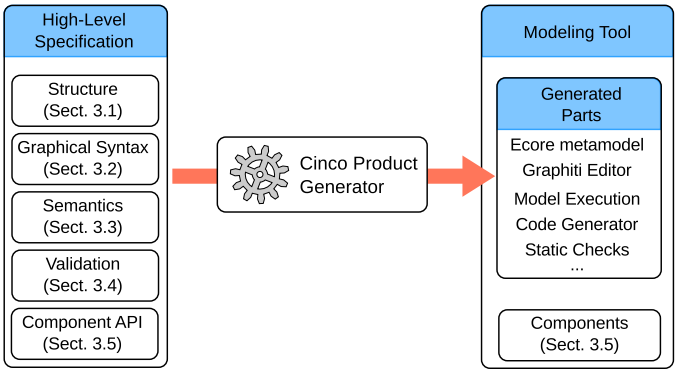


Abbildung 2.2: Cinco generiert sofort lauffähige Editoren [8].

Abbildung 2.2 stellt den Kern von Cinco dar. Es wird ermöglicht einen von vornherein lauffähigen Editor durch eine high-level Beschreibung zu generieren. Es können Knoten, Kanten und Container modelliert werden, welche wiederum Attribute oder Annotation beinhalten können.

Die Eigenschaft, dass direkt zu Beginn ein lauffähiger Editor entsteht, sowie die Einfachheit, dass ein Tool durch eine high-level Spezifikation generierbar ist und auch andere praktische Funktionen, waren für die Wahl des Frameworks ausschlaggebend.

2.2 Konzept

Zur Realisierung eines graphischen Editors, der das Modellieren von Ablaufdiagrammen ermöglicht, ist die Beschreibung der Komponenten des Editors, sowie deren graphische Umsetzung notwendig. Dies geschieht, wie bereits in den Grundlagen erläutert, durch die Implementierung eines MGL- sowie Style-Modells. Alles weitere wird durch das Framework Cinco automatisch generiert.

Hierfür sollte zunächst die grundlegende Idee der resultierenden graphischen DSL erläutert werden. Das Ziel der DSL ist das Modellieren von verschiedenen Szenarien für die Konstruktion von Sortieralgorithmen. Ein Szenario hat verschiedene, zur Erstellung notwendige Elemente, die nachfolgend beschrieben werden:

- **Charakteristik:** Eine Charakteristik beschreibt ein Merkmal eines Gegenstandes, wie die Farbe, die Form, das Material oder Ähnliches. Dabei stellt ein Charakteristikum ein Merkmal eines Gegenstandes dar, welches mit `true` oder `false` belegt werden kann. Ein Gegenstand kann somit theoretisch mehrere Farben oder Formen annehmen.
- **Gegenstandstyp:** Ein Gegenstandstyp kann bis zu fünf Charakteristika haben.
- **Delta-Roboter:** Es können bis zu zwei Delta-Roboter modelliert werden, die über dem Fließband platziert werden und Gegenstände greifen und in Kisten packen können.
- **Kiste:** Eine Kiste kann bis zu 50 Gegenstände beinhalten.
- **Platz:** Ein Platz repräsentiert eine Stelle in der Kiste, wo nur ein Gegenstandstyp abgelegt werden kann.

Nachfolgend wird die Implementierung der Modelle genauer erläutert.

2.3 Realisierung

Die Realisierung der DSL erfolgt mit Hilfe des Frameworks Cinco. Diese besteht aus zwei Abschnitten. Zunächst werden die Komponenten in der MGL-Datei beschrieben und anschließend in der Style-Datei optisch angepasst. Im Anschluss werden die implementierten Komponenten und Verbindungsarten, sowie deren Funktionsweise näher beschrieben. Abgeschlossen wird das Kapitel mit einem Beispiel.

2.3.1 Implementierung

Cinco übernimmt viele Teile der Implementierung automatisch. Abbildung 2.3 zeigt alle für diesen Editor notwendigen Modelle und Metamodelle, wobei die automatisch generierten mit inbegriffen sind.

Wie zuvor erläutert muss zunächst ein MGL-Modell (`EasyDelta.mgl`) definiert werden, aus dem sich anschließend das entsprechende Metamodell (`MGL.ecore`) generieren lässt. In diesem werden alle Knoten, Container und Kanten beschrieben.

Ein Ausschnitt aus der `EasyDelta.mgl`, mit der Implementierung des Knotens `ReturnAllItemType`, ist in Listing 2.1 zu sehen. Dieser besitzt ein Attribut, welchem später in den Eigenschaften ein Gegenstandstyp zugewiesen werden kann. Außerdem werden mit

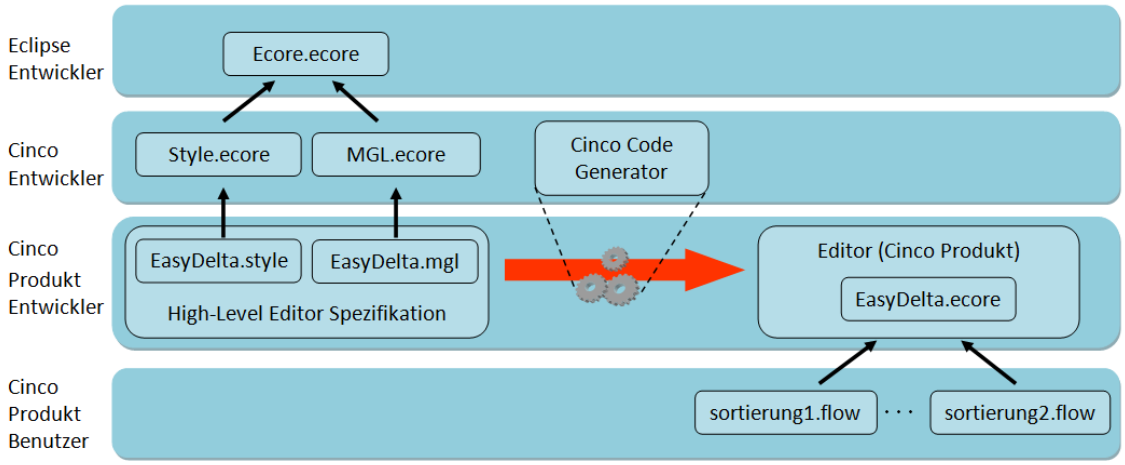


Abbildung 2.3: Übersicht der relevanten Artefakte für den Editor(adaptiert nach [8]).

`incomingEdges` und `outgoingEdges` die ein- und ausgehenden Kanten festgelegt, sowie die erlaubte maximale Anzahl dieser. Welche Funktionen die einzelnen Knoten, Container und Kanten widerspiegeln wird in den darauffolgenden Kapiteln genauer erläutert. Dies wäre bereits genug um das Metamodell `EasyDelta.ecore` zu generieren, welches die Basis für den Editor bildet.

```
1  //Delta X, return all items with item type Y
2  @style( ReturnTypeStyle, "Return with","${itemtype.name}")
3  @icon("icons/returnicontyp2_16.png")
4  @palette("Flow Nodes")
5  node ReturnAllItemType{
6    attr ItemConfiguration as itemtype
7    incomingEdges({CharacteristicSuccessor, ItemSuccessor, NotEmptySuccessor,
8        TrueSuccessor, FalseSuccessor,ControlFlow, ElseSuccessor,
9        EmptyBoxSuccessor}[0,1])
10   outgoingEdges(MultipleDataFlow[0,*], ControlFlow[0,1])
11 }
12
13 @style( LabeledEdgeStyle,"true")
14 edge TrueSuccessor{
15   }
```

Listing 2.1: Ausschnitt aus der EasyDelta.mgl

Um dies jedoch graphisch ansprechend zu gestalten, ist das Schreiben der `EasyDelta.style` notwendig, woraus daraufhin die Graphiti-Features generiert wird. Dieses Metamodell erlaubt das Beschreiben von Styles. In der `EasyDelta.style` kann jetzt das Aussehen, der zuvor, in der `EasyDelta.mgl` erstellten, Knoten, Containern und Kanten definiert werden. Dabei stellt nodeStyle, die graphische Repräsentation von Knoten und edgeStyle, die der Kanten dar. Im Ausschnitt der `EasyDelta.style` in Listing 2.2 ist das Aussehen des Knotens `ReturnAllItemType` und das Aussehen von Kanten mit Attribu-

ten definiert. Im oberen Teil ist die Form und Größe festgelegt, darauf folgend wird ein Bild, sowie Text in dem Knoten platziert. Bei der Kante wird lediglich eine Spitze gezeichnet sowie die Position des Inputs festgelegt, welches an der Kante stehen soll. Es muss nicht zwingend immer eine Style-Beschreibung zu jeder Komponente der `EasyDelta.mgl` existieren, sondern es können mehrere Komponenten das gleiche Aussehen haben.

Listing 2.1 zeigt, dass in der `EasyDelta.mgl` jeder Komponente ein Style zugeordnet wird. Um später in der Palette nicht nur die Namen der Knoten zu sehen, sondern auch ein passendes Symbol mit anzeigen zu lassen, kann in der `EasyDelta.mgl` jedem Knoten ein passendes Bild zugeordnet werden. Mit dem annotierten MGL-Modell und dem Style-Modell generiert Cinco automatisch einen graphischen Editor. Ebenso wird neben dem Editor auch die Ansicht für die Bearbeitung und Anzeige der Eigenschaften generiert.

```
nodeStyle ReturnTypeStyle(2){
  rectangle inoutRec{
    size(155,38)
  image {
    position(2,2)
    size(34,34)
    path("icons/returnicontyp2.png")}
    text t1{
      position (34,10)
      value " %s %s"}
  }
}
// style for the labeled edges
edgeStyle LabeledEdgeStyle{
  appearance {lineStyle SOLID}
  decorator {
    location(1.0)
    polyline poly{
      points [(-15,-5) (0,0) (-15,5)(-5,0) (-15,-5)]}}
  decorator {
    location(0.5)
    movable
    text t5{
      value "%s"}}
}
```

Listing 2.2: Ausschnitt aus der EasyDelta.style

Nachfolgend werden alle Komponenten des EasyDelta Editors genauer erläutert.

2.3.2 Komponenten der DSL

Die Modellierung spaltet sich in einen Konfigurationsteil und einen Ablaufteil. Zunächst sollten die benötigten Elemente erstellt werden, wie die Chrakteristika, Gegenstandstypen, Delta-Roboter sowie Boxen mit ihren Plätzen. Dies sind die Konfigurationselemente, welche erforderlich sind, um den Ablauf modellieren zu können. Der Ablauf findet nun in den

konfigurierten Delta-Roboter-Swimlanes statt. Diese beiden Knotenarten werden durch ihre Farbe unterschieden. Die Konfigurationselemente sind grün, die Komponenten, die den Ablauf darstellen, sind weiß. Von den Ablaufkomponenten können noch Gateways unterschieden werden.

In den nachfolgenden Tabellen sind alle Komponenten, die für das Modellieren eines Diagramms verwendet werden können, aufgelistet, beschrieben und visualisiert.

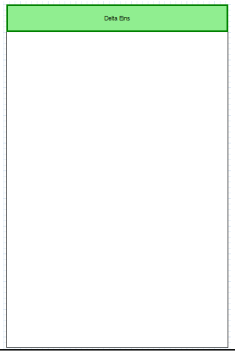
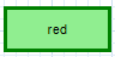
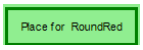
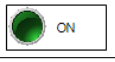
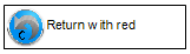
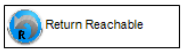
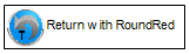
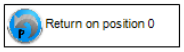
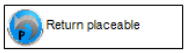
Konfigurationselement	**Funktion**	**Visualisierung**
DeltaSwimlane	Jeder Delta-Roboter besitzt eine Swimlane in der das Ablaufdiagramm für speziell diesen Delta-Roboter modelliert werden kann. Ebenso wird ein Name definiert sowie die Abfolgereihenfolge, also ob der Delta-Roboter sich an erster oder zweiter Stelle befindet, festgelegt.	
CharacteristicConfiguration	Hiermit wird eine Charakteristik mit einem Namen definiert.	
ItemConfiguration	Es kann ein Gegenstandtyp mit Namen definiert werden. Diesem Gegenstand können die vorher definierten Charakteristika zugewiesen werden.	
BoxConfiguration	Es kann eine Box mit einem Namen definiert werden. Diese enthält verschiedene Plätze von Gegenstandstypen. Jedem Delta-Roboter wird eine Box zugewiesen.	
PlaceConfiguration	Ein Platz repräsentiert einen Platz in der Box. Dabei kann der Gegenstandstyp festgelegt sowie dessen Anzahl und Drehung bestimmt werden. Diese Plätze werden mittels Drag & Drop in die zugehörige Kiste gezogen.	

Tabelle 2.1: Konfigurationskomponenten der DSL.

Flusskomponente	**Funktion**	**Visualisierung**
Start	Startet den Ablauf eines Delta-Roboters.	

Stop	An dieser Stelle soll die Anlage automatisch heruntergefahren werden.	
GrabAndPut	Der Delta-Roboter nimmt den übergebenen Gegenstand vom Fließband und legt diesen in einen geeigneten Platz der vordefinierte Box des Delta-Roboters.	
HomePosition	Der Delta-Roboter fährt in seine Ausgangsposition.	
Wait	Der Delta-Roboter wartet die definierte Anzahl von Millisekunden ab bevor dieser dem Kontrollfluss weiter folgt.	
ReturnAllCharacteristic	Es wird eine Liste aller erreichbaren Gegenstände auf dem Fließband zurückgegeben, die die ausgewählten Charakteristika besitzen. Zur Auswahl stehen nur die vorher definierten Charakteristika.	
ReturnAllReachableItems	Es wird eine Liste aller erreichbaren Gegenstände eines Delta-Roboters auf dem Fließband zurückgeliefert.	
ReturnAllItemType	Es wird eine Liste aller erreichbaren Gegenstände auf dem Fließband zurückgegeben, die dem ausgewählten Gegenstandtyp entsprechen. Zur Auswahl stehen nur die vorher definierten Gegenstandtypen.	
ReturnItem	Es wird der Gegenstand, mit der definierten Position in der mitgelieferten Liste, zurückgegeben.	
ReturnAllPlaceableItems	Es wird eine Liste aller erreichbaren und platzierbaren Gegenstände eines Delta-Roboters auf dem Fließband zurückgeliefert.	
JoinDataFlow	Es wird eine Liste mit allen, in diesen Knoten eingehenden Listen enthaltenden Elementen, zurückgeliefert.	

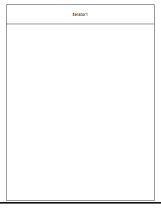
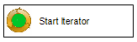
Intersection	Es wird eine Liste mit lediglich den gemeinsamen Elementen, aller in diesen Knoten eingehenden Listen, zurückgeliefert.	
JoinControlFlow	Mehrere Kontrollflüsse werden wieder zusammengeführt.	
ItemIterator	Ermöglicht es bestimmte Elemente einer Liste in einer neuen Liste zu speichern. Es wird für jedes Element der Liste entschieden, ob es in die neue Liste hinzugefügt wird oder nicht.	
StartIterator	Signalisiert den Start einer Iteration des Iterators. Davon ausgehend kann mit einem Gegenstand der Liste gearbeitet werden.	
EndIterator	Schleife des Iterators wird beendet und der Gegenstand wird nicht in die neue Liste hinzugefügt.	
AddToTempList	Eingehende Elemente werden in die neue Liste hinzugefügt.	

Tabelle 2.2: Komponenten der DSL.

Gateway	**Funktion**	**Visualisierung**
Item	Hier kann eine Fallunterscheidung eines Gegenstandes über den Typ des Gegenstandes getroffen werden. Eine else-Kante muss ebenfalls modelliert werden	
Characteristic	Hier kann eine Fallunterscheidung eines Gegenstandes über die Charakteristik des Gegenstandes getroffen werden. Eine else-Kante muss ebenfalls modelliert werden	

ListElements	Eine XOR Verzweigung prüft ob eine Liste leer ist oder mindestens eine definierte Anzahl an Elementen hat. Eine else-Kante muss ebenfalls modelliert werden	
BoxDimensions	Hier kann geprüft werden ob noch freie Plätze in der Box des Delta-Roboters verfügbar sind.	

Tabelle 2.3: Verzweigungen der DSL.

Prime-Referenzen

Prime-Referenzen sind ein technisches Konzept von Cinco. Es ermöglicht es extern definierte Komponenten über drag&drop im Diagramm zu platzieren, woraufhin ein anderer Knoten erzeugt wird, welcher automatisch mit dem externen Element gekoppelt ist. Auf diese Weise können beliebige Elemente aus Fremdmodellen in den generierten Editor integriert werden, ohne, dass es erforderlich ist, den Editor oder das Metamodell anpassen zu müssen.

In diesen Projekt werden die Prime-Referenzen verwendet, um dem Benutzer zu ermöglichen, eigene Fallunterscheidungen zu implementieren und in das Diagramm einzubauen. Hierbei kann ST-Code, der zur Entwicklungszeit noch nicht bekannt ist, in die Codegenerierung eingewebt werden.

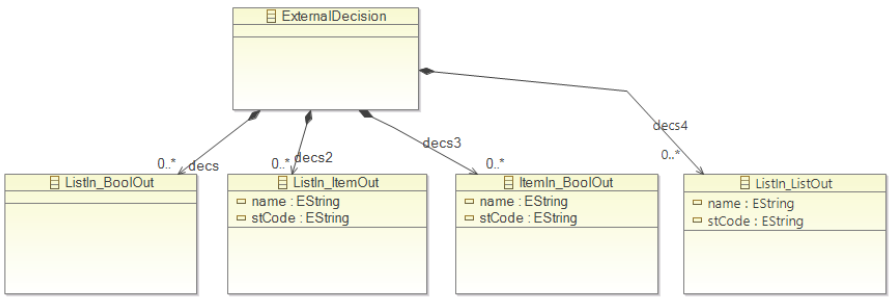


Abbildung 2.4: Metamodell der Prime-Referenz.

Für diesen Editor wurden verschiedene Prime-Referenzen erstellt. Das entsprechende Metamodell, welches allen zugrunde liegt, ist in Abbildung 2.4 zu sehen. Jedes Element besitzt einen Namen und den für die Visualisierung notwendigen ST-Code.

Es wurden drei Arten von externen Entscheidungskomponenten modelliert, welche sich durch ihre Ein- und Ausgabemöglichkeiten unterscheiden:

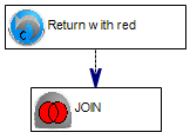
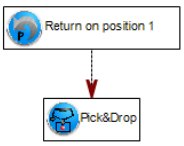
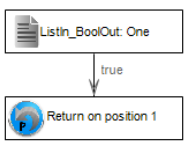
- **ListIn_BoolOut:**

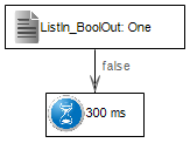
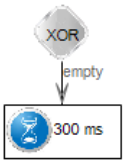
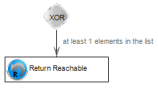
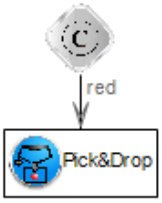
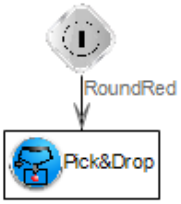
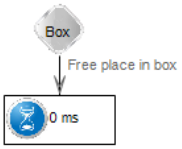
- Eingabe: Eine Liste von Gegenständen
- Ausgabe: true/false

- **ListIn_ItemOut:**
  - Eingabe: Eine Liste von Gegenständen
  - Ausgabe: Ein Gegenstand
- **ListIn_ListOut:**
  - Eingabe: Eine Liste von Gegenständen
  - Ausgabe: Eine Liste von Gegenständen
- **ItemIn_BoolOut:**
  - Eingabe: Eine Liste von Gegenständen
  - Ausgabe: true/false

2.3.3 Verbindungsarten

Um die verschieden Komponenten miteinander zu verknüpfen stehen folgende Verbindungen zur Verfügung:

Verbindung	Funktion	Visualisierung
ControlFlow	Stellt den normalen Ablauf des Diagramms dar.	
MultipleDataFlow	Die Übertragung der Listen von Gegenständen wird hiermit dargestellt.	
SingleDataFlow	Die Übertragung von einzelne Gegenständen wird hiermit dargestellt.	
TrueSuccessor	Diese Verbindung kann nach einer externen Entscheidung angewendet werden. Bei einem wahren Ergebnis wird der Ablauf hier fortgesetzt.	

FalseSuccessor	Diese Verbindung kann nach einer externen Entscheidung angewendet werden. Bei einem falschen Ergebnis wird der Ablauf hier fortgesetzt.	
EmptySuccessor	Diese Verbindung kann nur nach einem XOR angewendet werden. Der Ablauf wird hier fortgesetzt wenn die übergebene Liste der Gegenstände leer ist.	
NotEmptySuccessor	Diese Verbindung kann nur nach einem XOR angewendet werden. Der Ablauf wird hier fortgesetzt wenn die übergebene Liste der Gegenstände mindestens die angegebene Anzahl an Elementen hat.	
CharacteristicSuccessor	Diese Verbindung kann nur nach einem CharacteristicGateway angewendet werden. Es kann eine Fallunterscheidung anhand der zu Beginn definierten Charakteristika, auf Basis des übergebenden Gegenstandes, getroffen werden.	
ItemSuccessor	Diese Verbindung kann nur nach einem ItemGateway angewendet werden. Es kann eine Fallunterscheidung anhand der Typen der zu Beginn definierten Gegenstände, auf Basis des übergebenden Gegenstandes, getroffen werden.	
EmptyBoxSuccessor	Diese Verbindung kann nur nach Box-Dimensions angewendet werden. Der Ablauf wird hier fortgesetzt wenn noch mindestens ein freier Platz in der Kiste des Delta-Roboters verfügbar ist.	

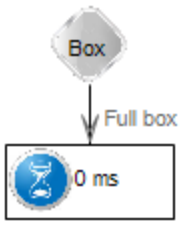
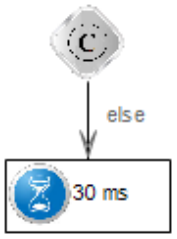
FullBoxSuccessor	Diese Verbindung kann nur nach Box-Dimensions angewendet werden. Der Ablauf wird hier fortgesetzt wenn kein freier Platz mehr in der Kiste des Delta-Roboters verfügbar ist.	
ElseSuccessor	Diese Verbindung kann nur nach einem Gateway angewendet werden. Jedes Gateway muss eine else-Kante besitzen. Der Ablauf wird hier fortgesetzt wenn keine der anderen Kanten des Gateways erfüllt sind.	
BoxDeltaConnection	Hiermit wird jedem Delta-Roboter eine Box zugeordnet.	

Tabelle 2.4: Verbindungsarten der DSL.

2.3.4 Beispiel

In den vorhergehenden Abschnitten wurden die verschiedenen Knoten, Container und Kanten des EasyDelta Editors zum Modellieren von Ablaufdiagrammen erläutert. Abbildung 2.5 zeigt den generierten Editor. Auf der linken Seite ist der Explorer zu sehen, der die aktuellen Projekte mit den Ablaufdiagrammen und Externen Bibliotheken beinhaltet. Darunter ist eine Übersicht des Diagramms abgebildet, welche bei komplexen Abläufen den aktuellen Standort signalisiert. Die rechte Seite zeigt die Palette mit allen Komponenten, welche in Verzweigungen (Gateways), Iterator Elemente (Iterator Nodes), Kofigurationskomponenten (Configuration Nodes) und Flusskomponenten (Flow Nodes) gegliedert sind. Mittig ist die Modellierungsoberfläche und darunter werden, wenn vorhanden, die Eigenschaften eingeblendet.

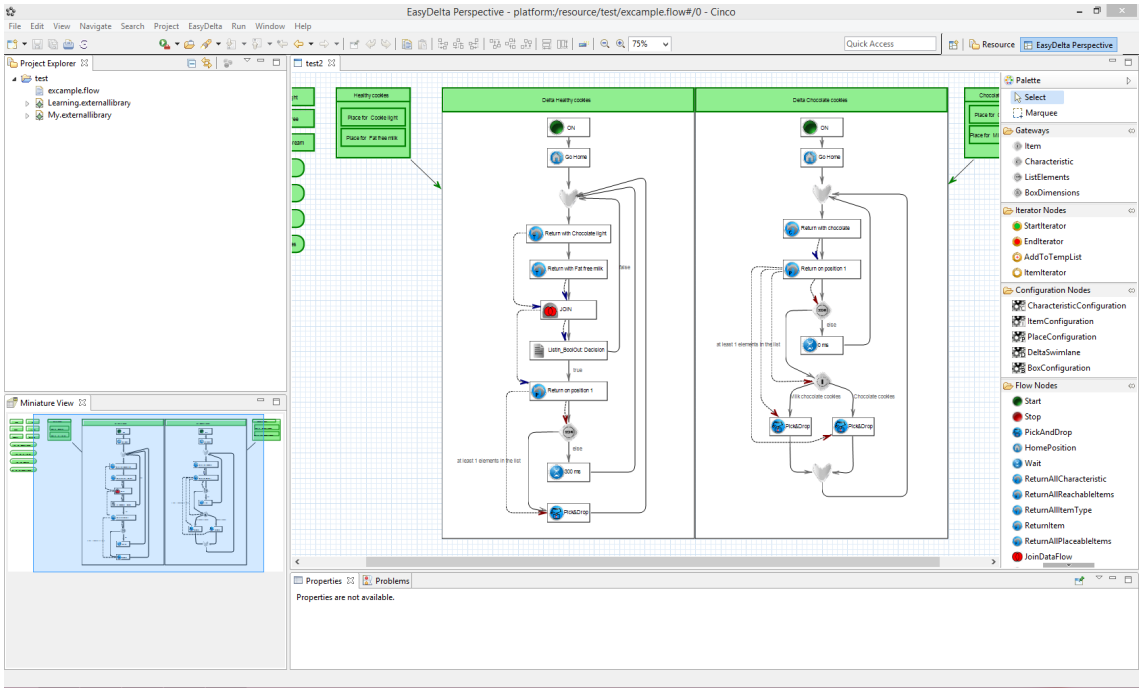


Abbildung 2.5: EasyDelta Editor.

Zunächst müssen alle Elemente der Konfiguration erstellt werden. Die Konfiguration für dieses Beispiel ist in Abbildung 2.6 abgebildet. Dort wurden zunächst sechs Charakteristika, vier Gegenstandstypen, zwei Boxen mit jeweils zwei Plätzen sowie zwei Delta-Roboter konfiguriert. Bevor der Ablauf modelliert wird, muss die Konfiguration vollständig abgeschlossen sein, da es sonst zu Fehlern führen kann.

Abbildung 2.6: Beispiel für eine Konfiguration eines Sortieralgorithmus.

Wurde dies erst einmal festgelegt, kann die Modellierung des Ablaufes beginnen. Das Ziel dieses Beispiels ist es eine Box mit gesunden, also fettfreien und zuckerreduzierten Keksen und eine Box mit allen Arten von Schokoladenkeksen zu verpacken.

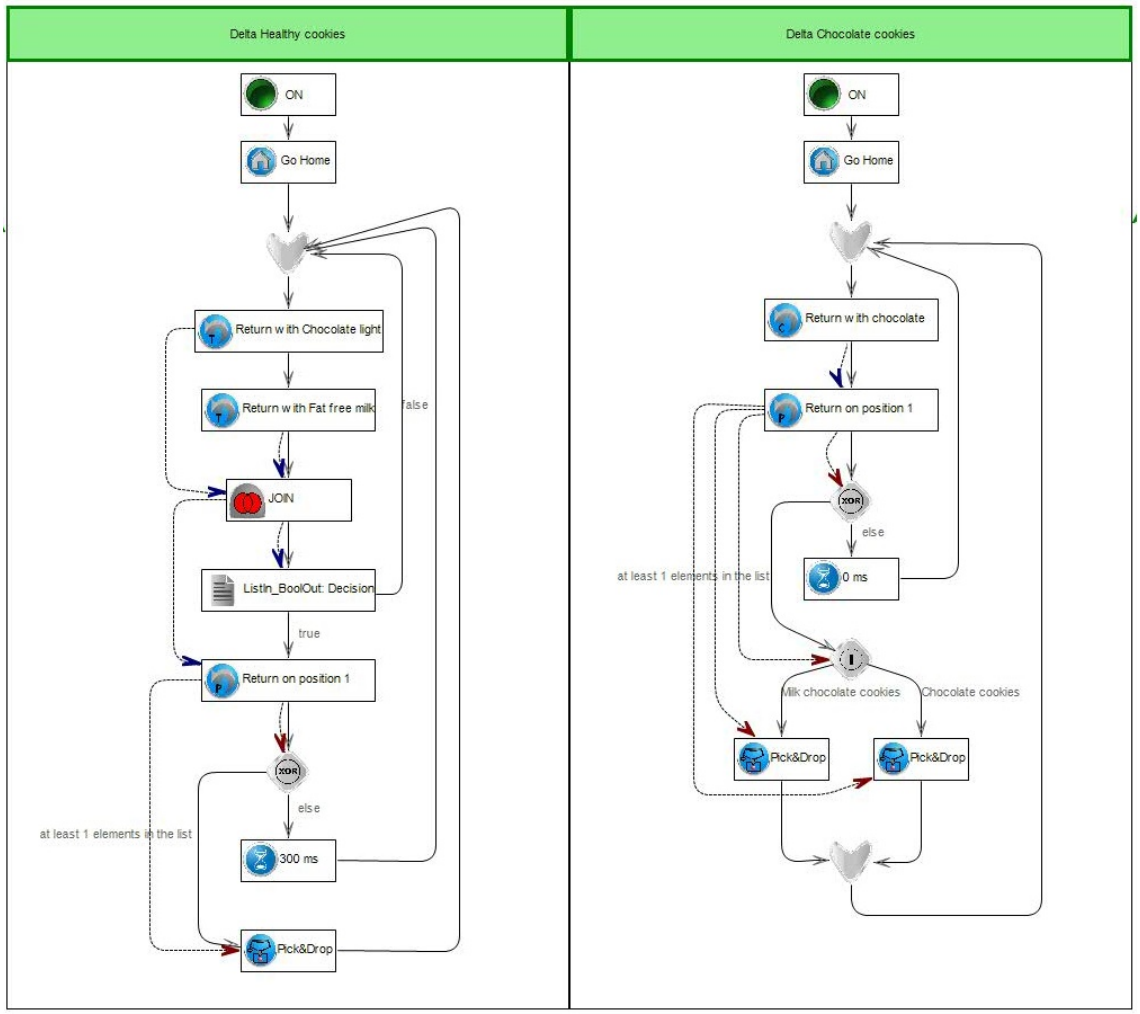


Abbildung 2.7: Beispiel für den Ablauf eines Sortieralgorithmus.

In Abbildung 2.7 ist der Ablauf dargestellt. Da ein Delta-Roboter immer nur eine Box packen kann, brauchen wir in diesem Beispiel zwei Delta-Roboter. Der erste Delta-Roboter übernimmt die gesunden Kekse und der zweite die Schokoladenkekse. Für die gesunden Kekse werden zunächst alle Gegenstände vom Gegenstandstyp *Chocolate light* und *Fatfree milk* in einer Liste zusammengeführt und daraufhin extern entschieden, ob diese Gegenstände verpackt werden sollen. Bei einer positiven Entscheidung und einer nicht leeren Liste wird das erste Element vom Delta-Roboter gegriffen und verpackt. Ist die Liste leer wird 300 Millisekunden gewartet und daraufhin eine neue Liste von Gegenständen angefordert. Bei einer negativen Entscheidung wird direkt nach einer neuen Liste gefragt.

Bei den Schokoladenkeksen wiederum werden alle Gegenstände mit der Charakteristik *chocolate* herausgefiltert und anschließend werden diejenigen mit Typ *Milk chocolate cookies* oder *chocolate cookies* in die Kiste gelegt. Bei einer leeren Liste wird ebenso 300 Millisekunden gewartet bevor eine neue Liste von Gegenständen abgefragt wird.

Dies ist ein simpler Sortieralgorithmus der nur den Grundgedanken des Editors widerspiegeln soll. Es können jedoch auch durchaus komplexere Algorithmen modelliert werden. Eine Besonderheit bei diesem Beispiel ist, dass es eine Endlosschleife ist. Es wird ständig nach neuen Gegenständen auf dem Fließband gewartet. Dies ist bei großen Fabrikanlagen wichtig, da so ein Leerlauf der Anlage verhindert wird. Somit endet der Algorithmus erst, wenn die Fabrikanlage ausgeschaltet wird.

In Abbildung 2.8 ist der vollständige Sortieralgorithmus mit Konfigurations- sowie Flusskomponenten zu sehen.

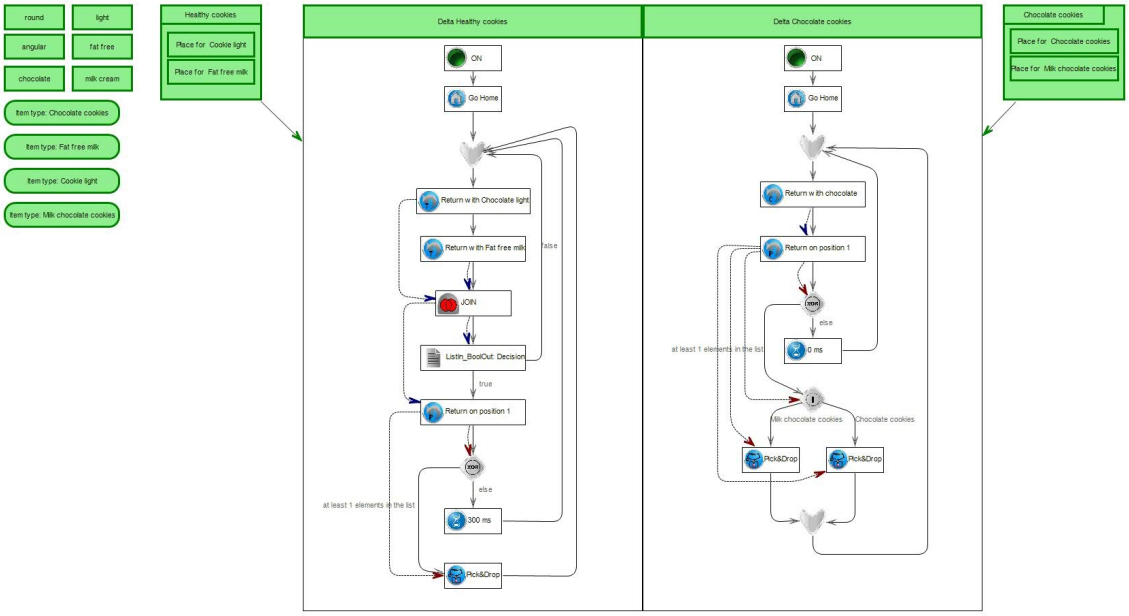


Abbildung 2.8: Beispiel eines Sortieralgorithmus.

Kapitel 3

Codegenerierung

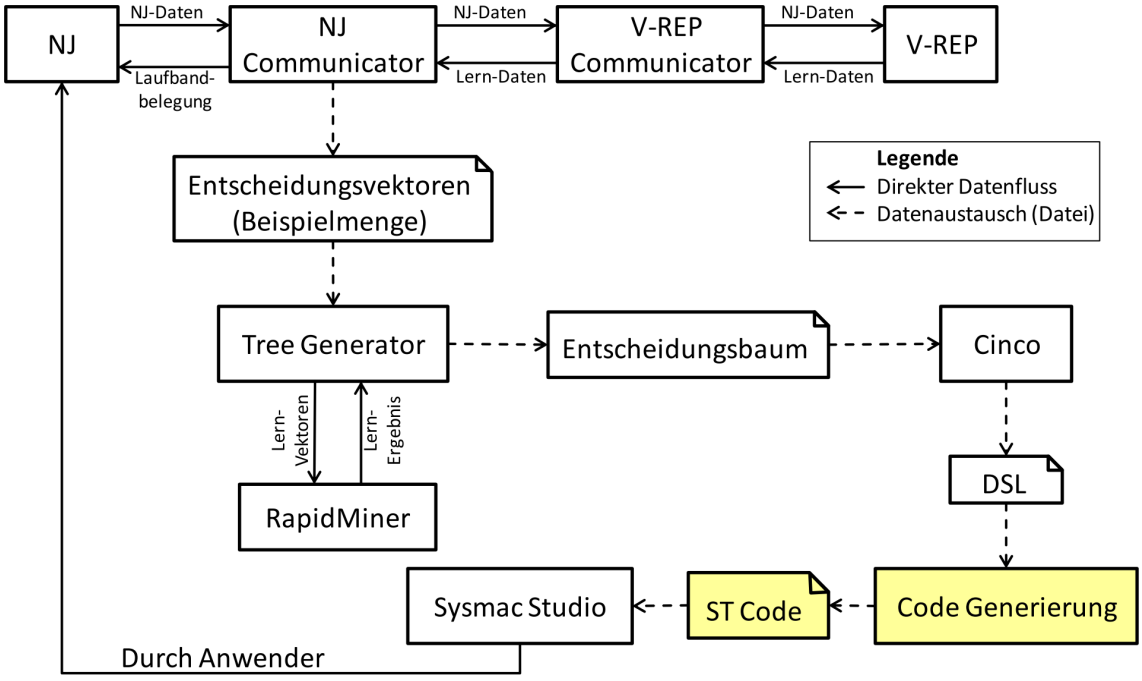


Abbildung 3.1: Einordnung der Codegenerierung in der Gesamtarchitektur.

Dieses Kapitel beschreibt den Codegenerator. Der Codegenerator ist ein Eclipse Plugin. Der Aufruf des Codegenerators erfolgt in Cinco. Die Zielsprache für den zu generierenden Code ist ST. Die Programmiersprache STt ist, neben dem Ladder Code, eine Möglichkeit Programme in Sysmac Studio(näheres siehe Kapitel 4) zu erstellen. Für ein besseres Verständnis werden zunächst Xtend und das Visitor-Pattern im Kapitel Grundlagen vorgestellt. Im Anschluss werden das Konzept und die Realisierung erläutert. Abschließend werden die bei der Realisierung aufgetretenen Herausforderungen beschrieben.

3.1 Grundlagen

3.1.1 Xtend

Xtend ist eine statisch getypte Programmiersprache für die Java Virtual Machine (siehe [13]). Im Vergleich zu Java bietet Xtend zusätzlich unter anderem Type Inferenz, Erweiterungsmethoden, Lambda Ausdrücke und eine Template-Engine, welche von uns für die Codegenerierung genutzt wird. Der in Xtend geschriebene Quellcode wird zu Java Source Code kompiliert. Daher gibt es keine Interoperabilitätsprobleme zwischen Java und Xtend. Auch der Aufruf von Xtend Methoden in Java ist möglich. Für die Codegenerierung wurde Java und Xtend verwendet. Da die Template-Engine das Hauptargument für die Verwendung von Xtend war, wird diese nachfolgend näher erläutert.

Template-Engines füllen Vorlagen in denen Platzhalter enthalten sind zur Laufzeit mit aktuellem Inhalt aus. In unserem Projekt ist ST der zu erstellende Quellcode. Die Platzhalter sind Parametrisierungen oder Werte für Variablen (siehe Listing 3.4). Die Vorlagen sind Methodenschablonen mit festen Anteilen und Lücken, die dynamisch gefüllt werden. Für die Lesbarkeit des generierten Quellcodes ist auf eine entsprechende Formatierung bzw. Einrückung zu achten. Die Template-Engine von Xtend übernimmt die Formatierung für den Zielcode aus der Vorlage. Templates werden in Xtend mit drei einfachen Anführungszeichen (''') gekennzeichnet und können sich über mehrere Zeilen erstrecken. Als Escape-Zeichen für die dynamischen Teile der Templates werden französische Anführungszeichen (Guillemets («»)) verwendet. Listing 3.1 zeigt den von Xtend übersetzten Java Quellcode, dieser befindet sich im Ordner xtend-gen. Ein Beispiel für generierten Quellcode zeigt Listing 3.2.

```
1  /**
2   * Set time in milliseconds
3   * bool_done ist true wenn der timer abgelaufen ist
4   */
5  public CharSequence setWait(final int ms, final int bool_done) {
6    StringConcatenation _builder = new StringConcatenation();
7    _builder.append("// sets the timer");
8    _builder.newLine();
9    _builder.append("TON(");
10   _builder.newLine();
11   _builder.append("\t");
12   _builder.append("In:=TRUE,");
13   _builder.newLine();
14   _builder.append("\t");
15   _builder.append("PT:=T#");
16   _builder.append(ms, " ");
17   _builder.append("ms,");
18   _builder.newLineIfNotEmpty();
19   _builder.append("\t");
20   _builder.append("Q=>UserBOOL");
21   _builder.append(this.id, "  ");
22   _builder.append("[");
```

```
23      _builder.append(bool_done, "  ");
24      _builder.append("]");
25      _builder.newLineIfNotEmpty();
26      _builder.append("); ");
27      _builder.newLine();
28      return _builder;
29    }
```

Listing 3.1: Der von Xtend generierte Java Quellcode

```
1 \\ sets the timer
2 TimerTON(
3   IN:=TRUE,
4   PT:=T#300ms
5   Q:=UserBOOL0[7]
6 );
```

Listing 3.2: Generierter Quellcode in ST

3.1.2 Visitor Pattern

Das Visitor-Pattern ermöglicht die Kapselung von Operationen, die auf Elementen einer Objektstruktur ausgeführt werden und das Hinzufügen neuer Operationen ohne eine Änderung an den Klassen der Elemente, auf denen diese Operationen ausgeführt werden. In Cinco kann dieses Entwurfsmuster mit Hilfe der Klassen TreeIterator und Switch implementiert werden. Die Klasse Switch wird in Cinco generiert. Sie bietet die Methode `doSwitch()` an, der ein Element der Objektstruktur übergeben werden kann. Entsprechend dem Typ des Elements wird eine eindeutig zugeordnete Methode `caseTypeXXX()` aufgerufen. Für jeden Typ existiert eine zugehörige Methode. Der Visitor erbt von der Klasse Switch und überschreibt die Methode `caseTypeXXX()` für den jeweiligen Typ. Es können verschiedene Besucher implementiert werden die unterschiedliche Funktionalitäten erfüllen. Mit dem TreeIterator kann über die Objektstruktur iteriert werden. Für jedes Element der Struktur wird auf dem Visitor die aus der Klasse Switch geerbte Methode `doSwitch()` aufgerufen und das aktuelle Element übergeben. Anhand des Types wird die eindeutig zugeordnete Methode `caseTypeXXX()` aufgerufen. Ihr wird das Element übergeben, damit die Informationen aus dem Element zur Verfügung stehen. Bei dieser Implementierung des Visitor-Patterns muss auf den Elementen der Objektstruktur nichts implementiert werden. Die Elemente müssen auch nicht wie eigentlich üblich eine Schnittstelle für den Empfang des Besuchers anbieten und im Anschluss auf den Visitor die Besuchsfunktion, in der sie sich selbst übergeben, aufrufen.

3.2 Konzept

Die von der grafischen DSL definierten Ablaufdiagramme können mit Hilfe des Editors modelliert werden. Anschließend sollen die so erstellten Diagramme in Quellcode für Sysmac Studio übersetzt werden. Der Quellcode für Sysmac Studio kann in Ladder oder ST geschrieben sein. Zur Übersetzung der modellierten Diagramme in kompilierbaren Quellcode für Sysmac Studio, sollte eine Codegenerierung erstellt werden.

Das in Cinco erstellte Diagramm ist ein Graph. Unser Grobentwurf sah daher eine Iteration über den Graphen vor. Bei dieser Iteration sollten die ST-Codeblöcke entsprechend der Knotentypen erstellt und als String konkateniert werden. Der komplette String sollte das fertig erstellte Programm in ST darstellen. Dieses Programm sollte Sysmac Studio übergeben werden und muss dort fehlerfrei kompilierbar sein.

3.3 Realisierung

Die Realisierung erfolgt in demselben Eclipse-Workspace, in dem die DSL definiert wurde. Das Projekt besteht aus drei Hauptkomponenten. Die drei Hauptkomponenten sind der Generator, die ST-Komponente und die Visitor-Komponente. Das Projekt wird als Eclipse-Plugin dem Cinco-Build Prozess übergeben. Alle drei Komponenten werden im Einzelnen vorgestellt.

3.3.1 Generator

Der Generator (Generator.xtend) basiert auf Xtend, einer Erweiterung für Java (s. 3.1.1). Für jeden Block in der DSL bietet der Generator eine Funktion an. Dazu wurde dieselbe Funktion mit einen Parameter für jeden DSL-Block überladen. Die Funktion gibt als Rückgabewert einen formatierten String aus. Innerhalb der Funktion werden zunächst die benötigten Variablen definiert. Dazu gehören immer die folgenden Blöcke, die an diesen angrenzen, die eingehenden Kanten und eventuell Variablen zum Zwischenspeichern von Items, die in diesem DSL-Block generiert werden. Zuletzt können auch diverse Hilfsvariablen definiert sein. In Listing 3.3 wird ein Beispiel für eine solche Funktion gezeigt, welches nachfolgend im Detail erklärt wird.

```
1 def String getCode(Wait node, int deltaNr){
2   /* Node: Wait
3    * Description: Waits for a specific time in ms
4    * incoming edges: ControlFlow
5    * outgoing edges: ControlFlow
6    */
7   val st = new StructuredText(deltaNr)
8   val nextNodes = getNextNodes(node)
```

```
9
10   val bool_done=uV.getUserBool(1) as int
11
12
13   '''
14   IF (state = <<visitedNodes.get(node)>>) THEN
15     // Wait
16     <<st.setWait(node.milliseconds,bool_done)>>
17     <<IF !nextNodes.empty>>
18     <<IF nodeVisited(nextNodes.get(0))>>
19       IF (UserBOOL<<deltaNr>>[<<bool_done>>] = TRUE) THEN
20         UserBOOL<<deltaNr>>[<<bool_done>>] := FALSE;
21         state := <<visitedNodes.get(nextNodes.get(0))>>;
22       END_IF;
23     END_IF;
24     <<ELSE>>
25       <<markVisited(nextNodes.get(0))>>
26       IF (UserBOOL<<deltaNr>>[<<bool_done>>] = TRUE) THEN
27         UserBOOL<<deltaNr>>[<<bool_done>>] := FALSE;
28         state := <<visitedNodes.get(nextNodes.get(0))>>;
29       END_IF;
30     END_IF;
31     <<ENDIF>>
32     <<ELSE>>
33       END_IF;
34     <<ENDIF>>
35     <<uV.releaseUserBool(bool_done)>>
36     '''
37 }
```

Listing 3.3: Beispiel für eine *Generator*-Funktion

Diese Generator-Funktion gibt den ST-Code für den DSL-Knoten „Wait“ aus. Der DSL-Knoten lässt den Kontrollfluss für eine definierte Zeit anhalten und somit auch den Delta-Roboter. Ihm übergeben wird der Knoten und die Nummer des Delta-Roboters. Diese Nummer wird über die Swimlane definiert, in der sich dieser Knoten befindet. Als erstes wird eine Instanz der ST-Komponente erzeugt (Z.7). Der Aufruf enthält die Nummer des Delta-Roboters, damit Delta-Roboter spezifische Anpassungen an dem ST-Code vorgenommen werden können. In Zeile 8 werden die nächsten Knoten über die Funktion `getNextNodes()` ermittelt. Die Funktion orientiert sich dabei an den ausgehenden Kontrollflusskanten, welche aus dem übergebenen Knoten ausgehen. Darauf folgt die Variable `bool_done`. Diese Variable wird im ST-Code genutzt um zu signalisieren, dass ein Knoten abgearbeitet wurde. Gerade bei diesem Beispiel wird diese Variable also gesetzt, wenn die voreingestellte Zeit des DSL-Knotens abgelaufen ist. Die Initialisierung der Variable wird über eine Hilfsklasse `UserVariableProvider` übernommen. Der nähere Sinn hinter dieser Hilfsklasse wird in Abschnitt 3.4.3 erläutert.

Nach der Initialisierung der notwendigen Variablen folgt das Template, welches mit ' ' ' beginnt. Jeder Knoten beginnt mit der Abfrage des Zustandes, in dem sich der Kontrollfluss befindet. Die dafür notwendige Nummer wird aus einer Hashmap gelesen (Z.14). Die Hashmap wird über die Funktion `markVisited()` im vorangehenden Knoten befüllt. Nach der Abfrage des Zustandes folgt der Aufruf der passenden Funktion der ST-Komponente. Diese Funktion liefert den formatierten ST-Code zurück. Darauf folgt der Code zur Weiterleitung des Kontrollflusses in den nächsten Zustand (Z.17 - 34). Informell wird dabei zunächst abgefragt, ob ein nächster Knoten existiert. Existiert dieser wird überprüft, ob der nächste Knoten bereits über die Visitor-Komponente besucht wurde oder nicht. Ist dies nicht der Fall, wird der nächste Knoten in der bereits angesprochenen Hash-Map zusammen mit eine Zustandsnummer hinterlegt. Der Kontrollfluss wird in Abhängigkeit zur `bool_done`-Variable zu dem nächsten Zustand weitergeleitet.

Zuletzt wird in Zeile 35 die nun nicht mehr gebrauchte Variable `bool_done` wieder freigegeben.

3.3.2 ST-Komponente

Die ST-Komponente (StructuredText.xtend) wurde erstellt um den Hauptteil des ST-Codes in einer Klasse zu bündeln. Auch werden so mehrfach verwendete Code-Blöcke zentral abgelegt und können einfach angepasst werden.

In der Klasse wird für jeden Code-Block eine Funktion angeboten. In Listing 3.4 wird eine solche Funktion angezeigt. Die Funktion erhält über die Parameter alle notwendigen Informationen, welche dynamisch in das Template eingebunden werden. Das Template ruft eine ST-Funktion auf. In diesem Fall wird die Funktion `setWait` aufgerufen. Diese besitzt die drei Parameter „In“, „PT“ und „Q“. In Zeile 9 wird über den Paramter „In“ die Funktion gestartet. Der Parameter PT bekommt die Information „T#“, gefolgt von der an `setWait` gesendeten Zeitspanne. Zuletzt wird der Ausgabewert „Q“ der Funktion an eine UserBOOL-Variable übergeben. Bei jedem Aufruf der Funktion wird in diese Variable der Wert „false“ geschrieben, wenn der Timer noch nicht abgelaufen ist. Sobald der Timer abgelaufen ist bekommt die UserBool-Variable den Wert „true“.

```
1  /*
2   * Set time in milliseconds
3   * bool_done ist true wenn der timer abgelaufen ist
4   */
5  def setWait(int ms, int bool_done) {
6    '''
7      // sets the timer
8      TON(
9        In:=TRUE,
10       PT:=T#«ms»ms,
11       Q=>UserBOOL«id»[«bool_done»]
12     );
```

```
13      '''
14    }
```

Listing 3.4: Beispiel für eine *ST-Komponente*-Funktion

Die BOOL-Variablen werden über Arrays realisiert. Über den Parameter „bool_done“ wird die Variable im Array genutzt, die zuvor noch als frei definiert wurde. Wie diese Variablen identifiziert werden, ist näher im Abschnitt 3.4.3 beschrieben.

3.3.3 Visitor-Komponente

Um den DSL-Graphen effektiv zu lesen, haben wir uns für das Visitor-Pattern entschieden. Die Grundlagen dazu werden im Abschnitt 3.1.2 erklärt. Die Visitor-Komponente erweitert die von Cinco bereitgestellte Klasse *EasyDeltaSwitch*. In der Klasse wird eine Funktion `doSwitch()` angeboten. Diese Funktion ruft eine der Implementierungen von `caseX()` auf. Die Visitor-Komponente (VisitorCodeGen.java) überschreibt die Funktionen `caseX()` für jeden DSL-Knoten und ruft die passende Funktion im Generator auf. In Listing 3.5 wird diese Funktion für den DSL-Knoten „Wait“ beispielhaft abgebildet.

```
1   @Override
2   public String caseWait(Wait object) {
3     logger.trace("caseWait(Wait object)...");
4     return gen.getCode(object, deltaNr);
5   }
```

Listing 3.5: Beispiel für eine *Visitor-Komponenten*-Funktion

In dieser Funktion wird der Generator in Zeile 4 mit dem DSL-Knoten und dem aktuellen Delta-Roboter aufgerufen. Zurückgegeben wird der generierte, formatierte Quelltext für diesen DSL-Knoten.

3.4 Herausforderungen

Dies beschreibt den grundlegenden Ablauf der Codegenerierung. Er garantiert, dass alle Knoten implementiert sind und der Baum der grafischen DSL komplett durchlaufen werden kann. Die Aufteilung mittels des Visitor-Patterns und das 1-1 Mapping ergaben allerdings auch Herausforderungen, die im Folgenden beschrieben werden sollen.

3.4.1 Datenfluss

Der Datenfluss bildet in der grafischen DSL den Fluss der Listen und Items zwischen den verarbeitenden Knoten ab. Die Knoten, welche Items oder Listen erzeugen, haben am Ausgang einen Datenfluss, welcher zu einem verarbeitenden Knoten führt. Die Herausforderung liegt nun darin im generierten ST-Code die Daten von der erzeugenden Funktion in die verbrauchende Funktion mit Hilfe der Datenflusskante zu überführen. Dazu muss der

Datenfluss nachverfolgt werden können, auch wenn er durch mehrere verarbeitende Knoten geht und dabei die Listen zwischenzeitlich verändert.

Der übliche Anwendungsfall besteht aus einem Knoten in der grafischen DSL, welcher eine Liste an Items ermittelt. Diese Liste wird dann auf verschiedene Charakteristika hin gefiltert. Zuletzt wird aus dieser gefilterten Liste ein Item ausgewählt. Listen und Items haben unterschiedliche Typen von Datenflusskanten.

Ein anderer Anwendungsfall baut die Listen wie folgt auf. Der Listenaufbau erfolgt über einen „Join"- bzw. „Intersect"-Knoten für Datenflüsse. Hier können beliebig viele Datenflüsse, die wiederum aus Listen erstellenden Knoten kommen, zusammengefasst bzw. geschnitten werden. Die Herausforderung besteht nun darin, dass die Methoden für diese vielen Knotentypen zusammenarbeiten müssen, ohne dass Sie unbedingt voneinander wissen. Außerdem dürfen die unterschiedlichen Listen nicht durcheinander gebracht werden.

Die Lösung bildet eine Regel in der DSL selbst. Sie verlangt, dass einem „pick&place"-Knoten immer eine Datenflusskante mit einem einzigen Element läuft. In die Knoten wiederum, welchen solch eine Datenflusskante ausgeht, führt immer eine Datenflusskante mit einer Liste. Verfolgt man diese Listen-Datenfluss-Kante, wird immer ein Listenverarbeitender bzw. -erzeugender Knoten erreicht.

Während des rekursiven Durchlaufs der DSL, werden die Listen erstellenden und verarbeitenden Knoten gesammelt und mit ihren DSL-Objekten eindeutig, zusammen mit ihrer Listen-ID, in einer Hash-Tabelle hinterlegt. Wird nun beim Aufruf des „pick&place"-Knotens das DSL-Objekt ermittelt, wie im vorangehenden Abschnitt beschrieben, kann über die Hash-Tabelle die passende Listen-ID gefunden werden.

Im Fall der „Join"- und „Intersect-Knoten" kann der Umgang wie folgt am Beispiel des „Join"-Knoten erklärt werden. Die „Join"-Operation wird dabei für jeweils zwei Listen, welche durch die Dateflüsse repräsentiert werden, durchgeführt. Die erste Liste ist immer die Liste, welche als Ergebnis diesem Knoten als weiterer Datenfluss ausgeht. Die zweite Liste ist nacheinander eine der eingehenden Listen. Wenn alle eingehenden Listen abgearbeitet sind, wird die erste Liste ausgegeben. Beim „Intersect"-Knoten verläuft dieses Verfahren analog, wobei zunächst eine „Join"-Operation auf die auszugebende, noch leere Liste und die erste eingehende Liste durchgeführt werden muss.

3.4.2 Schleifen

Ein typisches Programm für die Steuerung eines Delta-Roboters läuft in Endlosschleifen. Dementsprechend werden DSL-Programme auch Schleifen beinhalten. Diese Schleifen in ST-Code abzubilden, wurde über einen Automaten-ähnlichen Aufbau erreicht. In Listing 3.6 wird deutlich, dass einzelne DSL-Knoten wie Zustände betrachtet werden und mittels der Variablen `state` zwischen diesen Zuständen gewechselt wird. Obwohl also dass Pro-

gramm zyklisch durchlaufen wird, kann über die Automaten-ähnliche Strukturierung ein Flussgraph simuliert werden und somit auch die Schleife.

```
IF (state = 1) THEN
  // HomePosition
  // move the TCP in init position (4 := Delta0, 6 := Delta1)
  SystemBOOL[4]:=True;
  IF (SystemBOOL[5] = TRUE) THEN
    SystemBOOL[4] := FALSE;
    SystemBOOL[5] := FALSE;
    state := 2;
  END_IF;
END_IF;

[...]

IF (state = 7) THEN
  // PutIntoBox
  // trigger grabAndPut (0 := Delta0, 2 := Delta1)
  SystemUINT[0]:=UserItemSet2[0];
  SystemBOOL[0]:=TRUE;
  IF (SystemBOOL[1] = TRUE) THEN
    SystemBOOL[0] := FALSE;
    SystemBOOL[1] := FALSE;
    state := 1;
  END_IF;
END_IF;
```

Listing 3.6: Automaten-ähnlicher Aufbau des generierten ST-Codes

In diesem spezifischen Beispiel wird die Schleife über die beiden Zustände 1 und 7 gebildet.

Die technische Lösung bildet hier eine Hash-Tabelle, die sich die bereits besuchten Knoten und die vergebenen Zustände merkt. Vor jedem rekursivem Aufruf wird geprüft, ob der aufzurufende Knoten bereits besucht wurde. Falls ja, wird die Rekursion abgebrochen. Somit ist eine Endlos-Rekursion nicht möglich.

3.4.3 Begrenzter Variablenpool

ST arbeitet mit Variablen, die zuvor in einem speziellen Abschnitt des Programms definiert werden müssen. Sysmac bietet dafür eine Lösung über eine separate Tabelle an. Das Sysmac Studio verfügt zudem über ein proprietäres Projektformat. Über dieses Projektformat gibt es keine Dokumentation. Damit bleibt es dem Anwender verschlossen und dieser muss den Code manuell in ein vorhandenes Projekt einfügen. Die Variablendefinition muss manuell erfolgen und kann nicht direkt durch die Codegenerierung geschrieben werden.

Die Lösung ist ein Kompromiss. Der Anwender verfügt über einen statischen Pool von Benutzervariablen. Dadurch muss die Variablentabelle nicht bei jedem Bau angepasst werden. Die Herausforderung bestand nun darin, dass Variablen so effektiv wie möglich verwendet werden. Das bedeutet, dass Variablen, welche ab einer bestimmten Stelle im

Kontrollfluss verwendet werden, nicht mehr neu beschrieben werden bis der Ort im Kontrollfluss erreicht wurde, wo der Wert gebraucht wird. Eine simple Lösung würde alle Variablen verbrauchen und im gesamten ST-Code nicht mehr wiederverwenden.

Die implementierte Lösung bietet einen einfachen Algorithmus, der zwar nicht optimal arbeitet, aber auch nicht vollkommen naiv ist. Die Idee dahinter ist, dass Variablen während des Durchlaufs vergeben und gesperrt werden. Diese Sperre jedoch hält nicht über den gesamten Lauf des Programms an, sondern ist davon abhängig, wo die Variable wirklich verwendet wird. Das kann auch über Knotengrenzen hinweg geschehen, weswegen der Variablenpool als statische Klasse implementiert wurde. Wird eine Variable von einem bestimmten Typ (Bool, ItemSet, ...) gebraucht, kann diese über die statische Klasse angefordert werden. Über die gleiche Klasse kann dann eine beliebige Methode die Variable wieder freigeben, sobald sie gebraucht wurde.

In einem bestimmten Fall wird eine Variable auch über Knotengrenzen hinaus gebraucht. Das ist der Fall, wenn eine Gegenstandsliste erstellt wird. Diese intern als „itemset“ bezeichnete Liste wird in einem Knoten generiert und in eventuell vielen anderen Knoten benötigt. Diese entfernten Knoten werden während der Codegenerierung nacheinander abgearbeitet. Würde eine Variable zur Neubeschreibung freigeben werden, sobald sie das erste Mal gebraucht wurde, werden andere von der Variable abhängigen Knoten diese nicht mehr finden. Oder der Wert der Liste würde inzwischen überschrieben werden und die Knoten bekämen einen veralteten Wert geliefert. Deshalb hängt bei jeder internen Variabel auch eine Zahl an, die angibt, in wie viele Datenflüsse die Gegenstandsliste versendet wurde. Die nun normale Freigabe dekrementiert zunächst nur diesen Wert, jedes Mal, wenn die Gegenstandsliste benötigt wurde. Erst wenn, alle Datenflüsse abgearbeitet wurden, wird die Variable wieder freigegeben. Dadurch wird das Problem behoben.

Ein bewiesenes Konzept wäre die Verwendung einer Datenflussanalyse um im vornherein die Mindestanzahl an notwendigen Variablen zu ermitteln. Ein möglicher Ansatz wäre die Analyse nach „Live-Variables“ gewesen. Diese Analyse untersucht die Variablen und deren Inhalte. Wird eine Variable nach ihrer Definition an einer Stelle im Kontrollfluss verwendet, ist sie lebendig, also „[a]live“. Wird sie hingegen neu definiert, dann verliert sie diesen Status. Alle „toten“ Variablen können vernachlässigt werden, alle lebendigen Variablen sind bis zu dem Zeitpunkt, an dem sie „tot“ sind, notwendig. Über den naiven Ansatz aus dem zweiten Abschnitt und einer nachfolgenden „Live-Variables-Analysis“ [14] hätte die Anzahl an notwendigen Variablen bewiesenermaßen bestimmt werden können.

3.4.4 Mehrere Delta-Roboter

In unserem Testaufbau sind zwei Delta-Roboter vorhanden. Der rekursive Aufbau sieht jedoch vor, dass ein eindeutiger Startpunkt vorhanden ist. Dieser eindeutige Startpunkt geht bei zwei Delta-Roboter jedoch verloren.

Durch die recht generische Struktur des Algorithmus ist es einfach auch Code für einen zweiten Delta-Roboter zu erzeugen. Die Konvention sieht vor, dass ein Kontrollfluss pro Delta-Roboter immer mit einem „On“-Knoten beginnt. Über wenig Hilfscode lässt sich der Algorithmus auch für den zweiten „On“-Knoten instrumentalisieren. Das funktioniert erstaunlich gut. Ein Problem ergab sich noch durch den Variablenpool. Dies ließ sich jedoch ebenfalls über eine zusätzliche ID für jeden Delta-Roboter und einen zweiten Satz an Nutzervariablen lösen.

3.4.5 Konfiguration

Eine große Herausforderung stellte die Generierung der Konfiguration dar. Die Konfiguration beschreibt innerhalb der DSL den Namen der Delta-Roboter, die Boxen, die Plätze innerhalb der Boxen und die Arten von Gegenständen.

Die Konfiguration wird überall während der Codegenerierung wiederverwendet. Wenn zum Beispiel ein Return-Knoten wissen muss welchen ItemTyp er suchen soll, dann benötigt das ST-Pendant den dazugehörigen eindeutigen Bezeichner bzw. die ID.

Weil die Konfiguration so wichtig ist und eventuell Informationen bereithalten muss, welche nicht in dem Modell der DSL stehen, haben wir uns entschlossen die Konfiguration zunächst einzulesen und als zusätzliches Datenmodell in Java abzubilden. Ein Vorteil ist der höhere Freiheitsgrad in der Vergabe von speziellen Funktionen und Daten. Während des Einlesens der Konfiguration aus dem Modell wird darauf geachtet, dass die Reihenfolge der Datenabhängigkeiten eingehalten wird. So benötigt eine Box zunächst alle Places. Ein Place hingegen benötigt wieder ein ItemTyp, damit sicher ist, welcher ItemTyp zu welchem Place gehört. Zusätzlich war es wichtig, dass Informationen auch fehlen dürfen, ohne dass die Codegenerierung sofort abstürzt.

Zuletzt werden die gesammelten Informationen verwendet um die für ST notwendigen Informationen zu liefern, die später auch auf die Darstellung in der Visualisierungsumgebung auswirken. Dazu wurden für jeden Konfigurationsbaustein ST-Methoden implementiert. Diese ST-Methoden werden einmal oder mehrmals, abhängig vom Modell, aufgerufen. Der daraus entstandene ST-Code wird in einer separaten Datei hinterlegt.

Kapitel 4

Sysmac Studio

Sysmac Studio [9] ist die Automatisierungsplattform der Firma OMRON [10]. Sie ermöglicht die Steuerung einer gesamten Maschine oder Produktionsstraße. Die Steuerung erfolgt über den Machine-Controller NJ, welcher die Ablaufsteuerung, die Motion Controller und Visions kontrolliert und alle Elemente durchgängig vernetzt. Die Plattform ermöglicht die einheitliche Konfiguration und Programmierung der Elemente.

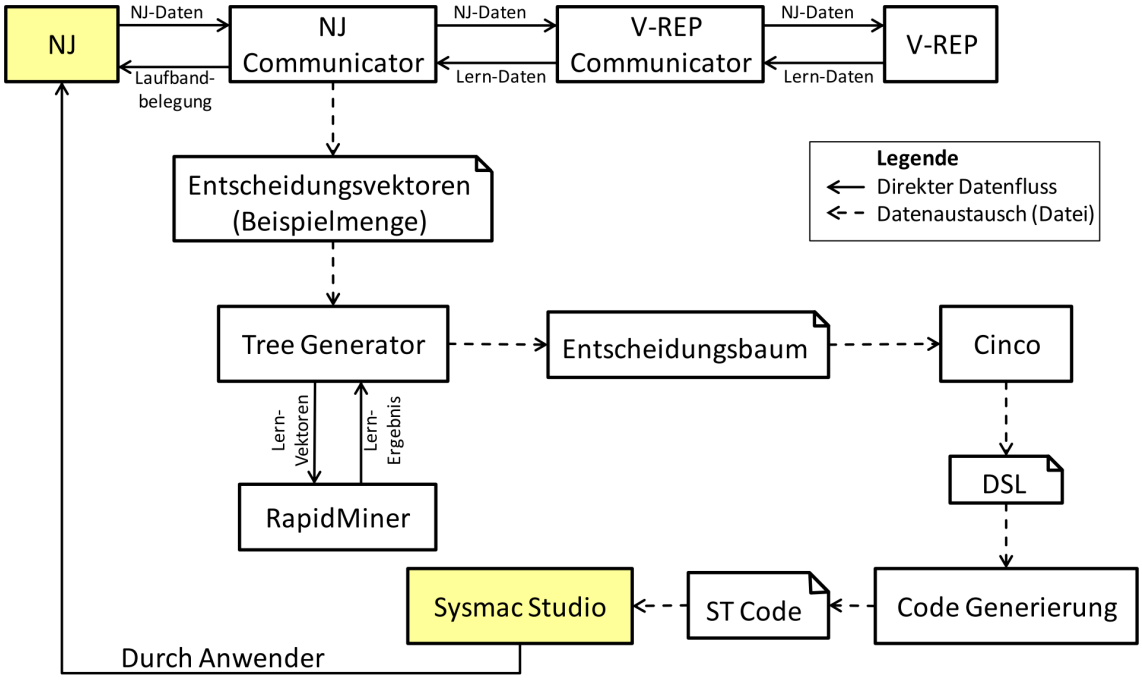


Abbildung 4.1: Einordnung des Sysmac Studio und der NJ in der Gesamtarchitektur.

Dieses Kapitel beschreibt die Entwicklung eines Rahmenprogramms in Sysmac Studio, in welches das in der Codegenerierung entworfene Programm ohne weitere Anpassungen eingefügt werden kann.

Wie in Abbildung 4.1 zu sehen ist, stellt Sysmac Studio das Bindeglied zwischen der Codegenerierung und der NJ dar. Sysmac Studio soll den generierten Code auf die NJ überspielen, sodass er ausgeführt und visualisiert werden kann.

Zunächst werden die Grundlagen innerhalb der Programmierung und Konfiguration von Komponenten mit Hilfe von Sysmac Studio erläutert. Mit den gewonnen Kenntnissen wird im zweiten Abschnitt dieses Kapitels ein Konzept erstellt anhand dessen das angestrebte Ziel erreicht werden soll.

4.1 Grundlagen

In diesem Abschnitt werden die Grundlagen für die Entwicklung des Rahmenprogramms in Sysmac Studio geschaffen. Zum besseren Verständnis werden zunächst Begriffe vorgestellt und erläutert, welche für dieses Kapitel von Bedeutung sind.

Die Entwicklungsumgebung teilt sich in zwei Teile: Konfiguration und Programmierung. Diese Module stellen die beiden unterschiedlichen Schnittstellen der Entwicklung dar und repräsentieren die Hard- und Softwareseite des Machine-Controllers.

4.1.1 Wichtige Begrifflichkeiten

Die folgenden Begriffe beziehen sich auf das in diesem Bericht beschriebene Szenario und die damit einhergehenden Einschränkungen.

Achsengruppe: Eine Achsengruppe repräsentiert eine Maschine in der mehrere Motion Controller abhängig voneinander Arbeit verrichten. In dem hier gegebene Kontext wird ein Delta-Roboter durch eine Achsengruppe beschrieben, welche aus vier Motion Controllern besteht. Jeweils ein Motion Controller für einen der drei Arme und einen Weiteren, welcher die Rotation des Werkzeugs ermöglicht.

Funktionsblock: Ein Funktionsblock entspricht einem aus einer objektorientierten Programmierhochsprache bekannten Objekt mit lediglich einer Methode. Ein solcher Block enthält gekapselte sequenziell ablaufende Anweisungen. Ein Aufruf erfolgt mit zuvor festgelegten Ein- und Rückgabedaten. Die Verarbeitung erfolgt zustandsbehaftet.

Funktion: Eine Funktion ist im Gegensatz zu einem Funktionsblock zustandslos, sodass die Verarbeitung ausschließlich auf den angegebenen Parametern erfolgt.

Laddercode: Äquivalente Programmiersprache zum strukturierten Text. Sie basiert auf dem Standard IEC 61131-3 und ist eine grafische Modellierungssprache. [3]

Motion Controller: Ein Motion Controller ist eine Schnittstelle zur Steuerungseinheit eines Servomotors, welcher in diesem Kontext durch eine virtuelle Achse repräsentiert wird.

ST: Der strukturierte Text (engl. Structured Text, Abkürzung: ST) wird für die Programmierung von Speicherprogrammierbaren Steuerungen (SPS) eingesetzt. Die Norm EN 61131-3 legt neben anderen auch den Sprachumfang von ST fest. Dabei ist die Syntax der Sprachelemente ähnlich denen der Hochsprache Pascal. [3]

TCP: Der TCP (Tool Center Point) beschreibt die durch die drei Arme des Delta-Roboters bewegte Werkzeug.

Virtuelle Achse: Eine virtuelle Achse ist der Wertebereich der von dem Servomotor dieser Achse durch eine rotatorische oder translatorische Bewegung zurückgelegten Strecke. Die virtuelle Achse arbeitet ausschließlich mit theoretischen und idealisierten Werten und unterliegt keinem realen Motor.

4.1.2 Konfiguration

Die Konfiguration stellt eine der zwei von Sysmac Studio angebotenen Schnittstellen zum Machine-Controller dar. In der Konfiguration wird das gegebene Szenario aus den Hardware-Komponenten modelliert (siehe Abbildung 4.2). Diese Komponenten umfassen Servo-Motoren, Spannungswandler, Benutzerkonsolen, Schalter, Visions und weiter Sensoren und Aktoren, aus denen einzelne Maschinen oder ganze Fertigungsstraßen zusammengesetzt werden können.

Diese Komponenten können mittels EtherCat miteinander in Reihe geschaltet werden, sodass die Kommunikation zwischen den Komponenten ermöglicht wird. Die Steuerung der Komponenten erfolgt mit Hilfe der NJ, welche ebenfalls an das EtherCat angeschlossen wird. Die NJ überwacht die Ist-Werte der Komponenten und setzt im Millisekundenbereich die Soll-Werte. Alle Informationen werden von der NJ gesammelt, verarbeitet und dienen als Grundlage zur Entscheidung des weiteren Vorgehens.

Um das diesem Bericht zur Grunde liegende Szenario zu modellieren, werden ausschließlich Servomotoren in Form von Motion Controllern benötigt. Das gegebene Szenario wird ausschließlich simuliert, sodass keine realen Servomotoren vorhanden sein müssen und an die NJ via EtherCat angeschlossen werden. Für diesen Zweck bietet Sysmac Studio die Möglichkeit, virtuelle Achsen zu definieren, welche ein idealisiertes Verhalten eines Servo-

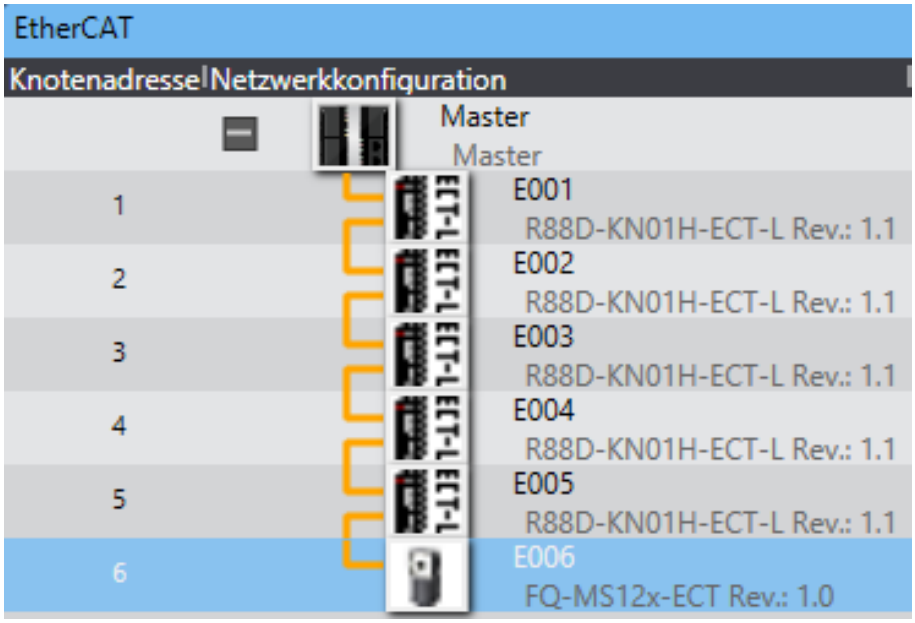


Abbildung 4.2: Beispielsweiser Aufbau einer Konfiguration von Komponenten in Sysmac Studio.

Motors simulieren können. Um eine in sich abgeschlossene Funktionseinheit (Maschine), welche aus mehreren Servomotoren zusammengesetzt ist, wie der Delta-Roboter, zu modellieren werden Achsengruppen definiert. Servomotoren deren Achsen in einer Achsengruppe zusammengestellt sind, werden ausschließlich als eine Einheit gesteuert. OMRON bietet verschiedene Steuerungen, welche eine bestimmte Achsengruppen auf eine vordefinierte Maschine abbilden. Diese Steuerungen bieten dem Entwickler komfortable Möglichkeiten die Maschine in ihrer Gesamtheit zu steuern, ohne die Notwendigkeit jeden Motion Controller einzeln zu bedienen. Achsengruppen können ebenfalls aus virtuellen Achsen zusammengestellt werden und simulieren eine gesamte Maschine. Die verwendeten Delta-Roboter werden ebenfalls durch eine Achsengruppe in Sysmac Studio dargestellt und gesteuert.

4.1.3 Programmierung

Die Programmierung der NJ erfolgt wie die Konfiguration ebenfalls mit Hilfe des Sysmac Studio in Programmen. Jedes Programm muss einem Task zugeordnet werden, welcher mit einer Wiederholungsperiode ausgeführt wird. Es werden zwei verschiedene Programmiersprachen angeboten, in denen die Programme verfasst werden können. Die textuelle Programmiersprache ST und die grafische Modellierungssprache Ladder Code. Beide Sprachen sind in ihren Möglichkeiten identisch und bieten nur visuelle Unterschiede. Unabhängig von der gewählten Sprache werden die Anweisungen in einzelnen Programmen definiert, welche iterativ und sequenziell ausgeführt werden. Beide Programmiersprachen bieten die Möglichkeit Kontrollstrukturen wie Entscheidungen, Zuweisungen, Arithmetik und Schleifen zu nutzen. Zudem können zu Beginn eines Programms typisierte Variablen definiert werden.

Diese Variablen werden ausschließlich mit einem Editor deklariert und initialisiert und können nicht dynamisch innerhalb eines Programms erstellt werden. Um mehrere verschiedene Datentypen zu bündeln kann der Entwickler eigene Datentypen, sog. Structs, definieren, welche sich aus weiteren Structs oder primitiven Daten zusammensetzen können.

Für die programmübergreifende Nutzung von Variablen bietet Sysmac Sudio die Möglichkeit globale Variablen zu definieren, welche im Anschluss als Externe Werte in ein Programm geladen werden können. Als Äquivalent zu bekannten Methoden einer objektorientierten Hochsprache kann der Entwickler wiederkehrende Anwendungsfolgen in sog. Funktionen (Fs) und Funktionsblöcken (FBs) kapseln und auslagern. Die Ein- und Ausgabe-Variablen werden als Parameter zur Initialisierung eines FBs oder Fs übergeben und können im Anschluss genutzt werden. Sysmac Studio bietet bereits ein große Anzahl an verschiedenen Fs, wie mathematischen Funktionen oder Zeitmessern und FBs, welche zur Steuerung von Achsen und Achsengruppen verwendet werden können. Mit Hilfe dieser von Sysmac Studio bereit gestellten Mitteln, kann die NJ programmiert werden, sodass alle angeschlossenen Aktoren und Sensoren angesteuert werden.

4.1.4 Problemstellung im Kontext

Sysmac Studio bildet das Bindeglied zwischen der Codegenerierung (siehe Kapitel 3), welche den in der DSL (siehe Kapitel 2) festgelegten Entscheidungsautomaten in ST generiert und der Visualisierung in V-Rep (siehe Kapitel 5.1), welche mit Hilfe der NJ-Com (siehe Kapitel 5.3.1) die Ist-Werte aus der NJ liest und simuliert (siehe Abbildung 4.1). Daraus resultieren verschiedene Problemstellungen:

1. Es muss eine Möglichkeit geschaffen werden, dass generierter ST-Code in Sysmac Studio eingefügt wird, ohne das der Entwickler weitere Schritte in Sysmac Studio unternehmen muss. Dies umfasst vor allem das Problem der nicht dynamisch zu deklarierenden Variablen innerhalb des Codes.

2. Das gesamte Szenario muss innerhalb der NJ simuliert werden, um alle nötigen Werte für die Visualisierung durch V-Rep bereit zu stellen. Das heißt, dass nicht nur die Maschinen und deren Servomotoren, sondern auch alle Gegenstände, Kisten und das Fließband abgebildet werden müssen. Ohne die Notwendigkeit der Visualisierung durch V-Rep, würden die Gegenstände durch eine Kamera erfasst und müssten nicht simuliert werden.

3. Um eine schnelle und einfache Anpassung der initialen Werte der Visualisierung zu ermöglichen, muss eine Möglichkeit bereit gestellt werden, die Daten innerhalb der Laufzeit in der NJ zu bearbeiten.

4. Jeder Delta-Roboter soll unabhängig von den anderen gesteuert werden können, sodass die entsprechenden Programme der einzelnen Roboter parallel ablaufen müssen.

Zur Bewältigung der hier aufgeführten Problemstellungen werden im nächsten Abschnitt dieses Kapitels Konzepte erarbeitet, welche anschließend realisiert werden.

4.2 Der Arbeitsbereich

In dieser Abschnitt wird gezeigt, wie die Arbeitsumgebung der Delta-Roboter dimensioniert ist. Dafür wird einerseits der Arbeitsbereich von einem Delta-Roboter beschrieben und anderseits wird ein allgemeines Koordinatensystem festgelegt und gezeigt, wie die Komponenten, beispielsweise Delta-Roboter, das Fließband und der Arbeitsbereich dargestellt sind.

Ein Arbeitsbereich (siehe Abb. 4.3-A) ist der Raum, in dem ein Delta-Roboter einen Gegenstand greifen kann. Jeder Delta-Roboter hat seinen eigenen Arbeitsbereich. Er ist in Sysmac Studio durch den Funktionsblock MC_Setkintransform zu konfigurieren und die Parametereingaben sind vom Hersteller des Delta-Roboter vorgegeben. Allerdings wird ein Fehler von Sysmac generiert, wenn ein Delta-Roboter aus dem Arbeitsbereich herausfährt. Zur Gewährleistung dieser Safetyeigenschaften, d.h ein unerwartetes Ereignis darf nicht geschehen, muss ein sicherer Arbeitsbereich definiert werden. Dies entspricht dem roten Rahmen in der Abbildung 4.3-B.

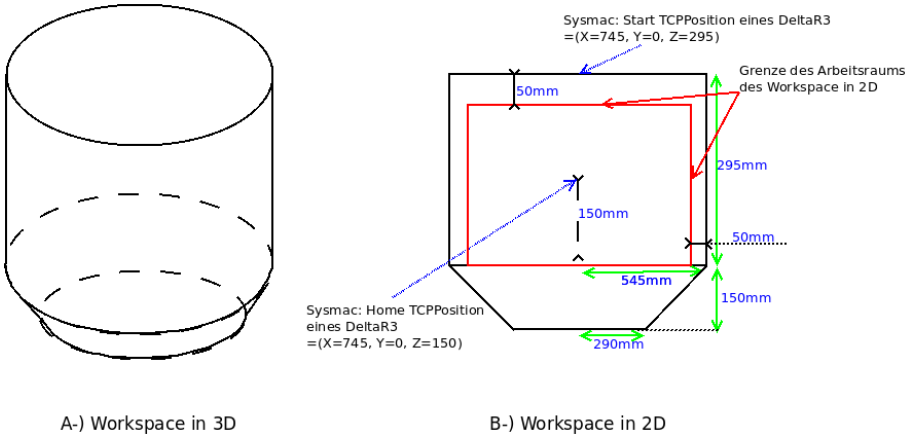


Abbildung 4.3: Arbeitsbereich eines Delta-Roboter.

4.2.1 Die Koordinatensysteme

In Sysmac Studio werden derzeit zwei Delta-Roboter betrachtet. Jeder Delta-Roboter hat seinen eigenen Arbeitsbereich und sein eigenes Koordinatensystem; das wird als Machine Coordinate Systeme (MCS) bezeichnet. Um bei beiden Delta-Roboter mit einem einzigen globalem Koordinatensystem zu arbeiten, bietet Sysmac Studio die sogenannten User Coordinate Systeme (UCS). Dadurch werden alle anderen Komponenten z.B das Fließband, der Arbeitsbereich und die Gegenstände über dieses globale Koordinatensystem referenziert. In der Abbildung 4.4 wird das Koordinatensystem dargestellt.

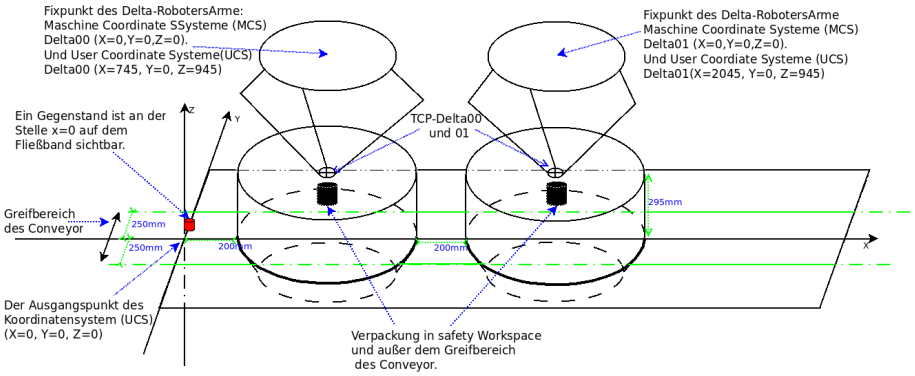


Abbildung 4.4: Das globale UCS.

Im globalen Koordinatensystem befindet sich das Fließband in der XY-Ebene ($Z = 0$, $0 \leq Y \leq 3000$, $-250 \leq Y \leq 250$), sodass seine zentrale Achse und die Y-Achse kongruent sind. Außerdem wurde festgelegt, dass das Fließband $500mm$ Breit und $3000mm$ lang sein muss. Die Ebene des Fließbands definiert den Bereich, indem sich die Gegenstände fortbewegen.

Sobald die Gegenstände, welche mit einem negativen X-Wert initialisiert werden, das Fließband betreten, werden sie im Simulator V-REP sichtbar. Die Kantenlängen bei einem quadratischen Gegenstand und Radius bei einem Zylinder, werden auf $40mm$ festgelegt. Um einen Gegenstand auf dem Fließband simulieren zu können, wird die aktuelle Position des Gegenstands in der NJ abhängig von der Geschwindigkeit des Fließbands berechnet und getaktet abgefragt.

Da jeder Delta-Roboter seinen Arbeitsbereich hat, wird im globalen Koordinatensystem festgelegt, dass sich die Arbeitsbereiche nicht überschneiden dürfen. Deswegen wurde ein Abstand von $200mm$ auf der X-Achse zwischen den Arbeitsbereiche festgelegt. Außerdem wird angenommen, dass sich die sensorischen Kameras in der YZ-Ebene an der Stelle $X = 0$ befinden. Aus diesem Grund wurde an dieser Stelle ebenfalls ein Abstand von $200mm$ auf Sicherheitsgründe zwischen dieser Ebene und dem ersten Arbeitsbereiche festgelegt. Die Mittelpunkte der Arbeitsbereiche befinden sich im globalen Koordinatensystem an den Punkten ($X = 745$, $Y = 0$, $Z = 0$) für den ersten Arbeitsbereich und ($X = 2045$, $Y = 0$, $Z = 0$) für den zweiten Arbeitsbereich.

Zum Einbau eines Delta-Roboter müssen die zwei Punkte definiert werden. Zunächst muss der Montagepunkt eines Delta-Roboters festgelegt werden, welcher sich mittig über dem festgelegten Arbeitsbereich befinden muss. Die Montagehöhe wird auf $945mm$ über dem Fließband festgelegt, wodurch sich der gesamte zylindrische Teil des Arbeitsbereichs oberhalb des Fließbandes befindet. Daraus resultieren die Montage-Punkte ($X = 745$, $Y = 0$, $Z = 945$) für den ersten und ($X = 2045$, $Y = 0$, $Z = 945$) für den zweiten Delta-Roboter.

4.3 Konzept und Realisierung

Dieser Abschnitt befasst sich mit dem Entwurf eines Konzepts zur Bewältigung der zuvor beschriebenen Problemstellung. Insbesondere konzentriert sich das Konzept auf die statische Datenstruktur, welche in einem dynamischen Kontext genutzt werden soll, die Abbildung von Funktionalitäten auf Funktionsblöcke zur Reduktion des zu generierenden Codes und auf die Nutzung von Programmen und Tasks zur parallelen Bearbeitung.

4.3.1 Datenstruktur

Die Datenstruktur dient als Abbildung des gesamten Szenarios und setzte sich aus den verschiedenen vorhandenen sichtbaren Objekte und Maschinen zusammen.

Abbildung der Objekte

Für die Objekte wurde ein Struct-Relational-Mapping (SRM-Datenbank) entwickelt und angelegt. Dieser Datenbank-Typ dient der Beseitigung von Redundanzen und Einsparung von Speicherplatz. Die SRM-Datenbank hält jede Struct-Instanz in einer für den jeweiligen Typen vorgesehen Liste. Die Relationen zwischen den einzelnen Instanzen werden mittels Join-Operatoren und den IDs der Instanzen realisiert. Für jeden Objekt-Typ wird ein Struct bereitgestellt:

- **Item**: Die Gegenstände, welche über das Fließband transportiert, von einem Delta-Roboter gegriffen und in einer Packung verpackt werden. Jeder Gegenstand wird durch einen Vektor beschrieben, welcher die Koordinaten des Greifpunkts in Millimetern, ausgehend vom Koordinatenursprung für einen Delta-Roboter darstellt. Neben diesen Koordinaten wird noch eine Rotation in Grad angegeben, welche eine Drehung des Gegenstands beschreibt. Um die Gegenstände zu charakterisieren, werden mehrere Eigenschaften (Characteristic-Sets) für einen Gegenstand festgelegt. Wenn ein Gegenstand verpackt wurde, wird festgehalten auf welchem Platz der Gegenstand platziert wurde.

- **Package**: Die Packungen, welche neben dem Fließband platziert werden können bestehen aus mehreren Plätzen, in denen die Gegenstände platziert werden können. Jede Packung wird einem Delta-Roboter zugeordnet und besitzt einen Vektor als Referenz-Position von der aus die einzelnen Platz-Positionen berechnet werden. Dies kann in einer Weiterentwicklung verändert werden, sodass mehrere Delta-Roboter die selben Packungen oder mehrere befüllen.

- **Place**: Ein Platz stellt einen Teil einer Packung dar. Die Position, welche den Absetzpunkt des Gegenstands darstellt, wird durch einen Vektor spezifiziert. Außerdem

wird eine Drehung in Grad angegeben, in welche der Gegenstand gedreht werden muss, bevor er platziert wird.

- **Charactersitic Set**: Jedem Gegenstand werden mehrere Eigenschaften zugeordnet, welche in einer Eigenschaftsliste gehalten werden.
- **Characteristic**: Eine Eigenschaft wird durch eine beliebige Zeichenkette festgelegt. Der Entwickler ist auf diesem Weg flexibel in der Definition der Eigenschaften von Gegenständen.

Die verschieden Struct-Instanzen werden in Listen verwaltet. Für jeden der vorgestellten Typen wird eine Liste in den globalen Variablen instantiiert, wodurch die Listen in jedem Programm und jedem Funktionsblock als externe Variable verfügbar sind. Die Datenstruktur der verschiedenen Objekte wird in Abbildung 4.5 verdeutlicht.

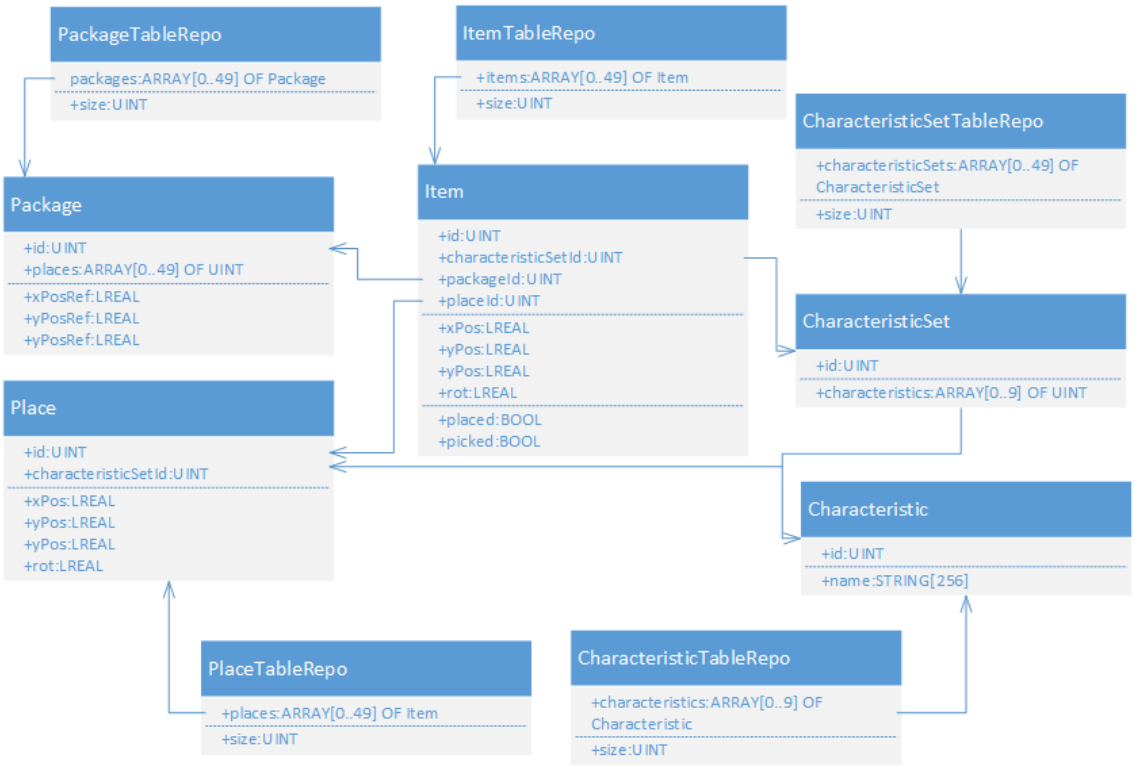


Abbildung 4.5: Datenstruktur zur Verwaltung der Objekten mittels SRM.

Abbildung der Maschinen

Die Abbildung der Maschinen befasst sich mit der Strukturierung und Kapselung der einzelnen Maschinen in dafür angepassten Datenstrukturen. Es werden Structs definiert, sodass alle relevanten Informationen und Einstellungen für die verschiedenen Maschinen zentral gehalten werden.

Die Datenstruktur besteht aus in sich verschachtelten Structs und nicht dem SRM, um unnötige Join-Operationen zu vermeiden, da die Anzahl der Maschinen nicht während der Laufzeit geändert werden muss. Die verschiedenen Maschinen des Szenarios werden wie folgt abgebildet:

- **Delta Robot**: Für jeden Delta-Roboter wird eine Struct-Instanz erzeugt, in welcher die einzelnen Achsen, die Achsgruppe und alle nötigen Einstellungen festgehalten werden. Zu den Einstellungen gehören insbesondere die Montagepunkte, Ausgangsposition und Beschleunigungswerte.
- **Conveyor**: Das Fließband wird durch einen Motion Controller gesteuert, dessen virtuelle Achse in dem Struct gehalten wird. Außerdem werden die Einstellungen der Soll-Geschwindigkeit, der Beschleunigung und der Länge verwaltet.

Die Verschachtlung der Structs und eine detaillierte Auflistungen aller Einstellungen, welche zur Verfügung gestellt werden, können der Abbildung 4.6 entnommen werden.

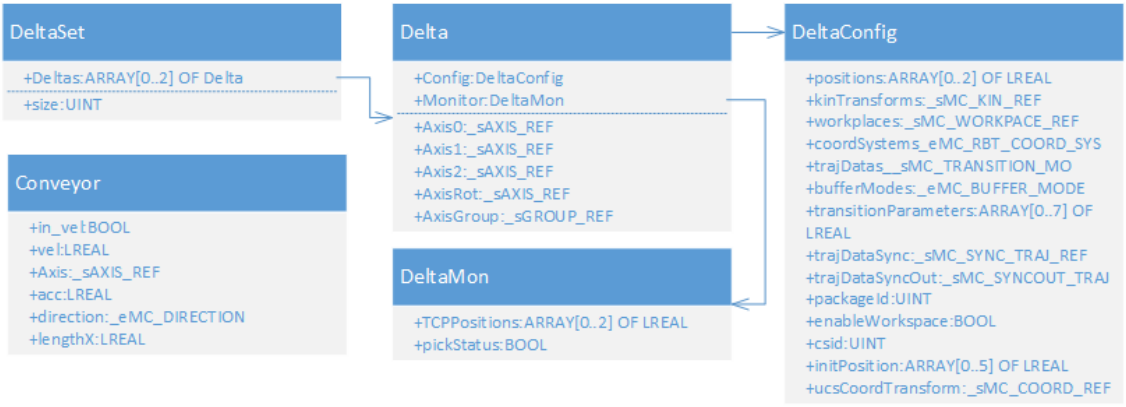


Abbildung 4.6: Datenstruktur zur Verwaltung und Einstellung der verwendeten Maschinen.

Die Datenstrukturen für die Delta-Roboter und des Fließbandes dienen lediglich der besseren Strukturierung und Zentralisierung aller Einstellungen und Definitionen, welche zur Steuerung benötigt werden.

4.3.2 Funktionsblöcke

Funktionsblöcke dienen der Kapselung von mehrfach verwendetet Anweisungsfolgen. Zur Generalisierung der Anweisungen können verschiedene Parameter übergeben werden, welche als Eingangs-Variable definiert werden. Die Eingangs-Variablen werden innerhalb eines Funktionsblocks als Konstanten behandelt und können nicht verändert werden. Aus diesem Grund sind Eingangs-Variablen ausschließlich als Werteparameter zu betrachten. Um die Ergebnisse des Funktionsblocks an das aufrufende Programm zurückzugeben, werden Ausgangs-Variablen definiert. Für den Fall, dass eine Eingangs-Variable innerhalb des

FBs verändert werden soll und implizit ebenfalls eine Ausgangs-Variable darstellt, können Ein/Aus-Variablen definiert werden. Mit Hilfe von Ein/Aus-Variablen können Referenzparameter realisiert werden. Außerdem müssen alle als Parameter übergebenen Komponenten, sowie virtuelle Achsen als Ein/Aus-Variable definiert werden, da sie innerhalb eines FBs nicht konstant bleiben. Jeder FB im Sysmac Studio besitzt mindestens eine Ausgangs- und eine Eingangs-Variable:

- **enable**: Diese Eingangs-Variable startet den FB, welcher die Anweisungen, analog zu einem Programm, iterativ, zustandsbehaftet und sequentiell ausführt.
- **done**: Diese Ausgangs-Variable signalisiert, dass der FB die Ausführung beendet hat. Dies ist notwendig, da die Funktionsblöcke parallel zu den Programmen ablaufen.

Wie im Abschnitt 4.1.4 beschrieben, können keine Variablen innerhalb eines Programms oder dynamisch während der Laufzeit deklariert werden. Aus diesem Grund ist es notwendig alle Anweisungen und Variablen, dessen Notwendigkeit bereits vor der Generierung des ST-Codes bekannt sind, mittels Funktionsblöcken zu kapseln. Alle Funktionalitäten, welche dem Benutzer von der DSL angeboten werden, sind bereits in Sysmac Studio als Funktionsblock implementiert. Dies bietet den Vorteil, dass die Generierung der Funktionalitäten und die dafür benötigten Variablen nicht nötig ist.

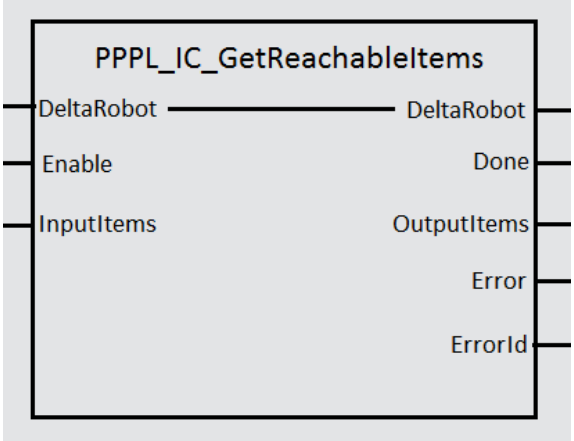


Abbildung 4.7: Die Laddercode darstellung des FBs `PPPL_IC_Get_Reachable_Items`.

Die einzelnen Funktionalitäten unterteilen sich in mehrere Gruppen und können detailliert in Kapitel 2 eingesehen werden. Im Pick and Place Packaging werden lediglich zwei Typen von Funktionsblöcken unterschieden:

- **Item-Controller (ICs)**: ICs dienen der Verarbeitung der Gegenstände und Packungen. Die Gegenstände können mittels Mengenoperationen gefiltert oder vereinigt werden. Es werden verschiedene FBs zur Unterscheidung der Gegenstände anhand ihrer Eigenschaften angeboten. Außerdem können die Gegenstände ermittelt werden,

welche in der Reichweite eines Delta-Roboters sind und in welchen Plätzen diese Gegenstände platziert werden können.

- **Motion-Controller (MCs)**: MCs dienen der Steuerung der Delta-Roboter. Diese FBs bieten die Möglichkeiten einen Gegenstand von dem Fließband zu greifen und ihn in einer Packung zu platzieren. Diese Funktionalitäten werden ausschließlich innerhalb von Sysmac Studio angeboten, da die DSL einer höheren Abstraktionsebene zugrunde liegt. Details können dem Kapitel 2 entnommen werden.

```
1  IF enable THEN
2    outputIndex:=0;
3    Done        :=FALSE;
4    Error       :=FALSE;
5    ErrorId     :=WORD#000;
6    FOR itemIndex:=0 TO 49 BY 1 DO
7      IF InputItemSet[itemIndex] > 0 THEN
8        findOneItemById(
9          InputItemSet[itemIndex],
10         itemTableIndex
11         );
12       tmp_Item:=ItemTable.items[itemTableIndex];
13       IF tmp_Item.placed  = FALSE
14       AND tmp_Item.picked = FALSE
15       AND tmp_Item.zPos  <= 295 THEN
16         //Entfernung zum Delta Montagepunkt berechnen
17         itemAbs:=SQRT( EXPT(DeltaRobot.Config.positions[0] - tmp_Item.xPos,2)+EXPT(
               DeltaRobot.Config.positions[1] -tmp_Item.yPos,2) );
18         IF itemAbs <= WBRadius THEN
19           OutputItemSet[outputIndex]:=tmp_Item.id;
20           outputIndex               :=outputIndex+1;
21         END_IF;
22       END_IF;
23     END_IF;
24   END_FOR;
25   IF outputIndex >0 THEN
26     Done    :=TRUE;
27     ErrorId:=WORD#000;
28     Error   :=FALSE;
29   ELSE
30     Done    :=FALSE;
31     Error   :=TRUE;
32     ErrorId:=WORD#1001;
33   END_IF;
34 ELSE
35   Done    :=FALSE;
36   Error   :=FALSE;
37   ErrorId:=WORD#000;
38 END_IF;
```

Listing 4.1: Implementierung des FBs: `PPPL_IC_Get_Reachable_Items`

An dieser Stelle wird zum besseren Verständnis der Funktionsblock `PPPL_IC_Get_Reachable_Items` im Detail vorgestellt: Die Abbildung 4.7 zeigt den Funk-

tionsblock in seiner Darstellung in Ladder-Code. Die Eingangs-Variablen werden auf der linken und die Ausgangs-Variablen auf der rechten Seite des FBs angeordnet. Die Ein/Aus-Variablen werden am oberen Ende des FBs angeordnet und durch einen durchgehenden Strich zwischen dem Ein- und Ausgang verdeutlicht. Der FB `PPPL_IC_Get_Reachable_Items` ermittelt abhängig von einem als Ein/Aus-Variable übergebenen Delta-Roboter alle erreichbaren Gegenstände auf dem Fließband. Die Implementierung des FBs kann dem Listing 4.1 entnommen werden.

Der FB wird mit der Eingangs-Variable `enable` gestartet und überprüft für jede der eingegebene Item-IDs in der globalen Tabelle ItemTable, ob der Abstand zum Delta-Roboter so gering ist, dass der Gegenstand gegriffen werden kann. Zunächst müssen anhand der eingegebenen Item-IDs aus der Liste InputItemSet mit Hilfe der Join-Funktion `findOneItemById` die entsprechenden Instanzen der Gegenstände ermittelt werden. Im Anschluss wird für jedes der ermittelte Gegenstände überprüft, dass es weder gegriffen noch platziert oder zu hoch ist, sodass es nicht gegriffen werden kann. Die Bestimmung des Abstands zwischen der Position eines Gegenstands auf dem Fließband und dem als Ein/Aus-Variable übergebenen Delta-Roboters erfolgt mittels des Satz des Pythagoras in der X-Y-Ebene des Fließbands. Die IDs der sich in Reichweite befindenden Gegenstände werden in der Liste `OutputItemSet` gesammelt. Nach Beendigung der Prozedur werden die entsprechenden Signale gesetzt und der FB beendet seine Arbeit. Falls sich kein Gegenstand in Reichweite des Delta-Roboters befindet wird der FB mit einem Fehler-Signal beendet. Die weiteren FBs sind analog realisiert und reduzieren den Aufwand der zu generierenden Funktionen aus der DSL.

4.3.3 Programme

Die Programme beinhalten den Ausführungskontext, welcher sich aus Instruktionen, Bedingungen und Schleifen zusammensetzt. Jedes Programm wird unabhängig von den anderen ausgeführt, sodass mehrere Programme parallel ablaufen.

Globale Variablen

Die globalen Variablen sind programmübergreifend verfügbar, sodass die Kommunikation zwischen den einzelnen Programmen oder eine Synchronisation realisiert werden kann. Außerdem werden die Listen der Struct-Instanzen in den globalen Variablen gehalten. Eine detaillierte Liste aller globalen Variablen wird in Tabelle 4.2 gezeigt. Neben den Listen der Struct-Instanzen existieren Variablen zur Speicherung eines Zustands und eines Signals zur Initiierung und Terminierung des Master-User-Programms (MUP).

Name	Datentyp	Kommentar
`RUN`	`BOOL`	Startet oder Terminiert das MUP.
`STATE`	`INT`	Zustandsvariable des MUP zur Synchronisation der DUPs.
`LostItems`	`ARRAY`[0..49] `OF UINT`	Liste aller Objekt-IDs, welche am Ende des Fließbands nicht gegriffen wurden.
`DefBOOL`	`ARRAY`[0..9] `OF BOOL`	Dient der Synchronisation der Definitions-Programme.
`Robots`	`DeltaSet`	Liste aller Delta-Roboter und ihrer Einstellungen.
`CharacteristicSetTable`	`CharacteristicSetTableRepo`	Struct-Instanz-Liste.
`CharacteristicTable`	`CharacteristicTableRepo`	Struct-Instanz-Liste.
`ItemTable`	`ItemTableRepo`	Struct-Instanz-Liste.
`PlaceTable`	`PlaceTableRepo`	Struct-Instanz-Liste.
`PackageTable`	`PackageTableRepo`	Struct-Instanz-Liste.

Tabelle 4.2: Die globale Variablen, welche programmübergreifend verfügbar sind.

MUP und DUPs

Wie im Kapitel 2 beschrieben muss jeder Delta-Roboter unabhängig von den anderen gesteuert werden können. Aus diesem Grund wird jeder Delta-Roboter und alle ihm zugehörigen Anweisungen in einem eigenen Programm, dem Delta-User-Programm (DUP) ausgeführt.

Zur Synchronisation dieser Programme wird ein weiteres Programm, das Master-User-Programm (MUP) benötigt, welches hierarchisch über denen der Delta-Roboter steht. So können initiale Anweisungen, wie das Aktivieren der einzelnen Servomotoren, welche unabhängig von den einzelnen Delta-Robotern ausgeführt werden müssen, im MUP realisiert werden. Des Weiteren können die einzelnen DUPs über die globalen Variablen gesteuert werden oder auf bestimmte Ereignisse warten. Die hierarchische Struktur der einzelnen Programme kann der Abbildung 4.8 entnommen werde. Der aus der DSL generierte ST-Code kann in das MUP und die DUPs eingefügt werden. Um die nötige Flexibilität und Dynamik zu ermöglichen werden innerhalb jedes Programms User-Variablen angelegt, welche im generierten Code genutzt werden können. Es werden zwanzig Variablen von jedem primitiven Datentyp angeboten. Zusätzlich werden Listen von primitiven Variablen, insbesondere von `UINT`s, angeboten um, die nötigen Ein- und Ausgabe-Listen von IDs für die verschiedenen FBs bereitzustellen.

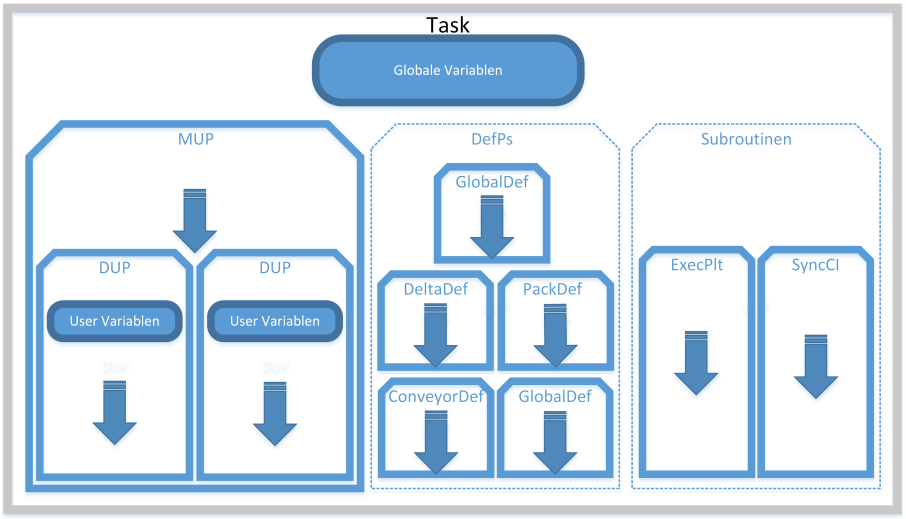


Abbildung 4.8: Hierarchische Programmstruktur innerhalb eines Tasks. Globale Variablen sind programmübergreifend verfügbar und dienen der Synchronisierung und Kommunikation. Die Definitions-Programme (DefPs) und Subroutinen werden parallel zum MUP gestartet und ausgeführt.

Definitions Programme

Neben dem MUP und DUPs werden weitere Programme benötigt, um die Initiierung der Variable vorzunehmen. Auf diesem Wege können beispielsweise die anfänglichen Positionen der Gegenstände und Eigenschaften in der DSL definiert und anschließend generiert und eingefügt werden. Diese Definitions-Programme (DefPs) (siehe Tabelle 4.3) initialisieren alle globalen Variablen und werden lediglich einmal zu Beginn aufgerufen. Es werden mehrere Programme zur besseren Strukturierung genutzt, welche sequentiell ausgeführt werden:

Nr.	Programm	Beschreibung
1.	`GlobalDef`:	Definiert die Anzahl der Roboter in der Szene.
2.	`CharacDef`:	Definiert alle Eigenschaften und Eigenschafts-Listen der Gegenstände.
3.	`PackDef`:	Definiert alle Packungen und Plätze.
5.	`DeltaDef`:	Definiert alle Delta-Roboter und deren Einstellungen.
6.	`ConveyorDef`:	Definiert das Fließband und seine Einstellungen.

Tabelle 4.3: Definitionsprogramme in der Reihenfolge ihrer Ausführung.

Subroutinen

Zur Realisierung verschiedener Abläufe innerhalb der Szene und der Informationsbereitstellung werden interne Subroutinen benötigt. Diese speziellen Programme arbeiten parallel

zu dem MUP und beginnen zeitgleich ihre Ausführung nach den DefPs. Die Programme werden benötigt um mehrere verschiedene wiederkehrende Aufgaben zu bewältigen, welche unabhängig von anderen Teilen der Szene ausgeführt werden müssen:

- **SyncCI**: Diese Subroutine aktualisiert die Positionen aller Gegenstände auf dem Fließband, abhängig von der Fließbandgeschwindigkeit, im $10ms$ Takt. Es werden nur Gegenstände aktualisiert, welche weder gegriffen, verpackt oder außerhalb des Fließbands sind.
- **ExecutionPlatform (ExecPlt)**: Um die nötigen Informationen über jeden Delta-Roboter und insbesondere deren TCP-Positionen, zu aktualisieren wird eine weitere Subroutine benötigt.

Kapitel 5

Visualisierungskomponente

In diesem Kapitel wird auf die Visualisierungskomponente eingegangen, mit deren Hilfe es möglich ist den Ablauf der Programme darzustellen. Abbildung 5.1 zeigt die Visualisierungskomponente in der Gesamtarchitektur.

Dazu werden im Folgenden zuerst die Grundlagen geschaffen, welche zum weiteren Verständnis nötig sind. Es wird zum einen der Roboter Simulator V-REP (s. Abb. 5.2) und zum anderen die Software *CX-Compolet* vorgestellt. Anschließend wird das Konzept worauf die Visualisierungsumgebung beruht erläutert. Im letzten Abschnitt dieses Kapitels wird letztendlich auf die Realisierung der Visualisierungskomponente eingegangen.

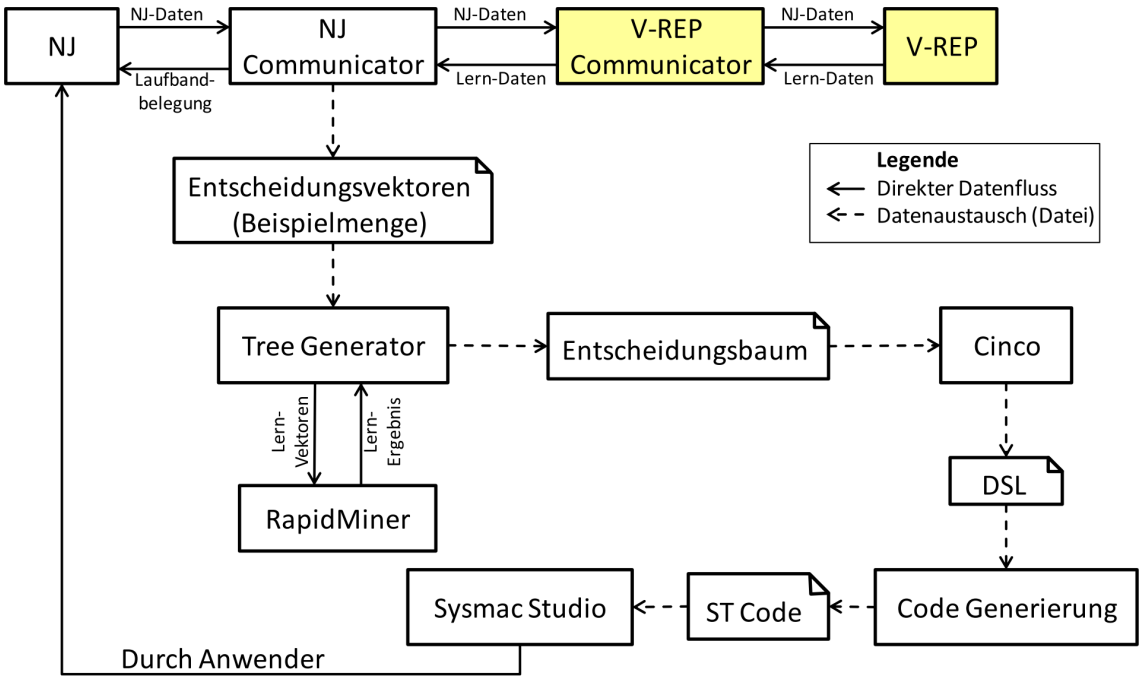


Abbildung 5.1: Kapitel 5, Komponenten in der Gesamtübersicht.

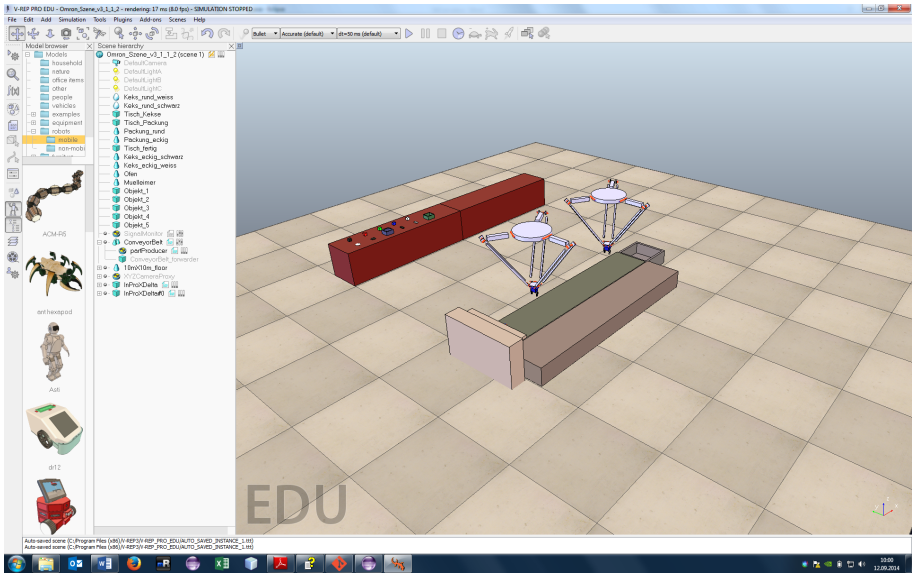


Abbildung 5.2: Der Roboter Simulator V-REP.

5.1 Grundlagen

Dieses Kapitel befasst sich mit dem Roboter-Visualisierungsprogramm V-REP. Es sollen die Möglichkeiten aufgezeigt werden, inwiefern diese Software zur Realisierung Der PG-Ziele beitragen kann. Dabei wird zunächst erläutert, was V-REP ist und was für Funktionen es bietet.

5.1.1 V-REP

V-REP ist eine Abkürzung für **V**irtual **R**obot **E**xperimentation **P**latform. Es ist ein Programm zur Verhaltensvisualisierung von Robotern. Es bietet eine umfassende Menge von Funktionen, unter anderem APIs für verschiedene Programmiersprachen, was die Möglichkeiten zur Erstellung einer Schnittstelle zwischen den einzelnen Komponenten der PG erleichtert, Physikberechnungen, welche die Visualisierung realer gestalten und viele weitere.

In V-REP kann auf unterschiedliche Arten programmiert werden. Es können bis zu sechs verschiedene Vorgehensweisen genutzt werden. Die einfachste und flexibelste Möglichkeit besteht in der Programmierung von eingebetteten Skripten. Diese Variante ist mit anderen V-REP Standardinstallationen kompatibel und erlaubt Einzelheiten während der Visualisierung zu personalisieren. Damit lassen sich allerdings nicht die Visualisierungseigenschaften verändern. Skripte werden in *Lua* [2] geschrieben. Wird allgemeingültiger Code benötigt, kann dies in Form von Add-Ons programmiert werden. Sie sind ebenfalls in Lua geschrieben und können beim Visualisierungsstart automatisch im Hintergrund gestartet werden oder als Funktionen fungieren. Sie eignen sich besonders für die Wiederverwendung und sollten deswegen generisch definiert werden. Ebenfalls ist es möglich V-REP

mit Plugins an die eigenen Bedürfnisse anzupassen. Neben diesen drei Varianten, welche unter der Regular API zusammengefasst sind, existieren noch drei weitere Möglichkeiten der Programmierung. Zum einen existieren die *Remote-API*[1], welche die Kommunikation mit externen Anwendungen wie Java, Python oder C/C++ erlaubt. Die ROS Schnittstelle gestattet einer externen Anwendung die Verbindung zu V-REP über das Robot Operating System. Als letztes gibt es die Möglichkeit eine Server/Client Verbindung aufzubauen.

Um den Delta-Roboter von außen zu steuern, wurde die Programmierung mit Hilfe der Remote-API gewählt, da dort eine Schnittstelle zur NJ geschaffen werden kann. Die Remote-API ist über C/C++, Pythonscript, Urbiscript, Matlab oder Java nutzbar. Damit V-REP den Aufbau einer Verbindung erwartet, muss im Mainscript, dass der Szene zugeordnet ist, die Funktion zum Starten der Remote-API im Initialisierungsbereich aufgerufen werden. Nachdem die Verbindung hergestellt ist, können Signale an V-REP übertragen werden. Um ein Signal zu senden, muss die entsprechende Methode der Remote-API in Java aufgerufen werden. Beim Senden eines Signals an V-REP muss der Übertragungsmodus (*OpMode*) festgelegt werden, es wurde dafür *simx_opmode_oneshot_wait* verwendet. Das bedeutet, dass die Signale versendet und gewartet wird bis eine Antwort von V-REP zurückkommt. Mit *simx_opmode_oneshot* dagegen werden die Signale asynchron versendet. Die Benutzung von *simx_opmode_oneshot_wait* ist nötig, da in V-REP die Signale nicht gepuffert und nacheinander abgearbeitet werden, sondern die Signale immer vom neuesten überschrieben und damit teilweise verloren gehen können. Detaillierte Informationen zu den OpModes und zum Aufbau der Verbindung sind im Manual der V-REP Remote-API zu finden.

5.1.2 CX-Compolet

CX-Compolet ist ein Paket von Softwarekomponenten der Firma OMRON, die es ermöglichen und vereinfachen Daten aus bzw. in einen Programmable Logic Controller (PLC) der Firma OMRON zu lesen bzw. zu schreiben. Ein PLC (deutsch: speicherprogrammierbare Steuerung) dient zur Steuerung einer Maschine oder einer Anlage. Die im Projekt verwendete NJ ist ein PLC, die zur Steuerung der Delta-Roboter verwendet wird.

Mit CX-Compolet wird das Erstellen von Programmen zur Kommunikation zwischen einem Computer und einem PLC vereinfacht. Mit Hilfe von CX-Compolet ist es möglich Speicher in einem PLC zu lesen und zu schreiben. Des Weiteren kann der Operationsmodus der PLC verändert und Error-Logs ausgelesen werden. Ein weiterer Vorteil von CX-Compolet ist die Möglichkeit auf Speicher in einer PLC mit Tag-Namen zuzugreifen und nicht mit konkreten Adressen arbeiten zu müssen. Auch können Variablen in einem Programm auf dem PLC gelesen und geschrieben werden. In der Abbildung 5.1 wird ein Überblick über die Hauptfunktionen von CX-Compolet gegeben. CX-Compolet kann mit Visual Basic.NET und Visual C#.NET benutzt werden [10].

Interface	Function	Description
Properties	Communications with OMRON PLCs	Specifies the PLC to communicate with, and reads network information
	Reading and writing I/O memory	Read and writes data in memory areas, such as the DM Area or CIO Area. For example, DM word 100 can be specified by using "D100ör by using a tag name.
	Operating status	Reads and changes the operating mode.
	Area information	Reads information such as the program area size and number of DM Area words.
	Error information	reads the value and error message when an error occurs.
	Other OMRON PLC information	Reads the model and reads and changes the clock.
	Getting tag information	Gets the NJ-series/CJ2 tag name list.
Methods	Reading and writing I/O memory	Reads and writes memory, such as consecutive words in the DM Area or CIO Area. For example, it is possible to specify the data type (integer, single, etc.) or change the data type (BCD, BIN, SBIN).
	Creating I/O tables	Creates the I/O tables for the present configuration.
	Force-setting, force-resetting and clearing bits	Force-sets, force-resets, and clears bits.
	Communications with OMRON PLCs	Specifies the PLC to communicate with.
	FINS service execution	Sends FINS commands and gets the responses that are received.
Events	Scheduled events	Events occur at regular intervals.

Tabelle 5.1: Die Hauptfunktionen von CX-Compolet [10].

Abbildung 5.3 zeigt ein Beispiel für die Verwendung von CX-Compolet. Außerdem ordnet die Abbildung 5.4 die Software CX-Compolet innerhalb des Softwarestapels der *FA Communications Software* [10] von Omron ein. Unterhalb der CX-Compolet befindet sich eine weitere Schicht, das Sysmac *Gateway*. Dabei handelt es sich um eine Middleware, die

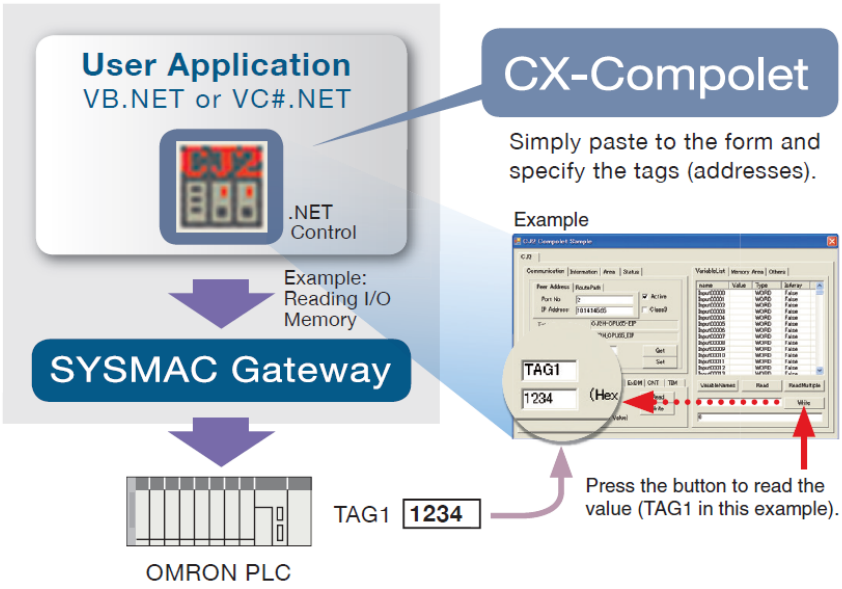


Abbildung 5.3: Beispiel Verwendung von CX-Compolet [10].

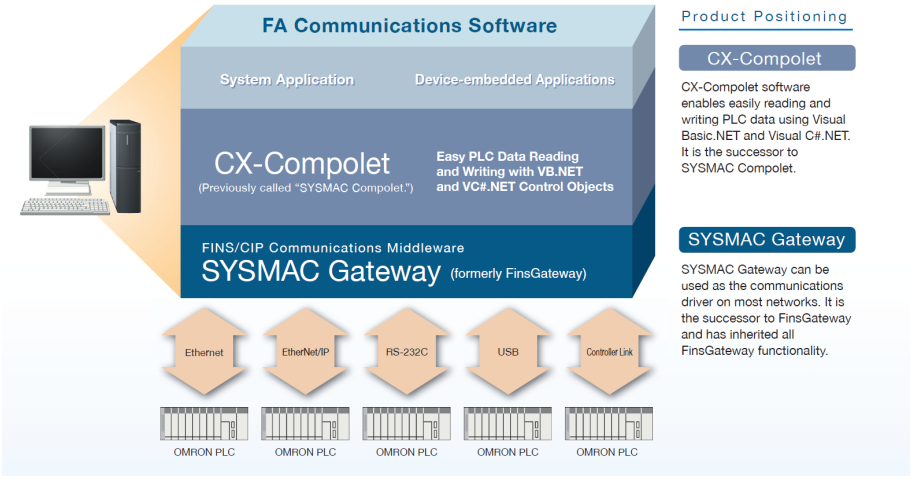


Abbildung 5.4: Einordnung von CX-Compolet in der Anwendung [10].

mit den einzelnen angeschlossenen PLCs über verschiedene Schnittstellen kommuniziert. Des Weiteren liefert sie die entsprechenden Treiber und einen virtuellen Speicher [10].

Es ist auch möglich ohne CX-Compolet, also nur mit der Hilfe von Sysmac Gateway mit einer PLC zu kommunizieren. In der Visualisierungskomponente wurde mit CX-Compolet gearbeitet. Die Gründe darfür werden im Abschnitt 5.2.1 erläutert.

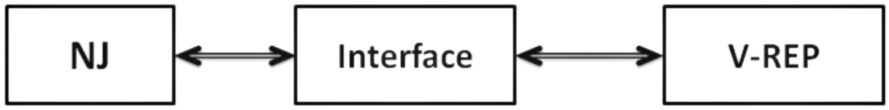


Abbildung 5.5: NJ und V-REP über eine Kommunikationsschnittstelle verbunden.

5.2 Konzept

Im Folgenden werden die Anforderungen sowie das daraus resultierende Konzept der Visualisierungskomponente beschrieben.

5.2.1 Anforderungen

Die Ausgangssituation ist, dass Daten aus der NJ bereitgestellt und diese mit Hilfe einer Visualisierung dargestellt werden sollen. Zur Visualisierung wird, wie auch schon in Abschnitt 5.1.1 erwähnt, das Visualisierungsprogramm V-REP verwendet. Mit diesem ist es möglich alle Objekte des Pick and Place Packaging-Prozesses darzustellen.

Da es nicht möglich ist Daten aus der NJ direkt an V-REP zu übertragen ist eine Kommunikationsschnittstelle erforderlich, welche diese Aufgabe übernimmt. Diese Situation ist in Abbildung 5.5 bildlich dargestellt und wird im Abschnitt 5.3 behandelt. Seitens der NJ besteht mit Hilfe, der im Abschnitt 5.1.2 beschriebenen CX-Compolet API, die Möglichkeit mit der NJ zu kommunizieren. Somit ist es möglich Daten aus der NJ zu lesen und zu schreiben. Die CX-Compolet API lässt sich mit den Programmiersprachen Visual Basic .NET, Visual C# .NET ansprechen. Neben der CX-Compolet API besteht mit Sysmac Gateway eine zusätzliche Möglichkeit über C bzw. C++ mit der NJ zu kommunizieren [10]. Auf der Seite von V-REP kann mittels Remote-API von V-REP mit dem Simulator kommuniziert werden [1]. Dadurch lassen sich Objekte im Simulator von außerhalb durch ein entsprechendes Programm ansteuern. Somit können die Daten aus der NJ, welche die Bewegungen der Delta-Roboter abbilden, mit Hilfe dieser API an V-REP zur Visualisierung weitergeleitet werden. Die Remote-API lässt sich mit den Sprachen C/C++, Python, Java, Mathlab, Octave und Urbi verwenden [1].

Zur Realisierung der Kommunikation gibt es, wie einleitend beschrieben, mehrere Möglichkeiten. Zur Wahl der Programmiersprache für die Kommunikationsschnittstelle würde sich zum einem C bzw. C++ eignen, da diese von beiden Seiten (NJ und V-REP) unterstützt werden. In diesem Fall würde der Zugriff auf die Daten der NJ durch das Sysmac Gateway erfolgen, welches sehr hardwarenah arbeitet [10]. Auf Grund der bestehenden Erfahrung von Projektgruppenmitglieder in der Programmiersprache C# wurde sich auf die Verwendung der CX-Compolet API geeinigt. Darüber hinaus bietet die CX-Compolet API durch ihre abstraktere Sicht auf die Daten der NJ eine schnellere Einarbeitungszeit.

Durch die zuvor beschriebene Kommunikationsseite zur NJ ist C# als Programmiersprache zur Realisierung der Schnittstelle zwischen NJ und V-REP vorgegeben. Allerdings

existiert keine Realisierung der V-REP Remote-API in der Sprache C#. Es existiert lediglich ein inoffizieller Wrapper, geschrieben in C#, der Aufrufe der C++ Remote-API von V-REP kapselt (siehe [15]). Dieser Wrapper bietet jedoch nur eine Teilmenge der Funktionalität der Remote-API. Da die Möglichkeit besteht, dass im Projektverlauf weitere Anforderungen an den Funktionumfang des Wrappers dazu kommen, wurde von der Verwendung dieser Variante abgesehen.

Aus den vorgestellten Gründen wurde das Konzept der Kommunikationsschnittstelle in zwei unterschiedliche Module aufgegliedert, welche jeweils eine Seite der Kommunikationsschnittstelle realisieren. Beide Module kommunizieren dabei untereinander über das Netzwerk (siehe Kapitel 5.3.1). Die Kommunikationsschnittstelle teilt sich somit zum einen in das Modul NJCom, welches die Kommunikation zur NJ realisiert, und zum anderen in das Modul V-REP Communicator auf. Das Modul V-REP Communicator ist in Java geschrieben, kommuniziert über eine Socketverbindung mit NJCom und realisiert die Kommunikation zum Visualisierungsprogramm V-REP. Die Abbildung 5.6 stellt das Konzept der Kommunikationsschnittstelle mit seinen zwei Teilmodulen bildlich dar. Ein Vorteil der Aufsplittung in zwei Module ist die einfache Ansteuerung und die Möglichkeit der Verwendung der vollständigen Remote-API von V-REP. Des Weiteren besteht somit die Möglichkeit beide auf verschiedene Systeme auszuführen, um mögliche Performanzprobleme bei der Visualisierung entgegenzuwirken.

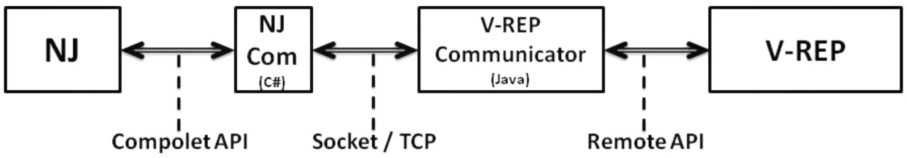


Abbildung 5.6: Konzeptioneller Aufbau der Visualisierungskomponente.

5.3 Realisierung

In diesem Abschnitt werden die Anforderungen und die Realisierung, der im Konzept vorgestellten Komponenten, beschrieben.

5.3.1 NJCom

Der NJCom ist die zentrale Steuerungskomponente der Visualisierung. Er dient der Aufbereitung der Programminstanzen für die Wiedergabe in V-REP und aggregiert die inferierten Daten aus den Benutzerinteraktionen als Entscheidungsvektoren für den Tree Generator.

In den zwei Phasen des Lernroundtrips 6.3 werden unterschiedliche Anforderungen gestellt. Der NJCom bietet daher drei verschiedene Betriebsmodi: Decision(für die Entscheidungsphase), Veto(für die Vetophase) und Demo. In allen drei Modi werden die erstellten

Programminstanzen via V-REP Communicator an V-REP gesendet und dort visualisiert. Verschieden sind die Erstellung der Programminstanzen und die Interaktionsmöglichkeiten für den Benutzer.

Im Demomodus sind keine Interaktionen für den Benutzer möglich. Dagegen sind im Decision- und im Vetomodus Interaktionen in V-REP notwendig. Die Ergebnise dieser Benutzerinteraktionen werden über V-REP Communicator an den NJCom gesendet und als Entscheidungsvektoren gespeichert.

In der Entscheidungsphase wird eine Fließbandbelegung vom NJCom generiert und mit dieser Fließbandbelegung ein Programminstanzen erstellt. Die Angaben für die Verpackungen werden aus der NJ ausgelesen.

In der Vetophase wird eine Fließbandbelegung in NJCom generiert. Die Fließbandbelegung wird anschließend auf die NJ übertragen. Auf der NJ wird ein Programm mit dieser Fließbandbelegung ausgeführt. Die resultierend Programminstanz wird aus der NJ ausgelesen und aufgezeichnet. Die Aufzeichnung kann im Anschluss in V-REP visualisiert werden.

Im Demomodus ist das Abspielen einer abgespeicherten Aufzeichnung möglich.

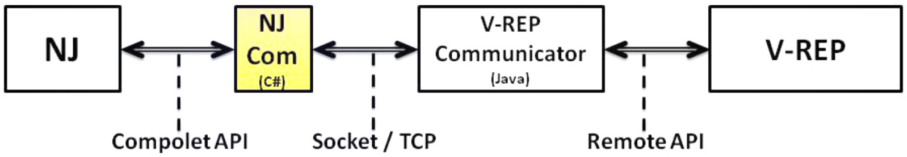


Abbildung 5.7: Kommunikation mit der NJ und dem VRepCommunikator über NJCom.

In den folgenden Abschnitten wird die Umsetzung der Anforderungen an den NJCom im Detail beschrieben.

Die Benutzeroberfläche

Für die Steuerung des NJCom wurde eine graphische Benutzeroberfläche implementiert. In der graphischen Benutzeroberfläche sind über die oberste Menüleiste die Projektverwaltung und die Konfiguration zugänglich. In der Projektverwaltung können Projekte angelegt, gespeichert oder geladen werden. In der Konfiguration sind die Verbindungsdaten zur NJ und die Parameter für die Erzeugung der Objekte für das Fließband einstellbar 5.3.1.

Im Menü kann der Modus festgelegt werden. Die Auswahl erfolgt durch Selektion eines der drei Radiobuttons. Decision ist in der Entscheidungsphase, Veto in der Vetophase und Demo zum Abspielen zu wählen.

Für jeden Modus stehen spezifische Funktionalitäten in der Toolbar unter den drei Radiobuttons zur Verfügung. Zum Beispiel sind im Decision Modus 5.3.1 (von links nach rechts) die Generierung einer Programminstanz, das Speichern einer Programminstanz, das Laden einer Programminstanz, das Abspielen einer Programminstanz, Pause, Stop,

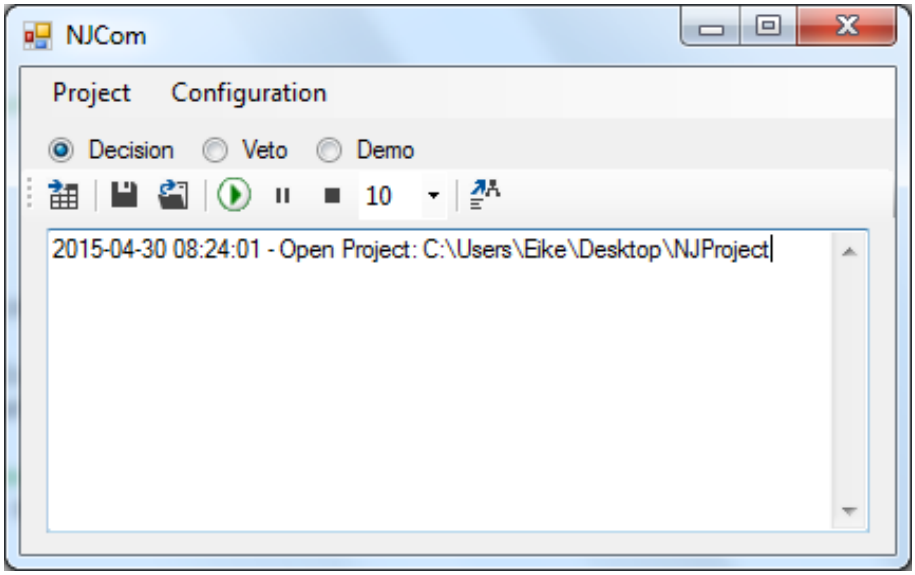


Abbildung 5.8: Die Benutzeroberfläche des NJCom.

eine Auswahl für die Wiedergabegeschwindigkeit und eine Exportfunktion für die Entscheidungsvektoren implementiert.

Der ConveyorLayoutgenerator

Mit dem *ConveyorLayoutgenerator* kann eine Fließbandbelegung generiert werden. Unter Angabe von Parametern wird eine zufällige Konstellation bzw. Anordnung von Objekten auf dem Fließband erstellt. Im Decisionmodus wird auf der Grundlage der Fließbandbelegung eine Programminstanz erstellt. Im Vetomodus wird die Fließbandbelegung in die NJ übertragen.

In beiden Modi erfolgen die Aufrufe des ConveyorLayoutgenerator in der Benutzeroberfläche des NJCom. Die generierten Daten werden zunächst im Hauptspeicher gehalten. Auf eine Speicherung der Daten auf der Festplatte wurde aus Gründen der Performanz verzichtet.

Eine Fließbandbelegung oder eine Programminstanz kann zu Testzwecken in einer CSV-Datei ausgegeben werden. Der Aufbau der CSV-Datei ist der Tabelle 5.2 zu entnehmen. Die erste Spalte gibt die eindeutige ID eines Objektes an. In den darauf folgenden drei Spalten werden die Koordinaten der Objekte angegeben (Angaben in mm), wobei die Z-Koordinate die Höhe eines Objektes angibt, gefolgt von der Rotation der Objekte (Angaben in Grad). Mit der Spalte *Picked* gleich false wird angegeben dass ein Gegenstand nicht von einem Delta-Roboter gegriffen wurde und mit *Placed* gleich false, dass ein Gegenstand noch nicht in einer Verpackung ist. Das setzen der Werte für Picked und Placed ist nötig um die entsprechenden Variablen in der NJ zu initialisieren, wenngleich die Werte für alle Zeilen gleich sind. Die letzte Spalte gibt die ID des CharacteristicSets eines Gegenstandes an.

Gegenstand-ID	X-Koord.	Y-Koord.	Z-Koord.	Rotation	Picked	Placed	Typ
1	-93	-187	40	25	False	False	5
2	-228	129	40	110	False	False	1
3	-298	199	40	-80	False	False	4
4	-510	53	40	72	False	False	8
5	-593	-176	40	178	False	False	5
6	-724	-42	40	-169	False	False	5
7	-794	49	40	-28	False	False	2
8	-806	175	40	-86	False	False	2
9	-937	64	40	37	False	False	8
10	-1036	-149	40	35	False	False	6
11	-1158	-3	40	144	False	False	7
12	-1237	190	40	32	False	False	5
13	-1400	-148	40	-152	False	False	1
14	-1679	-188	40	-136	False	False	7
15	-1960	11	40	126	False	False	7
16	-2035	-69	40	136	False	False	1
17	-2048	-168	40	-72	False	False	4
18	-2203	-52	40	-4	False	False	1
19	-2278	118	40	-108	False	False	7
20	-2278	195	40	129	False	False	5

Tabelle 5.2: Aufbau der Ausgabe CSV-Datei für 20 Gegenstände.

Die GUI des ConveyorLayoutgenerator ist in der Abbildung 5.10 zu sehen. Mit Hilfe der Parameter in der Mitte der GUI lässt sich der Generator konfigurieren und mit dem Button Generate CSV wird die bereits erwähnte CSV-Datei erstellt. Der Generator ist so entworfen worden, dass der Anwender möglichst viele Freiheiten bei der Erzeugung einer Belegung hat. Als Konsequenz daraus gibt es Parameter in der GUI, welche im Folgenden aufgezählt und erläutert werden (wobei alle Koordinaten dem globalen Koordinatensystam aus 4.2.1 entsprechen):

- **ConveyorLayout**: Mit diesem Parameter ist es möglich den Namen der Ausgabedatei festzulegen.
- **NumberOfItems**: Hiermit lässt sich die Anzahl der Gegenstände, welche erzeugt werden sollen einstellen.
- **MaxXDistanceToNextCluster, MinXDistanceToNextCluster** (in mm): Diese beiden Parameter stellen den minimalen bzw. maximalen X-Koordinatenabstand

zwischen zwei Clustern von Gegenständen ein. Ein Cluster von Gegenständen ist ein Bereich in dem alle Gegenstände zum vorherigen Gegenstand einen X-Koordinatenabstand innerhalb eines Intervalls haben. Dieses Intervall wird durch die nachfolgenden Parameter beschrieben. Wie viele Gegenstände in einem Korridor seien können, hängt von der Wahl des Parameters YDistanceBetweenItems ab. Je größer dieser ist, desto weniger Gegenstände passen in Y-Richtung in einem Korridor. Die Abbildung 5.9 stellt den Zusammenhang zwischen einem Cluster und den beiden Parametern da. Zu sehen sind zwei Cluster, deren Abstand von einander zwischen MaxXDistanceToNextCluster und MinXDistanceToNextCluster liegt.

- **MaxXRangeCluster, MinXRangeCluster** (in mm): Wie im vorherigen Punkt beschrieben legen diese Parameter das Intervall, innerhalb dessen der X-Koordinatenabstand der Gegenstände zum vorherigen Gegenstand in einem Cluster gewählt wird, fest.

- **TypeRange**: Mit Hilfe des TypeRange ist es möglich eine Liste der Typen anzugeben, von denen Gegenstände erzeugt werden sollen.

- **YDistanceBetweenItems** (in mm): Dieser Parameter legt den Mindestabstand der Y-Koordinaten zweier Gegenstände fest. Dieser Parameter wurde nicht als Intervall realisiert um die Komplexität der Parametrisierung nicht unnötig zu erhöhen.

- **YRange** (in mm): Durch diesen Parameter wird der Y-Bereich in dem überhaupt Gegenstände erzeugt werden dürfen festgelegt (z.B. beschränkt durch die Breite des Fließbandes). Wobei die Mitte des Fließbandes Null ist und der Wert des Parameter in positiver als auch in negativer Y-Koordinatenrichtung von dem Nullpunkt aus verwendet wird.

Weiterhin ist es möglich die Rotation eines Gegenstandes festzulegen (vordefiniert ist hier ein Intervall von $]-180, 180]$ Grad).

Das Konzept des Clusters und der Parameter MaxXRangeCluster, MinXRangeCluster wurde auch mit der Begründung entwickelt, dem Anwender möglichst viele Freiheiten beim Design der Fließbandbelegung zu lassen. So ist es damit möglich Gegenstände in Form von Gruppen auf dem Fließband zu erzeugen. Das ermöglicht z.B. eine Anordnung von Gegenständen auf dem Fließband bei denen der X-Koordinatenabstand zwischen zwei Gruppen größer ist, als der X-Koordinatenabstand der Gegenstände in einer Gruppe. Abbildung 5.9 stellt das Konzept des Clusters noch einmal graphisch dar.

Um eine möglichst zufällige Verteilung der Gegenstände auf dem Fließband zu bekommen, verwendet der ConveyorLayoutgenerator einen Zufallszahlengenerator. Dieser wählt die Werte aus den Intervallen [*MaxXDistanceToNextCluster, MinXDistanceToNextCluster*] und

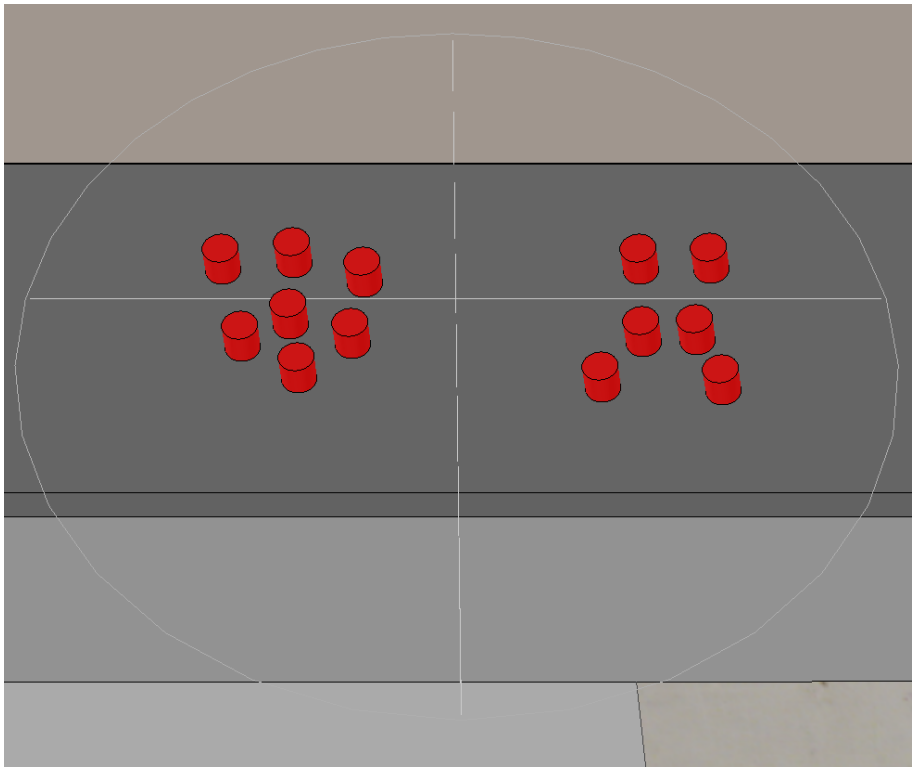


Abbildung 5.9: Zwei Cluster bei denen der X-Koordinatenabstand zwischen den Clustern größer ist, als der X-Koordinatenabstand der Gegenstände in einem Cluster.

[*MaxXRangeCluster, MinXRangeCluster*] und aus dem *YRange* zufällig aus. Weiterhin werden auch die Rotation und der Typ eines Gegenstandes zufällig gewählt. Wobei eine Abhängigkeit zwischen den Werten besteht, so dass keine Gegenstände an Stellen erzeugt werden wo sich schon andere Gegenstände befinden, um eine Kollision der Gegenstände zu vermeiden (auf den Mittelpunkt eines Gegenstandes bezogen). Dennoch ist es nötig, dass bei der Erzeugung einer Belegung die Maße aller Gegenstände bekannt sind, um die oben genannten Parameter richtig festzulegen, damit eine Kollision der Gegenstände bei der Erzeugung vermieden werden kann und die Gegenstände nicht seitlich vom Fließband fallen können.

Für die Erstellung einer Programminstanz werden eine generierte Fließbandbelegung, die Abtastrate und die Fließbandgeschwindigkeit verwendet. Mit der Abtastrate werden Zeitpunkte im Programmablauf und mit der Geschwindigkeit werden die Positionen der initialen Fließbandbelegung zu diesen Zeitpunkten berechnet. Die Fließbandbelegungen für die jeweiligen Zeitpunkte bilden die Programminstanz.

Auslesen der Daten

Die Laufzeitinformationen über das Roboterszenario werden durch den ConveyorLayoutgenerator und die NJ erstellt und für den weiteren Datentransfer aufbereitet. Auch wenn

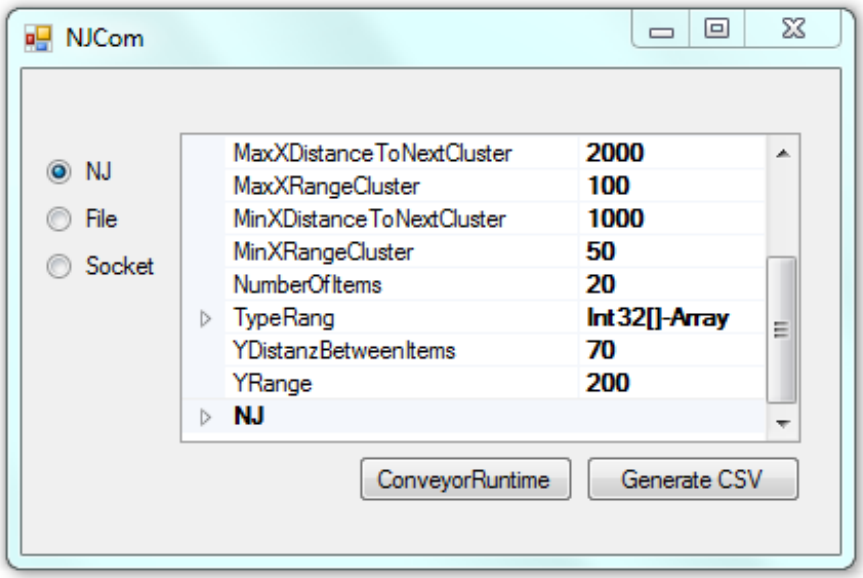


Abbildung 5.10: GUI des ConveyorLayoutgenerator.

sich die Aufbereitung der Daten in den zwei Phasen (Entscheidungs- und Vetophase) unterscheiden, müssen immer folgende Schritte durchlaufen werden: Verbindungsaufbau mit der NJ, Übertragung starten, Übertragung stoppen und Verbindung beenden. Für die Implementierung wurde daher, ein generischer Aufbau verwendet. Abbildung 5.11 zeigt die Konfiguration der Verbindung zur NJ. Nachfolgend werden der Verbindungsaufbau sowie der Start der Übertragung für beide Datenquellen beschrieben.

Entscheidungsphase

Die Laufzeitinformationen bestehen aus der Programminstanz und den Informationen über die verwendeten Verpackungen. Die Programminstanz wird durch den ConveyorLayoutgenerator generiert. Die Daten für die verwendeten Verpackungen werden aus der NJ ausgelesen. Das Auslesen der Daten aus der NJ wird mittels der CX-Compolet API ermöglicht. Laufzeitinformationen lassen sich hier über Methoden wie *ReadVariable* bzw. ReadVariables auslesen. Wenn alle Daten vorliegen, werden diese vor dem Start der Übertragung konvertiert (siehe Teilabschnitt 5.3.1 Aufbereitung der Daten).

Beim Starten der Übertragung werden die Daten anschließend nacheinander über die Socket-Verbindung transferiert.

Das Stoppen der Übertragung führt dazu, dass die Ausführung zurückgesetzt wird. Beim erneuten Start beginnt die Ausführung beim ersten Datensatz.

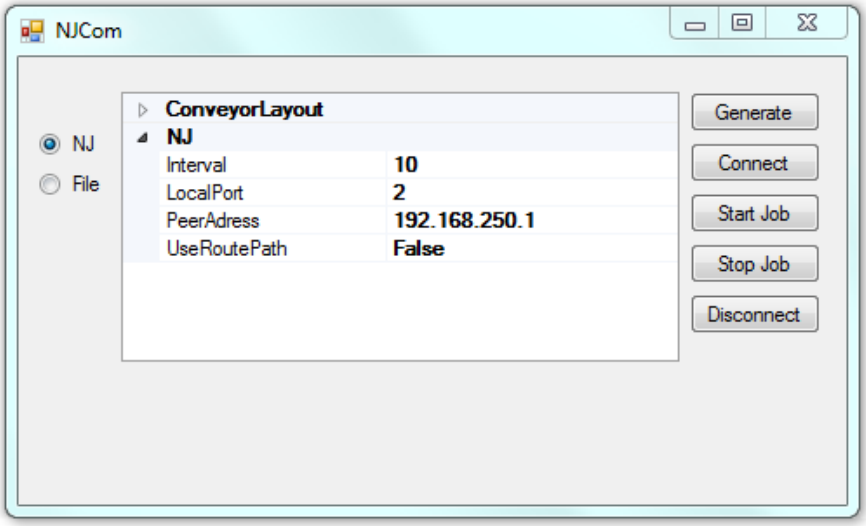


Abbildung 5.11: Der NJComManager.

Vetophase

Im Gegensatz zur Entscheidungsphase wird die Programminstanz in der Vetophase erst aufgezeichnet und kann im Anschluss visualisiert werden. Zunächst wird die NJ nach dem Verbindungsaufbau mit Gegenstandsdaten initialisiert. Dazu wird eine vom ConveyorLayoutgenerator generierte Fließbandbelegung verwendet. Durch die CX-Compolet API lässt sich anschließend die NJ über die *WriteVariable*-Methode mit der Fließbandbelegung konfigurieren. Vor dem Start der Übertragung stehen nicht die kompletten Laufzeitdaten zur Verfügung. Diese werden von der NJ berechnet. Deshalb werden nur die Bezeichner im Vorfeld verwendet, um die Daten bei der Übertragung schneller zu konvertieren (siehe Teilabschnitt 5.3.1 Aufbereitung der Daten)
Beim Starten einer Aufnahme wird die Fließbandbelegung zu NJ übertragen. Die NJ wird initialisiert. Dann werden alle notwendigen Variablen aus einem Array aus der NJ ausgelesen, konvertiert und im Hauptspeicher gespeichert. Der Benutzer kann die Programminstanz dauerhaft in einer Datei speichern und mit Hilfe von V-REP abspielen.

Aufbereitung der Daten

Der NJCom erhält Daten aus zwei unterschiedlichen Datenquellen (ConveyorLayoutgenerator und NJ) und muss diese in einem einheitlichen Protokoll an den VRepCommunicator übertragen. Die Verarbeitung und Übertragung der Daten an dem VRepCommunicator sollte nicht übermäßig zu Latenzen in der Visualisierung führen. Aus diesen Gründen werden bereits vor der Übertragung einige Informationen über die Bezeichner (Spalten- bzw.

Variablennamen) aus der Datenquelle aufbereitet, sodass die Übertragung wenig Laufzeit in Anspruch nimmt. Dazu werden alle Bezeichner der zu übertragenden Daten als eine Liste einem Parser übergeben. Anhand dieser Liste werden Indizies zu den Bezeichnern und dessen korrespondieren Objektvariablen der VREP-Objekte berechnet. Anschießend müssen bei der Übertragung die entsprechenden Daten in Form einer Liste, passend zur Liste der Bezeichnern, dem Parser übergeben werden. Für dieses Verfahren werden folgende Schritte durchlaufen:

1: Konvertierung der Daten in ein einheitliches Format
Im ersten Schritt werden die Bezeichner in der Liste in ein einheitliches Format umgewandelt. Dies geschieht durch Regular Expressions und entsprechenden Mappinginformationen der jeweiligen Datenquelle, die als Suchpattern verwendet werden. Dieser Schritt sorgt dafür, dass nur eine Implementierung der Datenübertragung für verschiedene Datenquellen notwendig ist. Das einheitliche Format setzt sich zusammen aus VREP-Objekttyp (Roboter, Fließband, Verpackungsobjekt), Instanz-Nummer sowie die Eigenschaften (`CharacteristicSetId`) des Gegenstands. Diese Informationen werden in den folgenden Schritten benötigt, um die Indizes der einzelnen Bezeichner zu berechnen.

Schritt 2: Indizes berechnen
Als nächstes wird aus den konvertierten Bezeichnern eine Datenstruktur aufgebaut, welches sich die Listenindizes aller Daten aus der Datenquelle merkt. Abbildung 5.12 zeigt den Aufbau der Datenstruktur anhand der Bezeichner-Liste. Zu Beginn wird der Bezeichner in die Bestandteile VREP-Objekttyp, Instanz-Nummer und Objektvariable zerlegt. Der VREP-Objekttyp wird als Schlüssel in der *HashMap* `typeMap` abgelegt. Dieser zeigt auf ein Array von `objectMaps` eines Objekttyps. Die `objectMap` besteht wiederum aus Schlüssel-Werte-Paaren, welche den Index aus der Bezeichner-Liste zu einer Objektvariable ablegt.

Schritt 3: Ausführung
Nachdem die Indizes berechnet wurden, wird für jeden Ausführungszyklus über die aufgebaute Datenstruktur iteriert und entsprechende Objekte erzeugt (Abbildung 5.13). Der Parser bekommt an dieser Stelle eine Liste mit Daten, die entsprechend der Bezeichner-Liste aufgebaut ist. Anhand der Instanz-Nummer wird zu jedem Schlüssel in der `typeMap` ein Objekt instantiiert, welches die dazugehörigen Objektvariablen aus der Daten-Liste

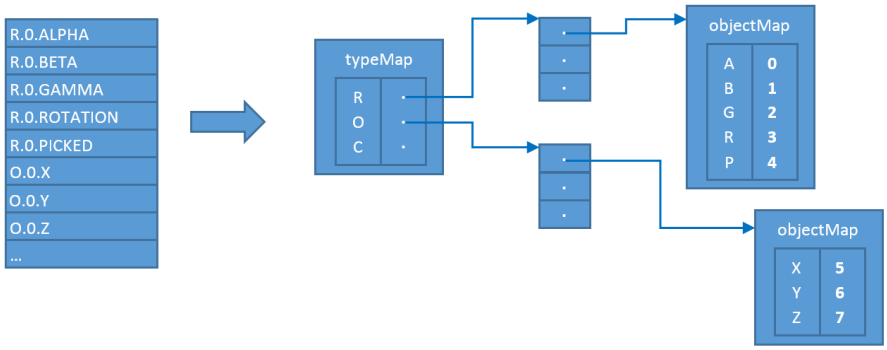


Abbildung 5.12: Berechnung der Indizies auf Basis der Bezeichner-Liste

entnimmt. Nachdem über die Datenstruktur iteriert wurde, resultiert eine Liste mit Objekten zu den einzelnen VREP-Objekttypen. Diese können anschließend über die Socket-Verbindung übertragen werden.

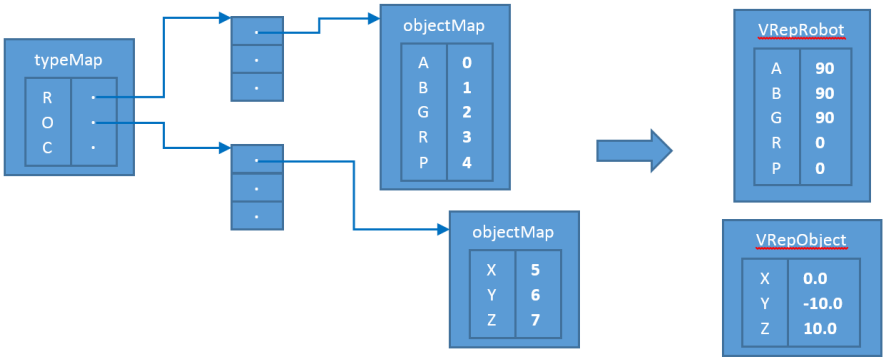


Abbildung 5.13: Konvertierung der Daten während der Ausführungszeit.

Kommunikation mit dem V-REP Communicator

Ein zentraler Bestandteil der Visualisierungskomponente ist die Kommunikation zwischen den beiden Teilmodulen NJCom und V-REP Communicator. Aufgrund der Anforderung, die beiden Teilmodule während einer Visualisierung auch auf getrennten Rechnern ausführen zu können (siehe Kapitel 5.2.1), kommunizieren die beiden Module über das Netzwerk miteinander. Hierbei kann wahlweise auf das *TCP* oder das *UDP* Protokoll als Transportprotokoll zur Übertragung der Daten zurückgegriffen werden. Beide Protokolle sind in den Kommunikationseinheiten der Module implementiert und können vom Benutzer eingestellt werden.

Die Kommunikation der beiden Module wird von der NJCom gestartet. Sie agiert somit als Initiator der Kommunikation, da diese in Verbindung mit der NJ steht und damit

Zugriff auf notwendige Daten hat, welche zur Visualisierung in V-REP benötigt werden. Der V-REP Communicator hingegen lauscht auf eingehende Nachrichten vom NJCom.

Für eine korrekte Visualisierung eines Pick and Place Packaging-Prozesses der NJ benötigt die Visualisierungssoftware V-REP, und damit das Modul V-REP Communicator, diverse Daten aus der NJ. Einige der benötigten Daten dienen einem unterschiedlichen Zweck und werden zur besseren Verwaltung in unterschiedliche Nachrichtentypen eingeteilt. Dabei handelt es sich um die folgenden Typen:

- **Roboterdaten:** Um korrekte Bewegungen der Delta-Roboter simulieren zu können, muss V-REP stets über die aktuellen Werte sowie Veränderungen der Motorenwinkel der Roboter verfügen. Neben den Werten zur Steuerung der Arme des Delta-Roboters werden ebenfalls der Winkel des TCP, sowie der Status benötigt, ob der Delta-Roboter aktuell ein Gegenstand greift oder nicht.
- **Objektdaten:** Zur Darstellung der unterschiedlichen Objekte, welche von dem Delta-Roboter verpackt werden sollen, werden Informationen über das Aussehen und der Form der einzelnen Objekte benötigt. Um eine bessere Performanz bei der Visualisierung innerhalb von V-REP zu erzielen, werden vordefinierte Objekte mit unterschiedlichen Attributen definiert. Während der laufenden Visualisierung werden diese Objekte dann verwendet und als neues Objekt von V-REP auf dem Fließband erzeugt anstatt dynamisch unterschiedliche Objekte anhand der Eigenschaften zu erzeugen (siehe Kapitel 5.3.3). Neben der Objekteigenschaft (`ChracteristicSetId`) werden bei jeder Übertragung auch die aktuellen Positionen der Objekte mitgeliefert.
- **Objekteigenschaften:** Objekteigenschaften sind verschiedene Attribute die ein Objekt aufweisen kann. Diese müssen nicht unbedingt das Äußere beschreiben, so dass in V-REP diese Eigenschaften textuell anzeigt werden. Zu Beginn des Durchlaufs werden die Objekteigenschaften für jeden Objekttyp an V-REP verschickt und an das zugehörige Objekttyp zugeordnet.

Neben diesen drei Nachrichtentypen gib es noch zwei weitere. Diese dienen zur Steuerung von bestimmten Ereignissen:

- **Reset V-REP:** Diese Nachricht dient dem Neustart der Visualisierung, so dass V-REP in den Anfangszustand versetzt wird.
- **Entscheidungssignal:** Diese Nachricht signalisiert V-REP, dass eine Entscheidung getroffen werden soll. Es werden keine neuen Nachrichten verschickt, bis V-REP eine Antwort auf das Entscheidungssignal liefert. Sobald eine Antwort erhalten wurde, wird diese an NJCom zur weiteren Verarbeitung weitergereicht. Die Antworten werden für die Lernkomponente benötigt. In Kapitel 6 wird die Lernkomponente genauer thematisiert.

Alle Nachrichtentypen werden als *Payloads* der Kommunikationspakete, des vom Benutzer zur Kommunikation ausgewählten, Transportprotokolls versendet. Der Aufbau der einzelnen Nachrichtentypen ist sehr einfach gehalten. Dies ermöglicht eine schnelle und simple Generierung von Nachrichtenpaketen auf Seiten der NJCom sowie eine einfache Realisierung eines Parsers innerhalb des V-REP Communicator. Jeder Nachrichtentyp ist in einzelne Blöcke unterteilt, die jeweils einen eigenen Wert repräsentieren. Die enthaltenen Werte sind als UTF-8 Strings formatiert. Die Trennung einzelner Blöcke erfolgt durch ein spezielles Literal - dem Semikolon. Der erste Block aller Nachrichtentypen enthält einen Identifier, der den aktuellen Nachrichtentyp identifiziert. Die entsprechenden Werte des Identifiers lauten:

- **R:** für den Nachrichtentyp der Roboterdaten,
- **O:** für den Nachrichtentyp der Objektdaten,
- **C:** für den Nachrichtentyp der Objekteigenschaften,
- **S:** für den Nachrichtentyp zum Reset von V-REP,
- **D:** für den Nachrichtentyp des Entscheidungssignals.

Zum Ende eines Nachrichtentyps enthalten alle denselben Wert im letzten Nachrichtenblock. Dabei handelt es sich um den Wert des *Carriage Return Line Feed*. Dieser wird benötigt, wenn als Kommunikationsprotokoll TCP verwendet wird, da dort statt einzelner Pakete ein Datenstrom vorliegt und somit das Ende eines einzelnen Nachrichtentypen festlegt. In den folgenden Abschnitten wird der Aufbau der Nachrichtentypen im Detail beschrieben.

Roboterdaten Nachrichtentyp

Eine grafische Darstellung des Aufbaus des Roboterdaten Nachrichtentyps ist in Abbildung 5.14 zu finden. Die Roboterdaten beginnen mit dem Identifier *R* im ersten Block. Im zweiten Block ist die *Roboter ID* enthalten. Diese spezifiziert den Delta-Roboter innerhalb der V-REP Szene für den die folgenden Roboterdaten adressiert sind. Es folgen in den nächsten drei Blöcken die Winkelwerte im Bogenmaß für die drei Motoren des Delta-Roboters. Die darauf folgenden drei Blöcke enthalten die Winkelwerte der Motoren und ermöglichen die übertragenen Bewegungen nachzuvollziehen. Die nächsten zwei Blöcken enthalten Daten, die den TCP des Roboters betreffen. Der erste dieser Blöcke zeigt den aktuellen Winkelwert des TCPs an, während der zweite Block festlegt, ob der TCP aktuell zugreift. Der letzte Block der Roboterdaten ist der zuvor beschrieben Carriage Return Line Feed.

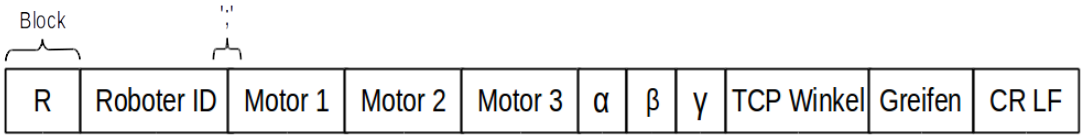


Abbildung 5.14: Nachrichtentyp für Roboterdaten.

Objektdaten Nachrichtentyp

Die Objektdaten zu einer V-REP Szene werden durch den Objektdaten Nachrichtentyp beschrieben. Der Aufbau dieses Nachrichtentyps ist in der Abbildung 5.15 dargestellt. Der Identifier im ersten Block des Nachrichtentyps enthält den Wert *O*, wodurch spezifiziert wird, dass es sich hierbei um Objektdaten handelt. Der zweite Block beschreibt die *Objekt ID*. Diese ID gibt an auf welches Objekt, der innerhalb der V-REP Szene verfügbaren zu verpackenden Objekte, sich die folgenden Daten beziehen. Die nächsten drei Blöcke beinhalten die X-,Y- und Z-Position des Objektes. Diese werden benötigt wenn das Objekt bisher noch nicht in der V-REP Szene existiert. In diesem Fall wird das Objekt an der beschriebenen Position neu erzeugt. Dies entspricht dem Moment im realen Prozess, in dem die Kamera einzelne Gegenstände auf dem Fließband erkennt und der NJ mitteilt. Der nächste Block mit dem Namen *Objekt Typ* (siehe Abbildung 5.15) gibt an um welchen Objekttyp es sich bei dem aktuellen Objekt handelt. Die nächsten beiden Blöcke *Farbe* sowie *Rotation* werden ebenfalls benötigt, wenn ein neues Objekt erstmalig in der Szene angelegt wird. Diese beschreiben, ihrem Namen entsprechend, die Farbe und die Rotation eines Objektes. Der darauf folgende und vorletzte Block dieses Nachrichtentyps gibt an, ob das Objekt aktuell von dem Roboter gegriffen wird.

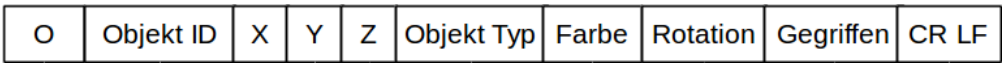


Abbildung 5.15: Nachrichtentyp für Objektdaten.

Objekteigenschaften Nachrichtentyp

Die Objekteigenschaften zu einer V-REP Szene werden durch den Objekteigenschaften Nachrichtentyp beschrieben. Der Aufbau dieses Nachrichtentyps ist in der Abbildung 5.16 dargestellt. Die Objekteigenschaften beginnen mit einem *C*, dass den Nachrichtentyp identifiziert. Darauf folgt die *Objekttyp ID* und schließlich die einzelnen Eigenschaften. Mit

einem senkrechten Strich werden die Daten für eine Objekteigenschaft abgeschlossen. Beginnend mit dem Identifier *C* können andere Objekteigenschaften am Ende hinzugefügt werden bis genug Objekteigenschaften vorhanden sind.

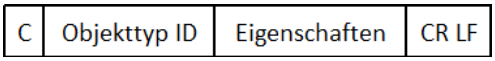


Abbildung 5.16: Nachrichtentyp für Objekteigenschaften.

Neustart Nachrichtentyp

Eine grafische Darstellung des Aufbaus des Neustarten Nachrichtentyps ist in Abbildung 5.17 zu finden. Das Neustarten der Visualisierung beginnt mit dem Identifier *S* im ersten Block.

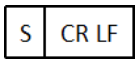


Abbildung 5.17: Nachrichtentyp für das Neustarten der Visualisierung.

Entscheidungsdaten Nachrichtentyp

Eine grafische Darstellung des Aufbaus des Nachrichtentyps für das Entscheidungssingal ist in Abbildung 5.18 zu finden. Das Entscheidungssignal beginnt mit dem Identifier *D* im ersten Block. Im zweiten Block ist die Entscheidung der Lernphase enthalten. Hierbei steht die 1 für die Entscheidungsphase und 2 für die Vetophase. Im nächsten Block steht die *Delta-Roboter ID*. Im vorletzten Block sind die *Objekt IDs* enthalten. Sie werden mit Semikolons getrennt übertragen.

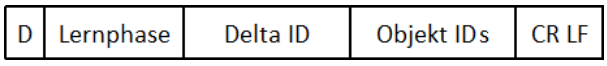


Abbildung 5.18: Nachrichtentyp für Entscheidungsdaten.

5.3.2 V-REP Communicator

Beim V-REP Communicator handelt es sich um eine in Java programmierte Schnittstelle, die den Datenaustausch zwischen NJCom und V-REP ermöglicht (siehe Abbildung 5.19). Im Folgenden werden die Aufgaben des V-REP Communicator vorgestellt.

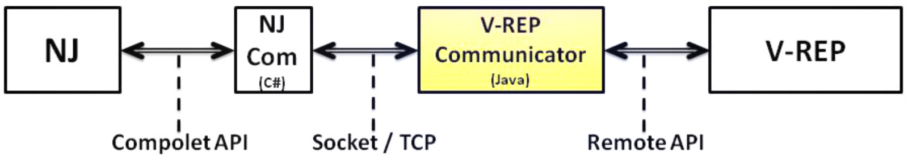


Abbildung 5.19: Teilmodul V-REP Communicator der Kommunikationsschnittstelle.

Aufbau

Das Modul V-REP Communicator unterteilt sich in zwei Komponenten. Dadurch werden die folgenden unterschiedliche Aufgaben realisiert:

- **Network:** Diese Komponente stellt die Netzwerkfunktionalität von V-REP Communicator zur Verfügung. Sie lauscht auf einem konfigurierbaren Port auf eingehende Nachrichten der NJCom. Als Transferprotokoll wird TCP verwendet.
- **Vrepcom:** Die Komponente agiert als Schnittstelle zu V-REP und kapselt die V-REP Remote-API.

Das genaue Zusammenspiel dieser zwei Komponenten und der Ablauf der Verarbeitung einer eingehenden Nachricht beim V-REP Communicator wird in den folgenden Abschnitten näher erläutert. Die Aufteilung der Komponenten findet sich ebenfalls in der Struktur des Quellcodes vom V-REP Communicator wieder.

Kommunikation mit NJCom

Als Teilmodul der Kommunikationsschnittstelle der Visualisierungskomponente (siehe Kapitel 5.2.1) muss der V-REP Communicator neben V-REP auch mit dem Modul NJCom kommunizieren. Hierbei übernimmt es eine passive Rolle in der Kommunikation. Das Modul V-REP Communicator lauscht zu Beginn der Kommunikation auf einem frei wählbaren Port. Alle dort eingehenden Nachrichten, die vom Modul NJCom stammen, werden entsprechend ihres Nachrichtentyps verarbeitet. Der V-REP Communicator selbst sendet keine Daten an das Modul NJCom.

Verarbeitungsprozess vom V-REP Communicator

Beim Start der Anwendung V-REP Communicator werden alle zwei Komponenten, welche im vorherigen Abschnitt vorgestellt worden sind, gestartet. Die Komponente V-REP Communicator verbindet sich zu Beginn mit der aktuellen Szene, die in dem Visualisierungsprogramm V-REP aktuell ausgeführt wird. Des Weiteren bindet sich die Netzwerkkomponente beim Start vom V-REP Communicator an einen Port, um dort auf eingehende Nachrichten des Moduls NJCom zu lauschen. Beim Eintreffen einer Nachricht an dem entsprechenden

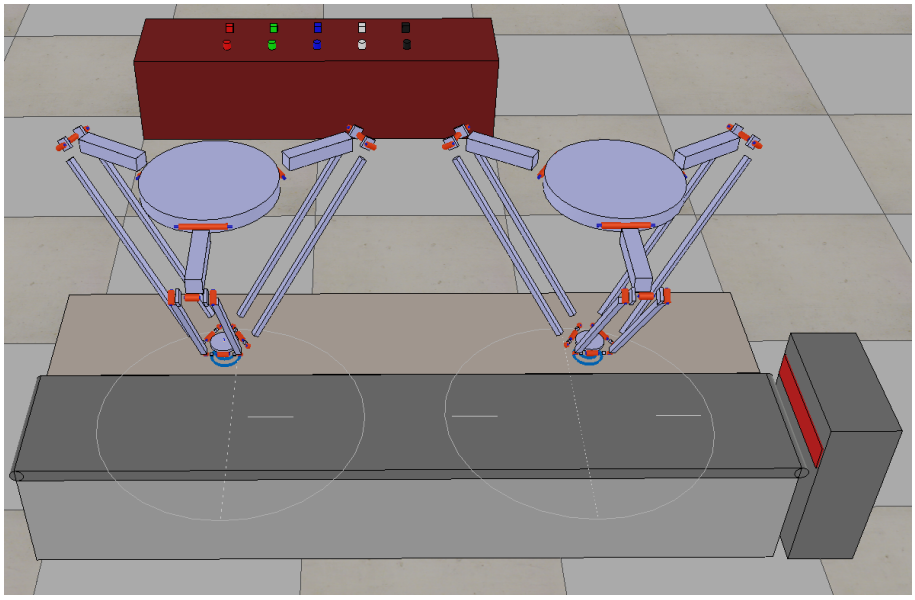


Abbildung 5.20: Aktuelle Szene.

Port wird diese von der Netzwerkkomponente geparst. Beim Parsevorgang wird unter anderem der Typ der vorliegenden Nachricht festgestellt. Nach einem erfolgreichen Parsen der Nachricht wird entsprechend dem festgestelltem Nachrichtentyp, die Nachricht verarbeitet und weiter verschickt.

V-REP Java API

Um Signale an V-REP zu übertragen wird die V-REP Remote-API genutzt. Sobald V-REP gestartet wird, beginnt ein Script auf V-REP Seite auf eine Verbindung zu warten. Das Java Programm baut die Verbindung zu V-REP auf, sobald es gestartet wird. Aus diesem Grund muss die V-REP Szene vor dem Java Programm gestartet werden. Um Daten von Java an V-REP zu übertragen werden Signale verwendet. Ein Signal ist eine Variable in V-REP, die durch die Remote-API gesetzt werden kann.

5.3.3 Aufbau der V-REP Szene

Abbildung 5.20 zeigt die aktuelle Szene wie sie in V-REP realisiert ist. Im vorderen Bereich befinden sich das Fließband, die zwei Delta-Roboter und eine Ablagefläche auf der die Objekte abgelegt werden. Im hinteren Teil der Szene liegen die Objektvorlagen, welche während der Visualisierung kopiert und an den richtigen Ort verschoben werden. Nachfolgend soll kurz erklärt werden, wie die einzelnen Objekte in V-REP erstellt worden sind:

Zunächst wird der hintere Teil der Szene betrachtet. Die Objekte sind als Würfel und Zylinder (primitive *Shapes*) realisiert worden, um die Objekte in der Form unterscheidbar zu machen. Als zweites optisches Merkmal wurden die Objekten in unterschiedlichen

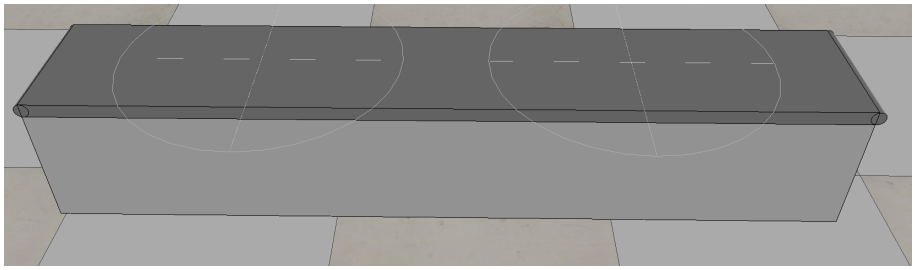


Abbildung 5.21: Das Fließband.

Farben erstellt (Rot, Grün, Blau, Schwarz und Weiß). Durch diese Methodik ist es auch in kürzester Zeit möglich, beliebig viele neue Objekte zu erzeugen. Weitere Merkmale, welche nicht durch die Optik repräsentiert werden können, werden, nachdem ein Objekt in der laufenden Visualisierung ausgewählt wurde, am unteren Bildschirmrand angezeigt. Der vordere Bereich der Szene wird Hauptsächlich durch das Fließband sowie die beiden Delta-Roboter bestimmt. Auf diese Elemente soll nachfolgend genauer eingegangen werden.

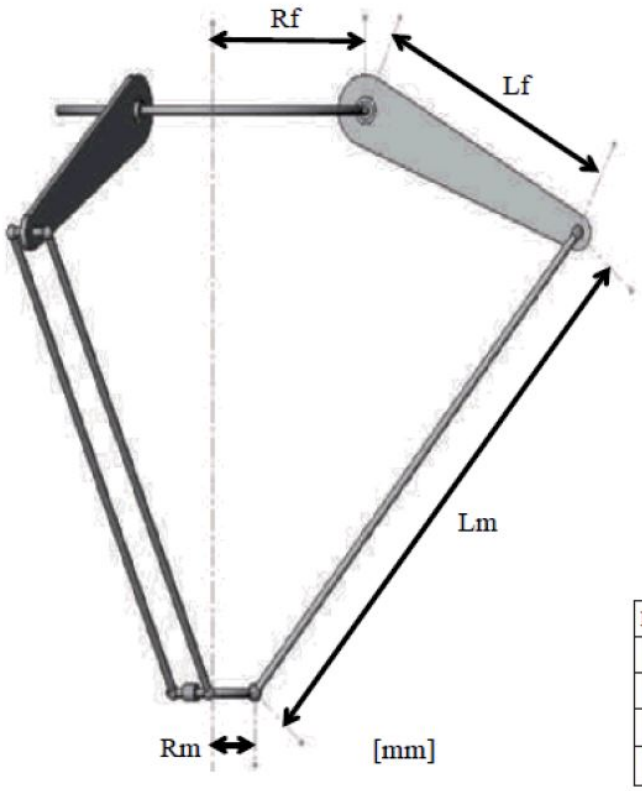
Fließbandsteuerung

Das Fließband in Abbildung 5.21 wurde aus mehreren einfachen Shapes zusammengesetzt, da die Objekte nicht durch das Fließband bewegt werden, sondern durch ankommende Signale. Es besitzt eine Länge von 3000 mm und ist 500 mm breit. Über dem Fließband befinden sich zwei Kreise, welche jeweils in vier Teile unterteilt sind. Diese Kreise befinden sich genau unter den beiden Delta-Roboter und beschreiben ihren Arbeitsbereich, welcher nochmal in die folgenden Bereiche eingeteilt wurde: oberer vorderer, oberer hinterer, unterer vorderer und unterer hinterer Arbeitsbereich. Diese Kreise dienen nur zur Illustration der Arbeitsbereiche auf dem Fließband.

Für die Objekterzeugung wurde ein Pseudoobjekt mit dem Namen *partProducer* erstellt und dem Fließband untergeordnet. In diesem Luaskript werden die neuen Objekte nach den Angaben, die von der Remote-API kommen, erzeugt. Zuerst wird das String Signal wieder in verschiedene Variablen aufgesplittet. Danach wird getestet, ob das Objekt bereits vorhanden ist. Wenn dem nicht so ist, wird je nach *CharacteristicSetId* eines der auf dem Tisch liegenden Objekte geklont und anschließend an die richtige Position mit der richtigen Drehung gesetzt.

Aufbau des Delta-Roboters

Während die Objektvorlagen und das Fließband als primitive Shapes erstellt worden sind, musste die Vorlage des Delta-Roboter, wie in Abbildung 5.23 dargestellt aus mehreren primitiven Shapes erstellt werden. Die von OMRON zur Verfügung gestellten Maße – wie in Abbildung 5.22 dargestellt – wurden in V-REP nachgebildet. Dazu wurden die einzelnen Komponenten in V-REP als Shapes erzeugt und in einer Baumstruktur, wie in



Kinematics parameters	
Rf	225 mm
Rm	62,5 mm
Lf	340 mm
Lm	840 mm

Rf: Distance (radius) from the center of the fixed frame to the motor axis [mm]
Rm: Distance (radius) from the center of the moving frame to the connection point of link 2 [mm]
Lf: Length of the link 1 [mm]
Lm: Length of the link 2 [mm]

Abbildung 5.22: Kinematic Parameters von Omron.

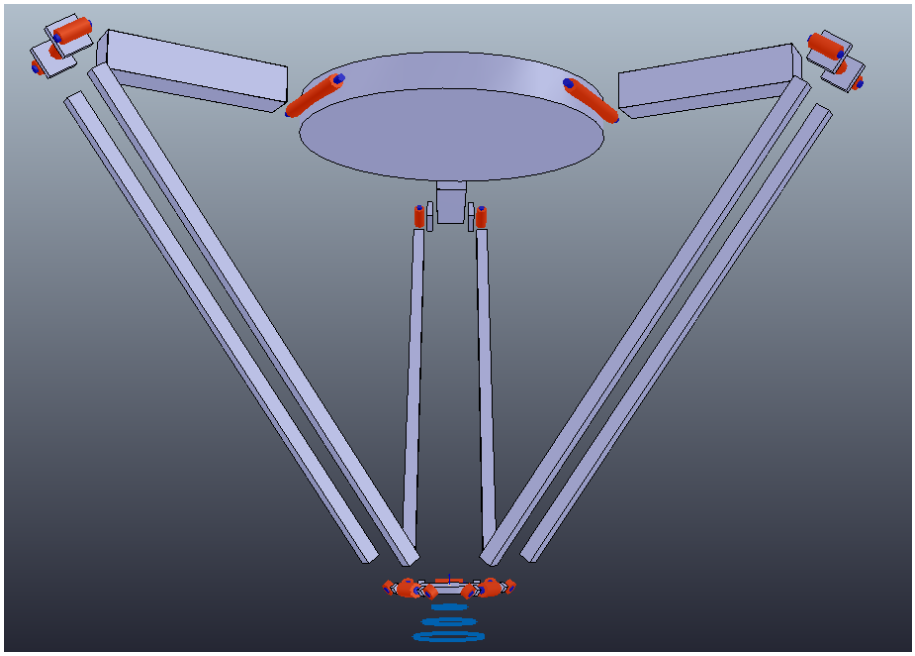


Abbildung 5.23: Der Delta-Roboter.

Abbildung 5.24 dargestellt, angeordnet. Dabei sind die Kinderelemente eines Objekts mit dem Objekt selbst physisch verbunden. Das bedeutet, dass wenn ein Objekt bewegt wird, sich auch seine Kindobjekte bewegen.

Um sicherzustellen, dass sich der TCP immer an der Stelle befindet, an der er sich laut NJ auch befinden soll, werden die Arme des Delta-Roboters und der TCP unabhängig voneinander bewegt. Dadurch kann es bei der Visualisierung allerdings dazu kommen, dass sich der TCP an einer anderen Stelle befindet, als die Arme es vermuten lassen. Am TCP befindet sich keine Greifkonstruktion, weil die Bewegung der Objekte von der NJ berechnet werden.

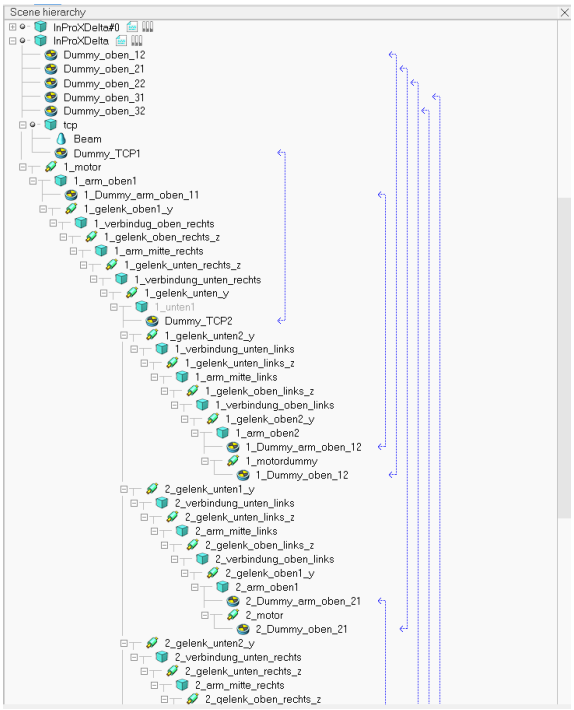


Abbildung 5.24: Baumstruktur in V-REP.

Kapitel 6

Lernkomponente

Bisher mussten die Delta-Roboter direkt über ST-Code programmiert werden. Aus diesem Grund sind nur explizit geschulte Entwickler in der Lage diese Programmierung vorzunehmen. Die Vorgaben, welche Gegenstände von den Robotern gegriffen und verpackt werden sollen, sowie die Beschreibung der zu greifenden Gegenstände obliegt dem Domainexperten. Zwischen dem Entwickler, welcher ausschließlich die technische Sicht der Dinge beherrscht und dem Domainexperten, welcher über Domainwissen von Verpackungsanlagen verfügt, erstreckt sich eine semantische Lücke, welche zu Mehraufwand innerhalb des Entwicklungsprozess führt, da eine ständige Übersetzung zwischen den verschiedenen Beschreibungen erfolgen muss.

Die 3PL (Pick and Place Packaging Language) wurde entwickelt, um diese semantische Lücke zu schließen, indem der Domainexperte die Möglichkeit erhält seine Vorstellung eines Verpackungsverfahrens im Rahmen einer graphischen DSL zu modellieren. Die grafische DSL bietet den Vorteil, dass sie keine technischen Vorkenntnisse erfordert, wie eine textuelle Programmiersprache, sondern auf die Intuitivität der Nutzer während der Modellierung abzielt.

In diesem Kapitel wird der nächste Schritt zur intuitiven Modellierung des Verhaltens einer Verpackungsanlage vorgestellt. Die bestehende 3PL wird um eine Lernkomponente erweitert, welche es dem Domainexperten ermöglicht seine Vorstellungen nicht modellieren zu müssen, sondern es anhand von Beispielen zu zeigen. Das „Programming by Example“ entspricht dem Erlernen eines bestimmten Verhaltens durch Nachahmung und Generalisierung. In dem konkreten Szenario der Verpackungsanlagen erfolgen die Beispiele als eine Fließbandbelegung von zuvor definierten oder zufällig generierten Gegenständen, welche innerhalb einer Visualisierung vom Domainexperten verpackt werden. Aus dem demonstrierten Verhalten und dem daraus resultierenden erlernten Wissen wird automatisch ein 3PL Programm erstellt. Wie in den vorherigen Kapiteln bereits vorgestellt wurde, wird aus diesem 3PL Programm ST-Code generiert, welcher im Anschluss auf die NJ übertra-

gen und ausgeführt werden kann. Abbildung 6.1 markiert die Komponenten die innerhalb dieses Kapitels beschrieben werden.

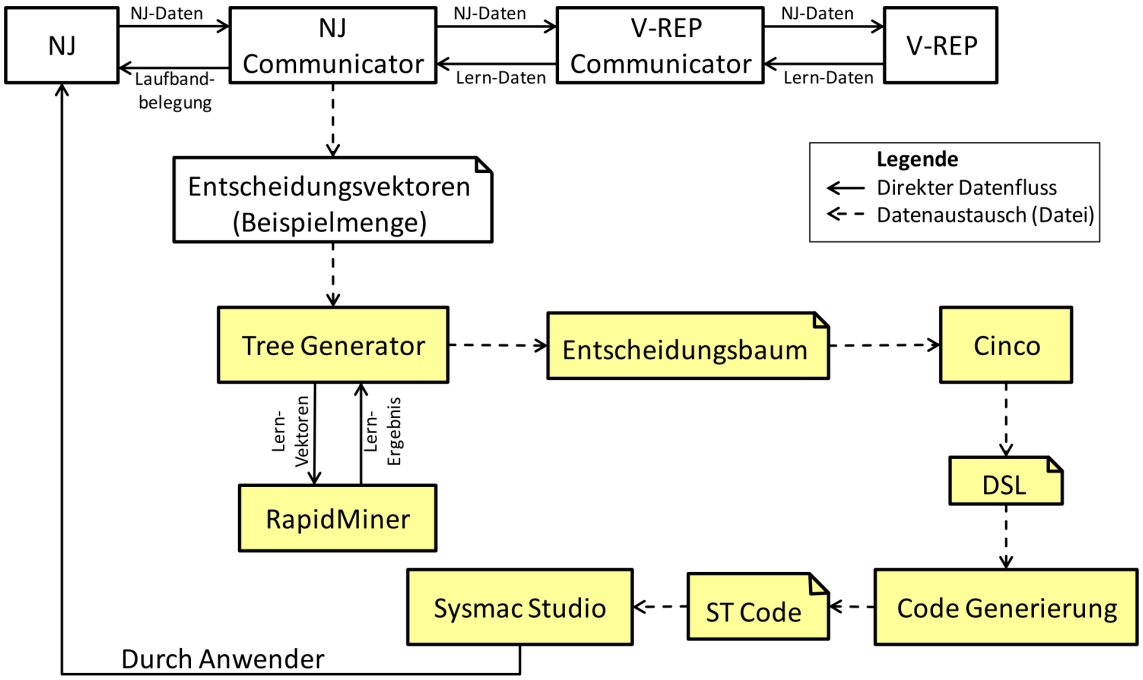


Abbildung 6.1: Kapitel 6, Komponenten in der Gesamtübersicht.

Zunächst wird das Lernverfahren vorgestellt, welches genutzt werden soll, um das „Programming by Example“ zu realisieren. Es werden aktive und passive Lernverfahren verglichen im Bezug auf das Verhältnis zwischen dem Nutzen eines Verfahrens und dem Aufwand, welcher erbracht werden muss, um das gewollte 3PL Programm zu erhalten.

Im Anschluss werden die verschiedenen Attribute aufgezeigt und erläutert, welche dem Lernverfahren helfen sollen die Intention des Domainexperten zu verstehen, indem die zur Entscheidung benötigten Informationen aufgezeichnet werden. Aus diesen Informationen wird das Verhalten in Form eines Entscheidungsbaums induziert, welcher für einen Gegenstand abhängig von seinen Attributen entscheidet, ob er gegriffen werden soll oder nicht.

6.1 Das Lernverfahren

Die Wahl eines geeigneten Lernverfahrens schafft die Grundlage zur intuitiven Nutzung und muss dem Anspruch genügen, die bisherige Form der DSL Modellierung maßgeblich zu verbessern. Zunächst muss geprüft werden, ob ein aktives oder ein passives Lernverfahren zum Einsatz kommen soll. Ein aktives Lernverfahren zeichnet sich durch seine häufige Interaktion mit dem Domainexperten aus. Die Antworten würden genutzt, um alle relevanten Fließbandbelegungen zu erzeugen, welche zur Bestimmung des 3PL Programms notwendig sind. Dieses Vorgehen erfordert ein hohes Maß an Mitwirkung des Domainexperten, sodass

der Aufwand die Fragen des aktiven Lernverfahrens zu beantworten eine mögliche direkte Modellierung überwiegt. Aus diesem Grund eignet sich ein passives Lernverfahren eher zu Realisierung des hier angestrebten „Programming by Example“. Passive Lernverfahren eignen sich deutlich besser, da auch aus unvollständigen Daten, welche aus einer zufälligen Bandbelegung stammen, gelernt werden kann, weswegen der Domainexperte nicht explizit nach jeder Möglichkeit gefragt werden muss. Beim passiven lernen hat der Lernende keinen Einfluss auf die ihm präsentierte Information und kann ausschließlich beobachten.

Das zu lernende Verhalten lässt sich zu einer binären Entscheidung generalisieren, welche für einen Gegenstand anhand seiner Attribute entscheiden muss, ob dieser Gegenstand gegriffen werden soll oder nicht. Auf Grund dieser Abstraktion lässt sich das benötigte Lernverfahren durch die Verwendung von *Entscheidungsbäumen* darstellen. Entscheidungsbäume sind geordnete, gerichtete Bäume, die der Darstellung von Entscheidungsregeln dienen. Die grafische Darstellung als Baumdiagramm veranschaulicht hierarchisch aufeinanderfolgende Entscheidungen. Sie haben eine Bedeutung in zahlreichen Bereichen, in denen automatisch klassifiziert wird oder aus Erfahrungswissen formale Regeln hergeleitet oder dargestellt werden. Die Klassifikationen befinden sich in den Blättern des Baumes. Die Knoten des Baumes enthalten die Entscheidungsattribute und die Kanten sind mit den zugehörigen Attributswerten beschriftet. [4]

Im Bezug auf diese Arbeit wird der Entscheidungsbaum von RapidMiner mit Hilfe von *Example-Sets* erzeugt. Die Example-Sets beschreiben eine Menge von Beispielen, welche von einem Domainexperten angegeben wurden. Jedes dieser Beispiele wird durch den gegriffenen oder nicht gegriffenen Gegenstand und einer Liste von Attributen dargestellt, welche sowohl die Merkmale des Gegenstands selbst sowie weitere Informationen im Bezug auf umliegende Gegenstände und seine Position beinhaltet. Der gegriffene oder nicht gegriffene Gegenstand wird im Folgenden als Referenz-Gegenstand bezeichnet. RapidMiner erstellt aus den gegebenen Example-Sets einen binären Entscheidungsbaum, welcher anhand von bestimmten Attributen der Gegenstände entscheidet, ob ein Gegenstand gegriffen werden soll oder nicht. Mit Hilfe von RapidMiner kann somit anhand von Beispielen in Form von Entscheidungen, gesammelt in Example-Sets, ein Entscheidungsbaum induziert werden. Dieser Entscheidungsbaum kann im Anschluss in 3PL nachgebildet werden, sodass der Delta-Roboter das anhand von Beispielen gezeigte Verhalten nachahmen kann.

6.2 Die Entscheidungsattribute

Im Folgenden wird erläutert aus welchen Attributen die Beispiele bestehen, welche zum Erlernen eines Entscheidungsbaumes verwendet werden. Alle Attribute sind binär, sodass nur unterschieden werden kann, ob ein Gegenstand ein durch ein Attribut dargestelltes Merkmal erfüllt oder nicht. Verwendet werden für alle Attribute die Werte 0 oder 1, was falsch und richtig entspricht. Also kann ein Attribut entweder wahr oder falsch sein. Die

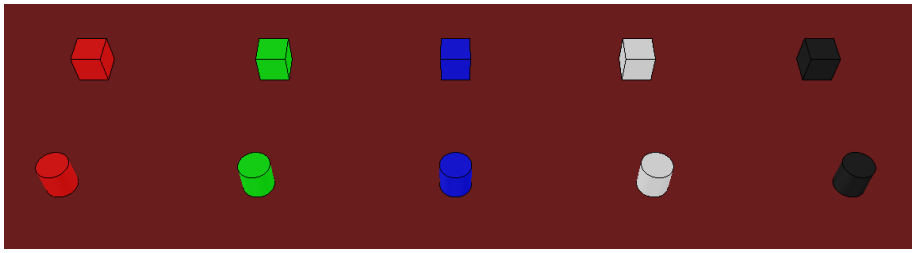


Abbildung 6.2: Vorhandene Gegenstände für das Fließband in der V-REP Szene.

Wahl der Attribute nach denen ein Beispiel klassifiziert werden kann, erschließt sich aus den vom Domainexperten wahrnehmbaren Unterschiede zwischen den Gegenständen auf dem Fließband. Grob werden drei Gruppen von Attributen unterschieden: Die erste Gruppe von Attributen bezieht sich auf Eigenschaften die sich direkt auf einen Gegenstand beziehen, wie zum Beispiel die Farbe und die Form. Die zweite Gruppe von Attributen bezieht sich auf die Umgebung und Lage eines Gegenstandes auf dem Fließband. Mit dieser Gruppe von Attributen ist es möglich die Umgebung und Lage eines Gegenstands auf dem Fließband zu unterscheiden. Die letzte Gruppe von Attributen ist das *Ziel-Attribut*, welches durch den erlernten Entscheidungsbaum vorhergesagt werden soll. Das Ziel-Attribut soll aussagen, ob ein Delta-Roboter ein Gegenstand greifen soll (richtig) oder nicht (falsch).

Nachfolgend werden die verwendeten Ausprägungen der Gruppen von Attributen beschrieben. Zu der ersten Gruppe zählen folgende Attribute, welche die Form und die Farbe eines Gegenstands beschreiben:

- black
- white
- blue
- green
- red
- round
- angular

Die Abbildung 6.2 visualisiert die Attribute anhand der Gegenständen, welche in der V-REP Szene Verwendung finden. So erfüllt z.B. der Gegenstand links unten die Attribute *red* und *round*. Die zweite Gruppe von Attributen, welche die Position des Referenz-Gegenstands im Arbeitsbereich eines Delta-Roboters, sowie die Anzahl der Gegenstände im näheren Umkreis beinhaltet des Referenz-Gegenstands, wird durch folgende Menge beschrieben:

- **topC:** obere Delta-Roboter Arbeitsbereich-Hälfte

- **frontWB:** vordere Hälfte des Arbeitsbereichs eines Delta-Roboters
- **cluster1:** mindestens ein Gegenstand im Umkreis
- **cluster2:** mindestens zwei Gegenstände im Umkreis
- **cluster3:** mindestens drei Gegenstände im Umkreis
- **cluster4:** mindestens vier Gegenstände im Umkreis

Das Attribut *topC* ist wahr, wenn sich der Referenz-Gegenstand in der oberen Hälfte des Arbeitsbereichs befindet (siehe Abbildung 6.3). Befindet sich der Referenz-Gegenstand in der unteren Hälfte des Fließbands ist das Attribut nicht erfüllt (siehe Abbildung 6.4). In

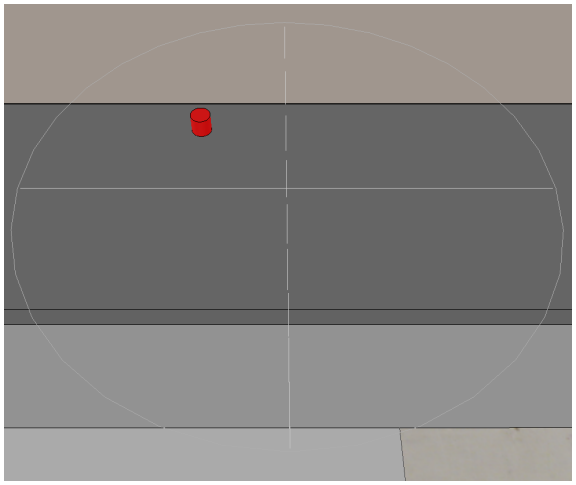


Abbildung 6.3: Ein Gegenstand befindet sich in der oberen Fließbandhälfte.

Verbindung mit dem Attribut *frontWB*, welches angibt, ob sich der Referenz-Gegenstand innerhalb der vorderen Hälfte des Arbeitsbereichs entgegen der Laufrichtung des Fließbands befindet (s. Abbildung 6.5) und dem Attribut *topC* wird der Arbeitsbereich eines Delta-Roboters in vier Viertel unterteilt. Damit ist es möglich zu unterscheiden ob sich der Referenz-Gegenstand in der vorderen oder hinteren Hälfte des Arbeitsbereichs eines Delta-Roboters befindet. Die Attribute *cluster1* bis *cluster4* beschreiben wie viele Gegenstände sich im näheren Umkreis um den Referenz-Gegenstand befinden. Der optimale Radius, welcher der Berechnung des Umkreises zugrunde liegt muss evaluiert werden und wird zunächst initial mit $200mm$ festgelegt. Das Attribut cluster1 ist erfüllt, wenn sich in einem Radius $200mm$ um den Referenz-Gegenstand mindestens ein weiterer Gegenstand befindet (s. Abbildung 6.7 unten links). Analog dazu ist das Attribut cluster2 erfüllt, wenn sich in einem Radius von 200mm um den Referenz-Gegenstand mindestens zwei weitere Gegenstände befinden (s. Abbildung 6.7 oben links). Aber auch ein cluster1 Attribut ist erfüllt, wenn ein cluster2 Attribut erfüllt ist. Das liegt daran, dass ein cluster1 Attribut erfüllt ist, sobald mindestens ein weiterer Gegenstand in einem Radius von $200mm$ um

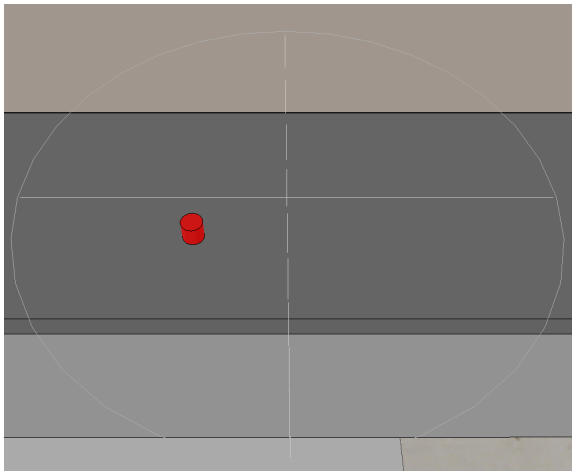


Abbildung 6.4: Ein Gegenstand befindet sich in der unteren Fließbandhälfte.

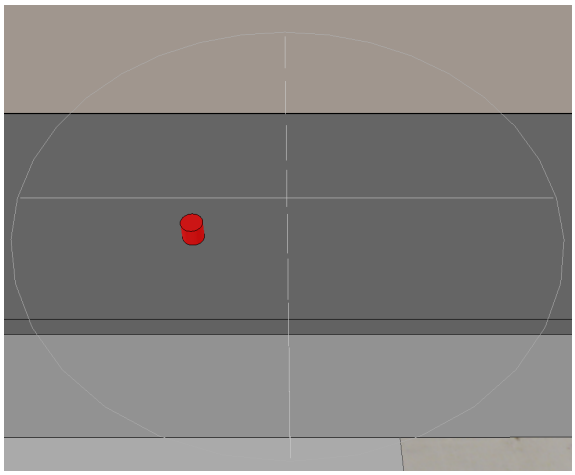


Abbildung 6.5: Ein Gegenstand befindet sich in der vorderen Hälfte des Arbeitsbereichs eines Delta-Roboters.

den Referenz-Gegenstand liegt und das erfüllt auch ein cluster2 Attribut. Die anderen Attribute cluster3 und cluster4 sind analog definiert. Mit Hilfe dieser Gruppe von Attributen ist es möglich sowohl die Relevanz der Position als auch die der räumlichen Häufung zu klassifizieren.

Zur Steigerung Genauigkeit können im Rahmen weiterer Optimierungen zusätzliche Attribute in die einzelnen Gruppen aufgenommen werden, um zum Beispiel den Arbeitsbereich in mehrere kleinere Teile zu differenzieren oder unterschiedliche Radien für verschiedene Cluster zu definieren. Die Relevanz der einzelnen Attribute muss dabei immer im Nachhinein evaluiert werden.

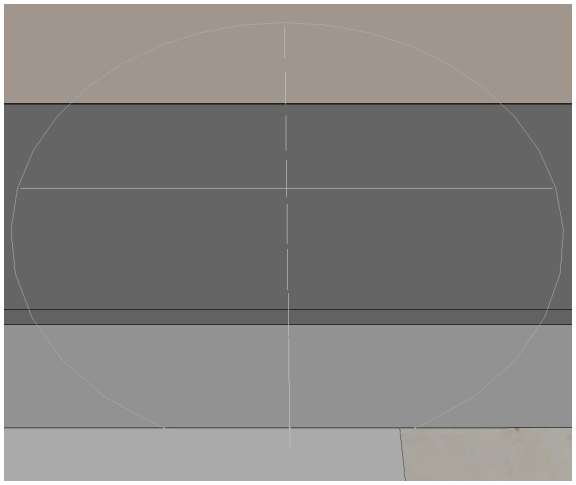


Abbildung 6.6: Aufteilung des Arbeitsbereichs eines Delta-Roboters.

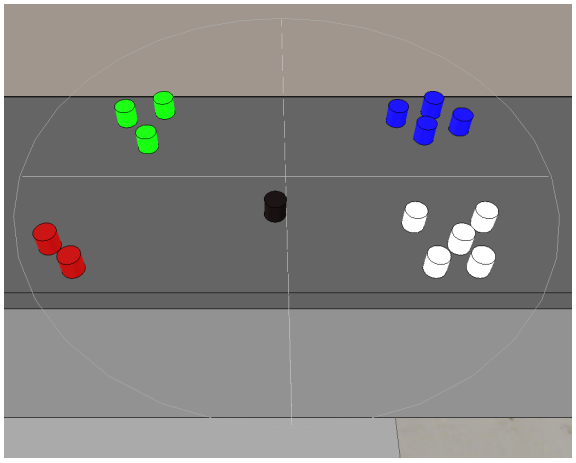


Abbildung 6.7: Darstellung von Clustern: cluster1 (unten links), cluster2 (oben links), cluster3 (oben rechts), cluster4 (unten rechts), kein Cluster (mitte).

6.3 Das Lernkonzept

In diesem Abschnitt wird das Lernkonzept im Detail vorgestellt, welches die Grundlage der Lernkomponente bildet. Das Lernkonzept beschreibt die verschieden Abläufe die notwendig sind, um ein Verhalten anhand von Beispielen zu induzieren. Die technische Realisierung des Lernkonzepts wird in dem darauf folgenden Abschnitt 6.4 beschrieben. Das Lernkonzept besteht aus drei Phasen, die nacheinander durchgeführt werden. Jede einzelne Phase besteht wiederum aus einer vorgeschriebenen Reihenfolge verschiedener Aktivitäten. Die erste Phase ist die Entscheidungsphase.Die zweite Phase, welche iterativ sowohl nach der ersten als auch der dritten Phase durchgeführt werden kann ist die *Generierungsphase*. Die Generierungsphase induziert die Example-Sets zu einem Entscheidungsbaum, welcher in ein 3PL Programm eingebunden wird. Nach mindestens einer Generierungsphase folgt die Durchführung die dritte Phase, der Vetophase.Sowohl in der Vetophase als auch in

der Entscheidungsphase können Beispiele erzeugt werden, welche in einem gemeinsamen Example-Set gehalten werden, um ein immer genaueres Abbild des gewünschten Verhalten zu erhalten. Ein Durchlauf aller drei Phasen wird als Lernroundtrip bezeichnet. Die Abbildung 6.8 stellt einen Lernroundtrip bildlich dar und zeigt zudem die drei Phasen mit ihren einzelnen Aktivitäten. Im Folgenden werden die einzelnen Phasen näher beschrieben und ihre einzelnen Aktivitäten vorgestellt.

6.3.1 Die Entscheidungsphase

Die Entscheidungsphase ist die erste Phase des Lernkonzepts und auch die erste Phase die innerhalb eines Lernroundtrips ausgeführt wird. In dieser Phase soll der Domainexperte der Lernkomponente den Delta-Roboter ein gewünschtes Verhalten, also das gezielte Aufnehmen von spezifischen Gegenständen, aufzeigen. Die Entscheidungsphase besteht aus drei unterschiedlichen Aktivitäten, welche in der folgende Reihenfolge durchgeführt werden (siehe Abbildung 6.8):

1. Bandbelegung definieren und generieren,
2. die Bewegung der Gegenstände auf dem Fließband aufzeichnen und
3. die Aufzeichnung in V-REP abspielen und Entscheidungen treffen, welche Gegenstände gegriffen werden sollen.

Die Entscheidungsphase startet mit der Aktivität *Bandbelegung generieren*, in der eine zuvor definierte Bandbelegung generiert wird. Der Benutzer kann mit Hilfe von einer Menge von Parameter den Generierungsprozess steuern und so beispielsweise festlegen wie viele Gegenstände sich über das Fließband bewegen sollen, welche Mindestabstände zwischen den Gegenständen vorhanden sind und welche Attribute die Gegenstände erfüllen sollen. Ist eine Bandbelegung generiert worden, geht die Entscheidungsphase in die zweite Aktivität über. In der Aktivität *Gegenstands-Bewegung aufzeichnen* wird die Bewegung der zuvor definierten und generierten Gegenstände aufgezeichnet. Hierbei werden die Positionen der einzelnen Gegenstände von Beginn bis Ende des Fließbands zu jedem Zeitpunkt berechnet und aufgezeichnet.

In der dritten Aktivität *In V-REP abspielen und Entscheidungen treffen* kann der Domainexperte entscheiden, welche Gegenstände gegriffen werden sollen. Diese Entscheidung kann getroffen werden solange sich ein Gegenstand innerhalb des Arbeitsbereichs eines Delta-Roboters befindet. Der Zeitpunkt zudem der Domainexperte entscheiden muss, ob er einen Gegenstand von einem Delta-Roboter greifen und verpacken lassen möchte wird im Folgenden als *Entscheidungspunkt* bezeichnet. Die Entscheidungspunkte werden in V-REP durch die Pausierung des Ablaufs und das Erscheinen eines Fensters symbolisiert. Der Domainexperte kann an dieser Stelle einen der Gegenstände innerhalb des Arbeitsbereichs auswählen, um ihn zu greifen oder den Entscheidungspunkt ignorieren, sodass kein

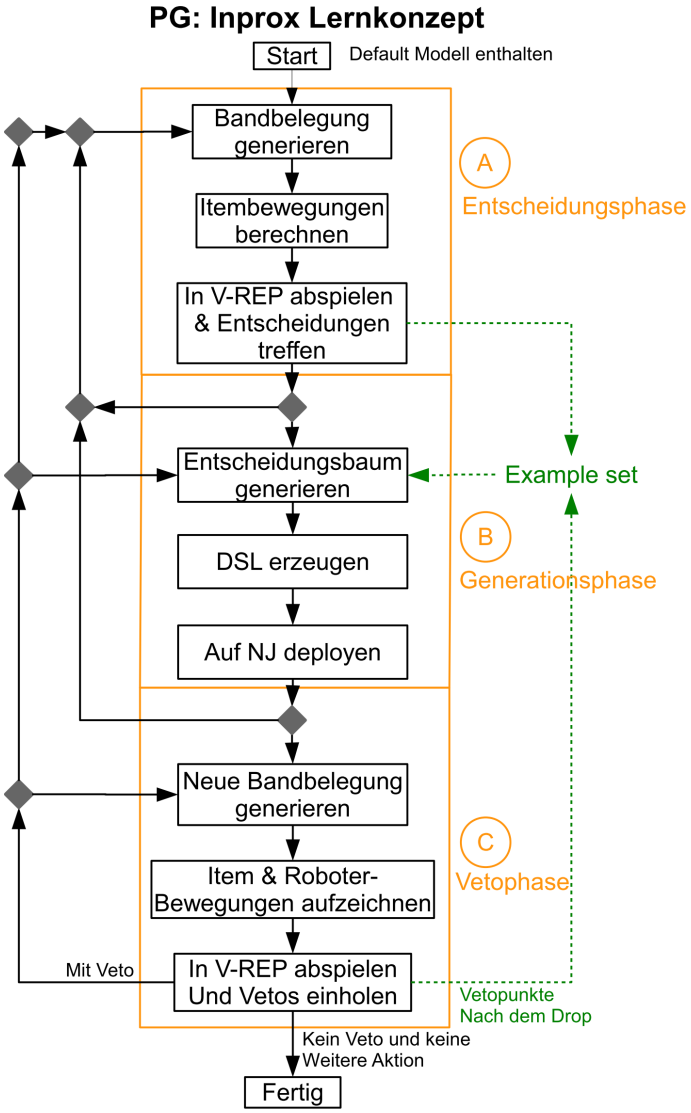


Abbildung 6.8: Darstellung des Lernkonzepts der Lernkomponente und seiner zwei Phasen mit den jeweiligen Aktivitäten.

Gegenstand gegriffen wird. In diesem Fall werden für die sich im Arbeitsbereich befindlichen Gegenstände für die nächsten 10 Sekunden keine Entscheidungspunkte berechnet, um nicht zu jedem Zeitpunkt entscheiden zu müssen. Die Zeitspanne wird aufgehoben, sobald ein neuer Gegenstand den Arbeitsbereich erreicht. In diesem Fall können wieder alle sich im Arbeitsbereich befindlichen Gegenstände ausgewählt werden, um zum Beispiel eine gegebenenfalls beabsichtigte Reihenfolge der gegriffenen Gegenstände zu erfassen, in welcher gewartet werden muss, dass ein bestimmter Gegenstand den Arbeitsbereich betritt, bis die restlichen Gegenstände im Arbeitsbereich gegriffen werden sollen.

Jede getroffene Entscheidung zu einem Entscheidungspunkt erzeugt einen Menge an Beispielen für ein Example-Set. Die erzeugten Beispiele enthalten die Attribute des Referenz-

Gegenstand und aller Gegenstände im Cluster im Radius von $200mm$ um den Referenz-Gegenstand 6.2. Die Erzeugung der Beispiele anhand zweier Fälle unterschieden:

- Der Domainexperte entscheidet sich einen Gegenstand aus dem Arbeitsbereich zu greifen und zu verpacken. In diesem Fall wird für den Referenz-Gegenstand ein positives Beispiel erzeugt, bei dem das Ziel-Attribut als wahr gesetzt ist. Für alle weiteren Gegenstände im Arbeitsbereich werden negative Beispiele erzeugt mit dem nicht gesetzten Ziel-Attribut, da der Domainexperte diese Gegenstände vorsätzlich nicht greifen wollte.

- Der Domainexperte entschiedet sich keinen der sich im Arbeitsbereich befindlichen Gegenstände zu greifen. In diesem Fall werden für alle Gegenstände im Arbeitsbereich negative Beispiele erzeugt.

Mit Abschluss der Entscheidungsphase hat der Benutzer jedem Delta-Roboter ein gewisses Muster an aufzunehmenden Gegenständen in Form von einem Example-Set gegeben. Dieses vom Domainexperten intendierte Muster soll von dem jeweiligen Delta-Roboter gelernt und dann in der zweiten Phase des Lernkonzept, der Vetophase wiederholt werden, wobei dem Domainexperten die Möglichkeit eingeräumt wird, die vom Delta-Roboter getroffenen Entscheidungen, zu bewerten.

6.3.2 Generierungsphase

Die Generierungsphase kann sowohl nach der Entscheidungs- als auch nach der Vetophase ausgeführt werden, sodass beliebige Wiederholungen ermöglicht werden. In dieser Phase werden die zuvor erlernten Beispiele innerhalb der Example-Sets zu einem Entscheidungsbaum induziert, welcher im Anschluss in ein 3PL Programm eingebettet wird. Der Ablauf erfolgt in drei Aktivitäten, welche als erstes den *Entscheidungsbaum generieren*. Die zuvor im Lernkonzepts gesammelten Example-Sets der einzelnen Delta-Roboter werden an dieser Stelle verarbeitet. Für jeden Delta-Roboter werden in der Aktivität *Entscheidungsbaum generieren* mit Hilfe von RapidMiner Entscheidungsbäume induziert. Diese Entscheidungsbäume beschreiben die Entscheidungsfindungsprozesse für die Delta-Roboters für einen Gegenstand den sie greifen könnten.

Die induzierten Entscheidungsbäume werden aus RapidMiner zur nächsten Aktivität *DSL erzeugen* portiert. Während dieser Aktivität werden die aus RapidMiner exportierten Entscheidungsbäume in Knoten und Kanten der 3PL Sprache transformiert und in dem Iterator-Container der Delta-Roboters eingebettet, sodass das durch den Entscheidungsbaum repräsentierte Verhalten in der 3PL Sprache nachgebildet wird.

Nachdem die Entscheidungsbäume in der 3PL Sprache abgebildet sind, muss das Modell mit Hilfe der CodeGen in ST-Code transformiert und auf die NJ übertragen werden, um nachfolgende Vetophasen zu ermöglichen.

6.3.3 Vetophase

Die Vetophase ist die dritte Phase des Lernkonzepts. Diese Phase wird innerhalb eines Lernroundtrips nach mindestens einer Einscheidungsphase und einer Generierungsphase ausgeführt. In dieser Phase soll der Domainexperte innerhalb der Lernkomponente das Verhalten der Delta-Roboter überprüfen und bei fälschlichen Greifen eines Gegenstands ein Veto einlegen. Dafür besteht die Vetophase aus drei unterschiedlichen Aktivitäten, welche in der folgende Reihenfolge durchgeführt werden:

1. Neue Bandbelegung definieren und generieren,
2. Gegenstands- und Roboterbewegungen von der NJ aufzeichnen und
3. die Aufzeichnung in V-REP abspielen und Vetos bei Fehlverhalten einholen.

Die Vetophase beginnt mit dem Schritt *Neue Bandbelegung generieren*. Dabei wird eine neue zufällige Bandbelegung abhängig von definierten Parametern, wie in der Entscheidungsphase (siehe 6.3.1) generiert, anhand derer, das in der vorherigen Entscheidungsphase erlernte Verhalten vom Anwender überprüft werden soll. Diese Bandbelegung wird mittels NJCom auf die NJ übertragen, woraufhin die NJ gestartet wird. Die Bewegungen der Gegenstände auf dem Fließband sowie die Bewegungen und Aktionen der einzelnen Delta-Roboters werden aufgezeichnet. Diese Aufzeichnung, welche innerhalb der *Item und Roboterbewegung aufgezeichnet* stattfindet, enthält alle Informationen, welcher von der NJ zur Verfügung gestellt werden. Zuletzt wird im Rahmen der Vetophase die Aufzeichnung mit Hilfe von V-REP abgespielt. An dieser Stelle beginnt der letzte Schritt der Vetophase: *In V-REP abspielen und Vetos einholen*. Der Domainexperte kann wie schon zuvor in der Entscheidungsphase über das erscheinende Fenster erkennen, dass die Lernkomponente eine Interaktion fordert. Die Zeitpunkte zu denen der Domainexperte interagieren muss werden, analog zu der zuvor beschriebenen Entscheidungspunkten, als *Vetopunkte* bezeichnet. Diese Vetopunkte werden durch den Abschluss einer Verpackungsaktion eines Delta-Roboters ausgelöst und erzeugen genau ein Beispiel für das Example-Set. Das erzeugte Beispiel enthält ausschließlich Informationen über den Referenz-Gegenstand, der gegriffen und verpackt wurde. Der Domainexperte kann an dieser Stelle entscheiden, ob die gezeigte Aktion dem gewünschten Verhalten entspricht. Es wird zwischen zwei Möglichkeiten unterschieden:

- Die gezeigte Aktion entspricht dem gewünschten Verhalten. Ein positives Beispiel wird erzeugt.
- Die vom Delta-Roboter durchgeführte Aktion entspricht nicht dem gewünschten Verhalten. Es wird ein negativ Beispiel für den Referenz-Gegenstand erzeugt.

Die erzeugten Beispiele sollen die bereits vorhandene Menge von Beispielen ergänzen und bieten dem Domainexperten die Möglichkeit gezielt Einfluss auf das bisher gelernte zu

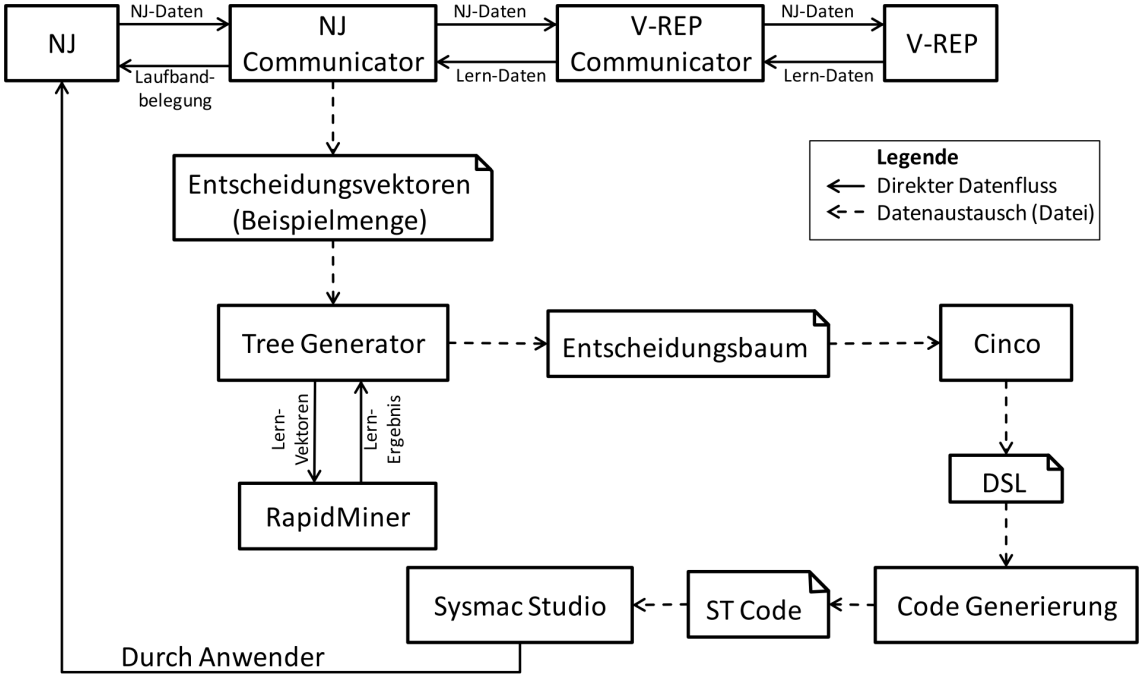


Abbildung 6.9: Technisches Konzept für die Realisierung der Lernkomponente.

nehmen, indem bestimmte Situationen bewertet werden können. Durch eine iterative Ausführung der Vetophase soll sichergestellt werden, dass das gezeigte Verhalten der Delta-Roboters dem gewünschten Verhalten entspricht.

6.4 Technische Realisierung

Abbildung 6.9 zeigt die Komponenten der technischen Realisierung und deren Interaktion untereinander. Dabei stellen die durchgezogenen Pfeile einen direkten Datenfluss zwischen den Komponenten in Pfeilrichtung dar. Die gestrichelte Pfeillinie zeigt den Austausch von Dateien oder Textfragmenten an. Die ausgetauschten Elemente oder Dateien werden durch die abgeknickte Ecke in der Umrandung dargestellt. Der Datenfluss verläuft von der NJ über den NJCom zum V-REP Communicator und von dort aus nach V-REP. Dort werden die eingehenden Delta-Roboter und die Gegenstände für den Anwender visualisiert. Die Rückrichtung der Daten aus V-REP enthält, je nach Lernphase, die Entscheidungen, die vom Anwender in der V-REP GUI eingeben wurden. Diese Lerndaten werden zurück bis zum V-REP Communicator geleitet, um daraus die Entscheidungsvektoren zu generieren. Die Entscheidungsvektoren werden zu einem Example-Set zusammengefasst aus der mit Hilfe vom Tree Generator, RapidMiner und Cinco eine DSL erzeugt wird. Diese DSL wird von der Codegenerierung verwendet um daraus den entsprechenden ST-Code automatisiert zu erzeugen. Der ST-Code wird an Sysmac-Studio übertragen und von dort aus durch den Anwender auf die NJ deployed.

6.4.1 Aufzeichnung der Laufzeitdaten

aewZur Ermittlung der Entscheidungspunkte für die Veto- und Entscheidungsphase, ist es notwendig die Laufzeitdaten der Gegenstände auf dem Fließband und der Delta-Roboter aufzuzeichnen, um sie zu einem späteren Zeitpunkt wiederholt oder modifiziert wiederzugeben. Auf diesem Weg können die Entscheidungspunkte zeitlich unabhängig von der NJ ermittelt werden.

Der *Tracer*, welcher die Laufzeitdaten aufzeichnet, kann sowohl mit generierten, als auch mit zur Laufzeit der NJ ausgelesenen Daten befüllt werden. Alle zu einem Zeitpunkt vorhandenen Daten werden in einem *Snatch* gespeichert. Zu Beginn der ersten Entscheidungsphase, in welcher ausschließlich der Benutzer entscheidet welcher Delta-Roboter welchen Gegenstand verpacken soll, wird eine Fließbandbelegung generiert. Diese Generierung erfolgt anhand zuvor festgelegter Parameter, wie z.B. die Dichte der Gegenstände auf dem Fließband. Ausgehend von dieser Fließbandbelegung wird ein Tracer erstellt, welcher in einem $10ms$ Intervall alle Gegenstände und ihre Positionen in einem Snatch erfasst. Nachdem die Aufzeichnung abgeschlossen ist, kann der gesamte Tracer zur späteren Nutzung gespeichert werden. Alternativ kann während der Vetophase der Tracer mit den Laufzeitdaten aus der NJ über eine Netzwerkschnittstelle gespeist werden.

6.4.2 Die Entscheidungsphase

Innerhalb der Entscheidungsphase wird ein zuvor aufgezeichneter Tracer wiedergegeben, indem alle enthaltenen Snatches iterativ an die Visualisierungskomponente weitergegeben werden. Für jeden Snatch wird geprüft, ob ein Gegenstand in den Arbeitsbereich eines Delta-Roboter eingedrungen ist und ob in der Packung des Delta-Roboters noch mindestens ein Platz vorhanden ist. Der Nutzer muss nun für dieses und alle weiteren Gegenstände im Arbeitsbereich entscheiden, ob der Delta-Roboter in Aktion treten und ein Gegenstand greifen und verpacken soll. Es besteht auch die Möglichkeit, dass der Benutzer keines der greifbaren Gegenstände verpacken lassen möchte. In diesem Fall werden die sich im Arbeitsbereich befindlichen Gegenstände für den Delta-Roboter für 3 Sekunden blockiert , solange kein neuer Gegenstand in den Arbeitsbereich eindringt. Zur Ermittlung, ob ein Gegenstand in den Arbeitsbereich eines Delta-Roboters eingedrungen ist, wird für jeden Gegenstand im aktuellen Snatch die Entfernung mit Hilfe des Satz des Pythagoras in der XY-Ebene zu den Montageachse der Delta-Roboter geprüft. Sobald ein Gegenstand weniger als $500mm$ von einem Montageachse entfernt ist, befindet er sich innerhalb des Arbeitsbereich und kann gegriffen werden, wodurch ein Entscheidungspunkt vorhanden ist. Die gegriffenen Gegenstände dürfen in den nachfolgenden Snatches nicht mehr angezeigt werden. Aus diesem Grund wird eine *Blacklist* innerhlab des Tracers gehalten, welche die gegriffenen Gegenstände aus allen nachfolgenden Snatches während der Weitergabe an die Visualisierungskomponente filtert.

6.4.3 Die Vetophase

Die Vetophase erlaubt dem Benutzer die Aktionen der Delta-Roboter zu kontrollieren und zu bewerten, indem nach jedem verpackten Gegenstand festgelegt wird, ob die Aktion der angestrebten Arbeitsweise entspricht. Während der Vetophase wird ebenfalls ein Tracer iterativ wiedergegeben. Die Zeitpunkte zu denen der Benutzer eine Entscheidung treffen muss, werden während dieser Phase ermittelt, wenn ein Delta-Roboter einen Gegenstand verpackt hat. Um diese Zeitpunkte zu ermitteln muss der aktuelle und der vorherige Snatch betrachtet werden, um die Veränderung des Greifstatus des Delta-Roboters zu erfassen. Sobald ein Delta-Roboter einen Gegenstand verpackt hat, wechselt der Zustand des Delta-Roboters. Der Benutzer muss anschließend entscheiden, ob die Aktion des Delta-Roboters der angestrebten Arbeitsweise entspricht. Um die notwendigen Informationen für die Entscheidung zu erfassen, muss der Snatch ermittelt werden zu dem Zeitpunkt zu dem der Gegenstand vom Fließband gegriffen wurde, um beispielsweise die Anzahl der umliegenden Gegenstände zu ermitteln. Dieser Zeitpunkt kann anhand des Greifstatus des Gegenstands ermittelt werden, welcher angibt ob der Gegenstand von einem Delta-Roboter gegriffen ist oder nicht. Die gesammelten Informationen werden als Entscheidung an den *Validator* weitergegeben.

6.4.4 Das Example-Set

Die Entscheidungen aus der Veto- und Entscheidungsphase des Benutzers werden in einem Validator gespeichert, welcher ebenfalls zu jedem Zeitpunkt gespeichert werden kann, um zu einem späteren Zeitpunkt weitere Entscheidungen hinzuzufügen. Die Entscheidungen enthalten Informationen welcher Gegenstand von welchem Delta-Roboter gegriffen werden sollte, welche Eigenschaften der Gegenstand besitzt und wie viele Gegenstände sich in einem Umkreis von $200mm$ befinden. In der Entscheidungsphase wird ebenfalls erfasst, welche Gegenstände vom Nutzer vorsätzlich nicht gegriffen wurden. Anhand dieser Informationen kann ein Example-Set in Form einer CSV erstellt werden. Zu diesem Zweck werden für jede Entscheidung eine Zeile für jeden in dieser Entscheidung enthaltenen Gegenstand erstellt. Jede Zeile kodiert binär, welche Eigenschaften, wie viele Gegenstände sich im Umkreis befanden, in welchem Viertel des Arbeitsbereich sich der Gegenstand befand und ob er gegriffen wurde oder nicht. Zur Bestimmung der Eigenschaften eines Gegenstands werden zunächst alle Eigenschaften die vorkommen können benötigt. Diese Eigenschaften stellen die ersten Spalten im Example-Set dar, wodurch im Anschluss lediglich für jeden Gegenstand überprüft werden muss, ob eine die Eigenschaft der Spalte auf den jeweiligen Gegenstand zutrifft, sodass diese Zelle in der CSV mit einer 1 gekennzeichnet werden kann. Die Anzahl der umliegenden Gegenstände Cluster ergeben sich aus einer äquivalenten Berechnung der Arbeitsbereiche der Delta-Roboter, mit dem Unterschied, dass nicht die Montageachse, sondern die XY-Koordinaten des zu greifenden Gegenstands

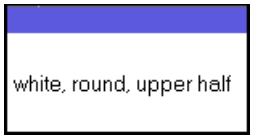


Abbildung 6.10: Anzeige der Gegenstandeigenschaften in V-REP.

den Ausgangspunkt bilden. Das Viertel des Arbeitsbereich in dem sich der Gegenstand zum Zeitpunkt des Entscheidungspunkts befand, wird durch die Position der Gegenstands im Bezug auf die Mitte des Arbeitsbereich bestimmt. Die letzte Spalte kennzeichnet, ob ein Gegenstand gegriffen wurde oder nicht. Im Example-Set wird zwischen der oberen und unteren Hälfte in positive Y-Richtung und der vorderen und hinteren Hälfte in negative X-Richtung unterschieden um jedes Viertel binär zu kodieren.

Um das Example-Set erzeugen zu können, müssen gewisse Informationen zwischen V-REP und dem NJCom ausgetauscht werden. Zunächst wird ein Signal, welches alle Eigenschaften der Gegenstände beinhaltet, von dem NJCom zu V-REP gesendet. Anhand dieser Information wird dem Benutzer am unteren Rand der Szene ein Informationsfenster (Abbildung 6.10) angezeigt, dass die Eigenschaften eines ausgewählten Gegenstandes anzeigt.

In Abbildung 6.11 sind die Dialoge abgebildet, welche bei Entscheidungspunkten erscheinen und eine Benutzereingabe erwarten. In der Entscheidungsphase kann der Benutzer einen Gegenstand auswählen, woraufhin die ID des Gegenstands unter *Selected Objects* erscheint, und mit *Confirm* die Auswahl bestätigen oder durch *Do nothing* keine Aktion durchführen. Wird ein Gegenstand ausgewählt und bestätigt, wird dem Example-Set dieser Gegenstand als Positivbeispiel hinzugefügt. Entscheidet sich der Nutzer nichts auszuwählen, werden alle greifbaren Gegenstände als Negativbeispiele aufgefasst.

In der Vetophase wird nach jeder Aktion eines Delta-Roboter der Dialog eingeblendet. Der Benutzer kann nun entscheiden, ob die Aktion korrekt war, woraufhin der Gegenstand dem Example-Set erneut als Positivbeispiel hinzugefügt wird um die Entscheidung zu verstärken, oder ob die Aktion falsch war und der Gegenstand als Negativbeispiel aufgefasst werden soll.

Generierung der Entscheidungsbäume

Die technische Umsetzung des Abschnittes „Entscheidungsbaum generieren“ der Vetophase (siehe Kapitel 6.1) umfasst drei unterschiedliche Komponenten. Das Zusammenspiel dieser drei Komponenten ist für die Umsetzung der beiden, in Kapitel 6.1 beschriebenen Aktivitäten „Entscheidungsbaum erlernen“ und „Entscheidungsbaum zeichnen“, sprich der Generierung von Entscheidungsbäumen aus den Example-Sets, zuständig. Zunächst werden die drei Komponenten kurz vorgestellt und danach deren Verwendung anhand der Aktivitäten erläutert. Die drei Komponenten lauten wie folgt:

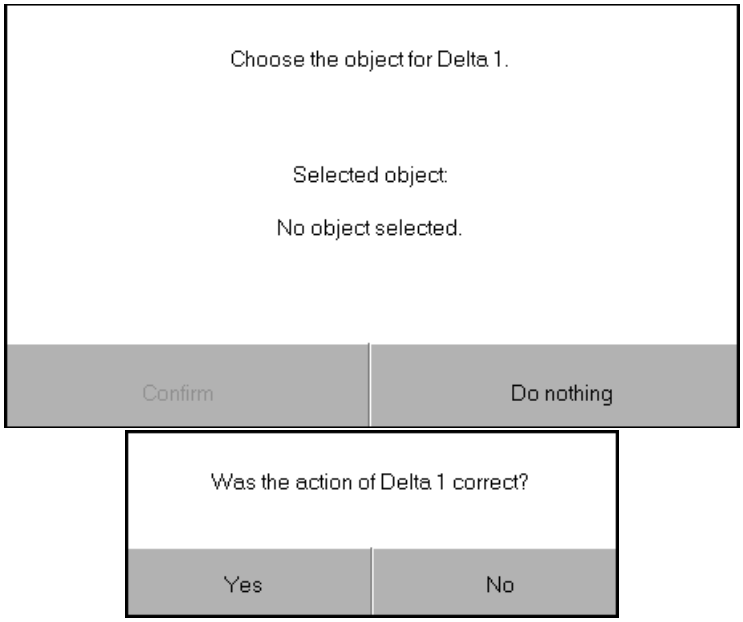


Abbildung 6.11: Dialoge der Entscheidungs- und Vetophase.

- **RapidMiner:** Bei RapidMiner [11] handelt es sich um ein Programm zur Durchführung von maschinellem Lernen sowie Data-Mining.
- **Zeichnungskomponente:** Die Zeichnungskomponente ist, wie der Name schon verrät, zuständig für Zeichnung des Entscheidungsbaumes als Elemente der DSL (siehe Kapitel 2).
- **TreeGen:** Die Komponente TreeGen dient als Schnittstelle zum Lernprogramm RapidMiner. Es ist für die Ansteuerung von RapidMiner sowie die Weitergabe des daraus generierten Entscheidungsbaumes zuständig.

Die Steuerung der einzelnen Komponenten ist in einer *Cinco Custom Action* implementiert. Diese dient als Schnittstelle zum Benutzer und ist als eigener Eintrag im Kontextmenü innerhalb von einem EasyDelta-Projekt in Cinco für den Benutzer auswählbar. Sie dient somit als Startpunkt für die Generierung der Entscheidungsbäume aus den Example-Sets heraus.

Aktivität: Erlernen des Entscheidungsbaums

Der Abschnitt „Entscheidungsbaum generieren“ startet mit der Aktivität *Erlernen des Entscheidungsbaums*. Zu Beginn wird die Komponente TreeGen innerhalb der Cinco Custom Action mit dem Example-Set des Delta-Roboter aufgerufen zu dem ein Entscheidungsbaum erzeugt werden soll. Da TreeGen, wie Eingangs bereits erwähnt, als Schnittstelle zu RapidMiner fungiert, hat es die Aufgabe das Tool RapidMiner zu initialisieren und das Erlernen des Entscheidungsbaums, aus dem betreffenden Example-Set heraus, anzustoßen.

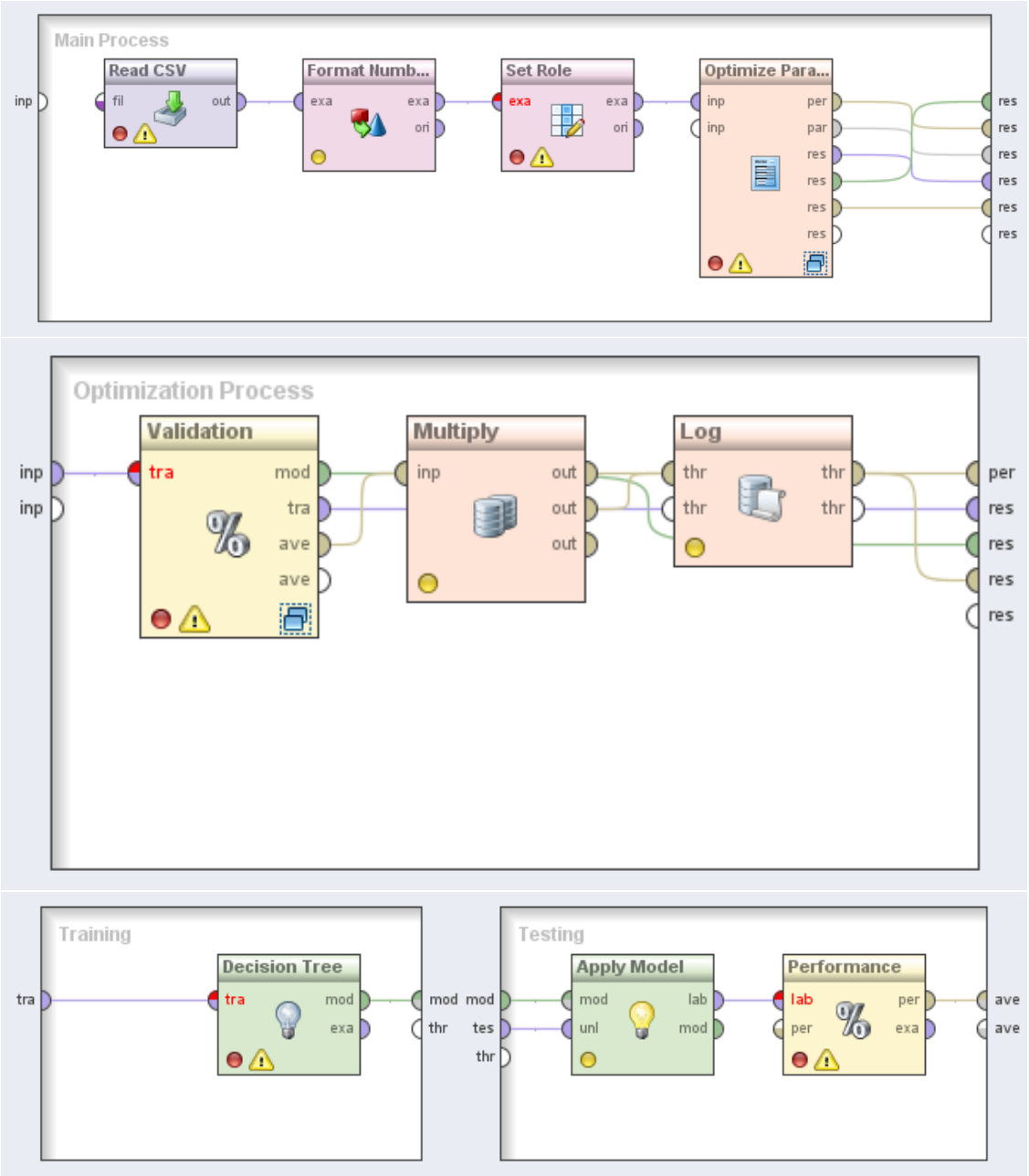


Abbildung 6.12: Grafische Darstellung des eingesetzten RapidMiner Prozesses.

Dazu wird einem RapidMiner-Prozess der Pfad zu einem Example-Set übergeben. Der Prozess ist in Abbildung 6.12 dargestellt. Zunächst wird die CSV Datei in der sich das Example-Set befindet mit Hilfe des Operators *Read CSV* geladen. Anschließend werden die vorhanden Attribute im Operator *Format Numbers* als numerische Werte klassifiziert und der Operator *Set Role* markiert das Attribut *pick* als Label. Das Label ist dasjenige Attribut, nachdem sich das maschinelle Lernen richtet. Als nächstes wird der Subprozess *Optimize Parameters* aufgerufen. Dieser Subprozess stellt das eigentliche Lernen dar. In ihm befinden sich der Subprozess *Validation* sowie die Operatoren *Multiply* und *Log*. Die letzten beiden protokollieren die Lerndurchläufe und dienen lediglich zum Debuggen. Der Subprozess *Validation* generiert einen Entscheidungsbaum mit Hilfe des Operators *Decision Tree* und 90% der Testdaten und validiert diesen Baum mit *Apply Modell* anschließend mit den übrigen 10% der Daten und ermittelt ihre Performanz mit dem Operator *Performance* bezüglich der Richtigkeit, der *accuracy*. In Abbildung 6.13 sind die Parameter des Operators *Decision Tree* dargestellt, über welche der Subprozess *Optimize Parameters* optimiert. Der Parameter *criterion* bestimmt das Kriterium nach dem der Entscheidungsbaum generiert werden soll. Die Kriterien sind *information_gain* (welche die Entropie als Verzweigungskriterium wählt), *gin_ratio* (welches das *information_gain* auf ein Attribut fokusiert), *gini_index* (welches die Störstellen in der Beispielmenge misst und versucht den gemessenen Index zu reduzieren) und *accuracy* (welches nur nach Attributen verzweigt, wenn die Richtigkeit des Baumes steigt). Die Parameter *minimal_size_for_split* und *minimal_leave_size* geben an, wie viele Beispiele in einem Knoten bzw. in einem Blatt vorhanden sein müssen damit dort verzweigt oder es als Blatt gewählt wird. Dabei wird für jede Parameterkombination ein Durchlauf gestartet und die Performanz verglichen. Zum Abschluss wird derjenige Baum ausgewählt, der die beste Performanz aufweist.

Hat RapidMiner den Lernprozess des Entscheidungsbaums abgeschlossen wird dieser an die Komponente TreeGen zurückgegeben. TreeGen transformiert den Entscheidungsbaum an dieser Stelle in ein Zwischenmodell. Dieser Schritt ist notwendig, da viele Informationen über einzelne Knoten des Entscheidungsbaums von RapidMiner in ausgehenden Kanten gespeichert sind. Diese Informationen werden für den späteren Vorgang des Zeichnens der Zeichnungskomponente benötigt. Um innerhalb der Zeichnungskomponenten unnötige Indirektionen zum Abrufen von Informationen zu vermeiden und somit die Traversierung durch den Entscheidungsbaum zu erleichtern, transformiert TreeGen den Entscheidungsbaum in ein eigenes Baummodell, in dem alle Informationen zu einzelnen Knoten in den jeweiligen Knoten selbst gespeichert sind. Darüber hinaus enthält jeder Knoten eine Referenz zu seinem Vaterknoten, um somit auch eine Bottom-Up Traversierung durch den Entscheidungsbaum zu ermöglichen, welches mit dem von RapidMiner generierten Baum nicht möglich ist.

Zum Abschluss der Aktivität *Erlernen des Entscheidungsbaums* wird das Zwischenmodell von TreeGen an die Zeichnungskomponente übergeben.

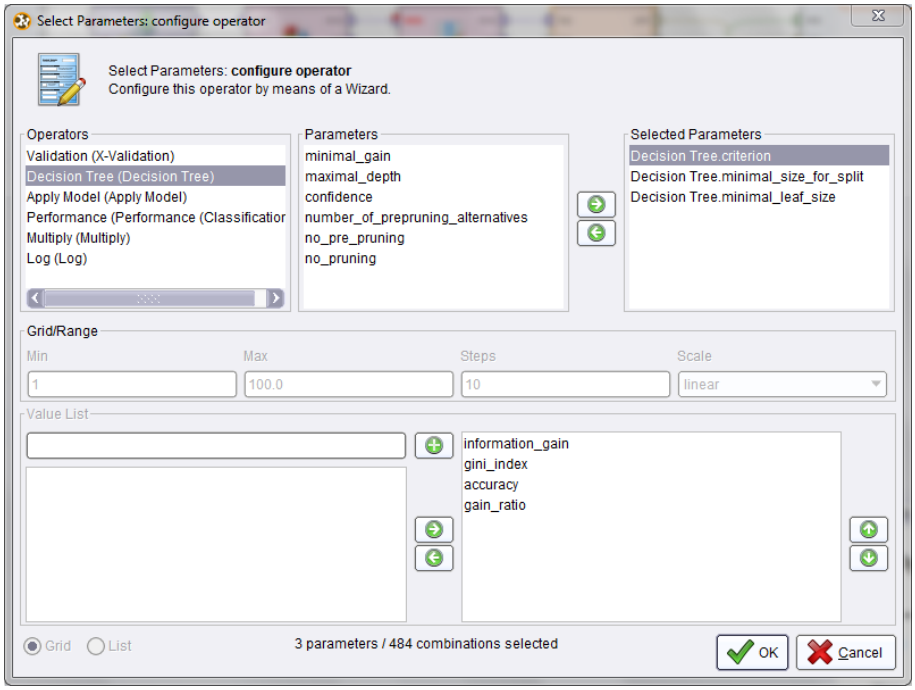


Abbildung 6.13: Die Parameter über die optimiert wird.

Aktivität: Zeichnen des Entscheidungsbaums

Die Aktivität *Zeichnen des Entscheidungsbaums* startet mit der Übergabe des Zwischenmodells von dem erlernten Entscheidungsbaums von RapidMiner an die Zeichnungskomponente durch TreeGen. Das TreeGen Zwischenmodell ist bereits so aufgebaut, dass die Struktur weitestgehend übernommen werden kann. Die Knoten des Modells werden auf Komponenten der DSL(siehe Kapitel 2.3.2) abgebildet. Dabei entsprechen die Entscheidungsattribute, welche in Kapitel 6.2 erläutert wurden, entweder einem *Characteristic Gateway* oder einer *externen Entscheidung*. Die Unterteilung findet wie folgt statt:

- Abbildung durch ein Characteristic Gateway:
  - black
  - white
  - blue
  - green
  - red
  - round
  - angular
- Abbildung durch eine externe Entscheidung:
  - topC (obere Fließbandhälfte)

- frontWB (vordere Hälfte des Arbeitsbereich eines Delta-Roboters)
- cluster1
- cluster2
- cluster3
- cluster4

Zur Traversierung des Zwischenmodells von TreeGen wird eine Tiefensuche anstatt einer Breitensuche durchgeführt. Dies bietet den Vorteil gegenüber der Breitensuche, dass ohne komplexes Berechnen der Abstände und Knoten, sondern einfach mit einem festen Offset gearbeitet werden kann. Die Laufzeit einer Tiefensuche ist $O(V+E)$ mit `V` als Anzahl der Knoten und `E` als Anzahl der Kanten. Durch den festen Offset kann es zu Überschneidungen von Knoten kommen, jedoch lag hier der Schwerpunkt auf der Richtigkeit des Ergebnisses und weniger auf einem optisch schönen Graphen.

Beim Durchlaufen des Baumes wird der Name des Knotens geprüft und der entsprechende Knoten der DSL gezeichnet. Der Name spiegelt jeweils die oben genannten Eigenschaften wider. Bei einem Characteristic Gateway gibt es zwei ausgehende Kanten. Zum Einen die Kante mit der Eigenschaft, welche gewählt wird wenn der Gegenstand die Eigenschaft besitzt und zum Anderen eine *else-Kante*, welche gewählt wird wenn der Gegenstand die Eigenschaft nicht besitzt.

Eine externe Entscheidung hat eine *true-* und eine *false-Kante*. Besitzt somit ein Gegenstand die gewünschte Eigenschaft, wird die *true-Kante* gewählt, ansonsten die *false-Kante*.

Da für das Lernen feste Entscheidungsattribute notwendig sind, wird an dieser eine feste Konfiguration benötigt. Es müssen bestimmte Charakteristika und Gegenstände vordefiniert sein. Aufgrund dessen wurde ein Rahmenprogramm entwickelt, welches die benötigte Konfiguration des Lernens beinhaltet.

Das Rahmenprogramm

Das Rahmenprogramm ist durch eine Cinco Custom Action realisiert. Dies ermöglicht es, dass in jedem EasyDelta Modell, durch Auswahl des Eintrages „Draw example program“ im Kontextmenü, beim Rechtsklick auf die Zeichenfläche, das Rahmenprogramm automatisch gezeichnet wird. In Abbildung 6.14 ist das fertig gezeichnete Rahmenprogramm zu sehen. Dort sind sieben Charakteristika sowie zehn Gegenstandstypen, welche jeweils eine Kombination aus einer Farbe und einer Form darstellen, definiert. Ebenso sind zwei Delta-Roboter-Swimlanes mit den dazugehörigen Kisten dargestellt.

Der Ablauf in den Delta-Roboter-Swimlanes besteht darin, dass zunächst alle erreichbaren und platzierbaren Gegenstände angefordert werden, daraufhin durch einen Iterator gefiltert werden und abschließend das Erste dieser Elemente vom Delta-Roboter gegriffen und verpackt wird. Im Iterator wird dann der gelernte Baum gezeichnet, worauf später noch genauer eingegangen wird.

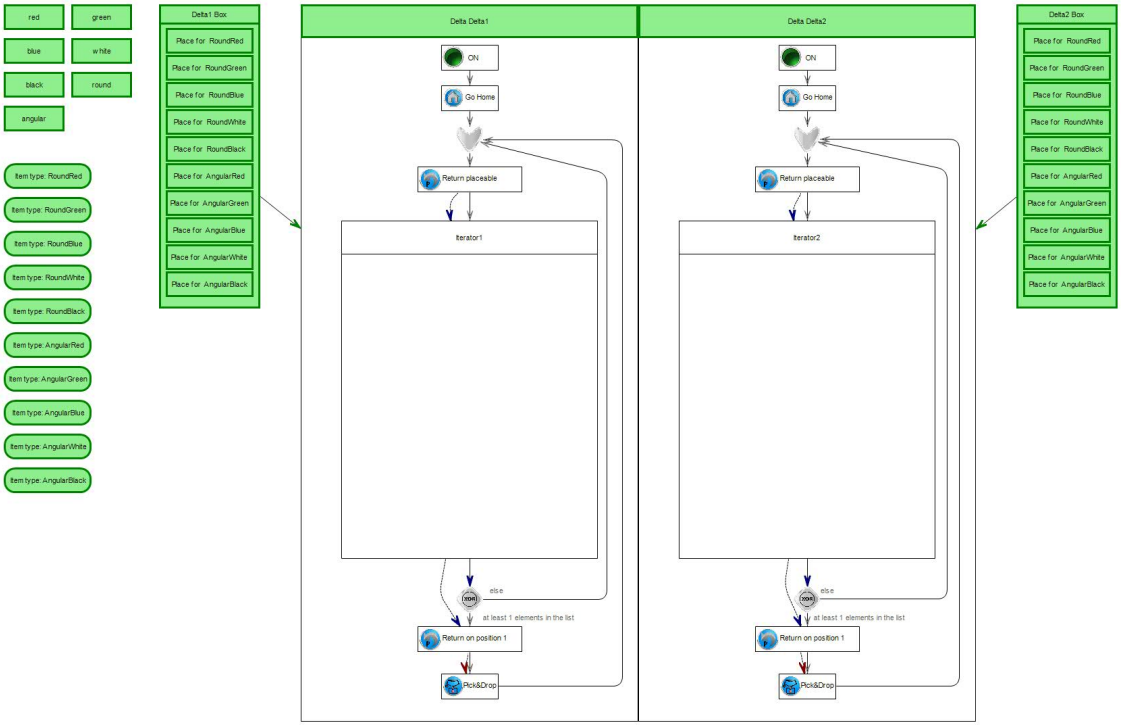


Abbildung 6.14: Rahmenprogramm fürs Lernen.

Neben den Charakteristika und den Gegenständen, werden ebenso externe Entscheidungen fürs Lernen benötigt. Auf Abbildung 6.15 sind alle Entscheidungen abgebildet, die beim Zeichnen des Rahmenprogramms mit generiert werden. Dabei entspricht:

- **topC:** linke Seite des Fließbandes
- **frontWB:** vorderer Arbeitsbereich
- **cluster1-4:** Cluster 1-4

Diese externen Entscheidungen beinhalten schon den notwendigen ST-Code um dies später auch simulieren zu können.

Ebenso wie das Rahmenprogramm ist das Lernen mit Hilfe einer Cinco Custom Action implementiert worden. Um den erlernten Baum zeichnen zu lassen, muss hier im Kontextmenü „Draw learned sorting algorithm“, durch Rechtsklick auf einen beliebigen Iterator, ausgewählt werden. Es werden daraufhin in beide Iteratoren der Delta-Roboter-Swimlanes die entsprechenden Abläufe gezeichnet. Dabei wird folgendermaßen vorgegangen:

1. Zunächst wird *StartIterator* gezeichnet, da jeder Iterator damit startet.
2. Daraufhin wird der Baum des Zwischenmodells durchlaufen und alle entsprechenden Knoten der DSL, wie vorhin beschrieben, gezeichnet.

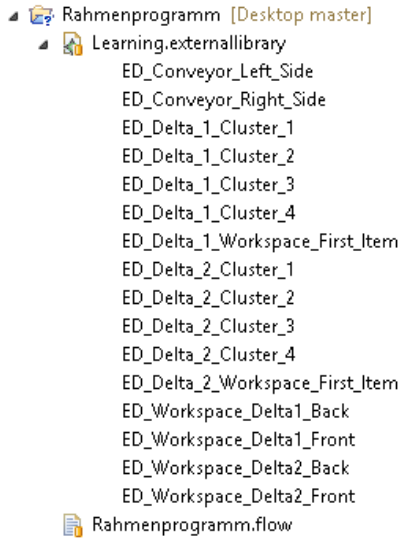


Abbildung 6.15: Externe Entscheidungen fürs Lernen.

3. Bei den Blättern des Zwischenmodells wird abhängig davon, ob das Element gegriffen werden soll oder nicht, entweder *AddToTemplist* oder *Enditerator* gezeichnet. Wurde einer dieser beiden Knoten bereits einmal gezeichnet, wird nur noch eine Kante dorthin gezogen.

Soll ein Delta-Roboter alle oder gar keine Gegenstände greifen, so sind dies Sonderfälle beim Zeichnen. In diesem Fall existiert kein Zwischenmodell und es muss vorher geprüft werden, ob einer dieser Fälle vorliegt. Soll ein Delta-Roboter alle Gegenstände greifen, so besteht der Ablauf im Iterator lediglich aus dem Knoten StartIterator, welcher eine Kante zum Knoten AddToTempList besitzt. Soll der Delta-Roboter hingegen gar keine Gegenstände greifen, so geht die Kante vom Knoten StartIterator, statt zum Knoten AddToTempList, zum Knoten EndIterator.

In Abbildung 6.16 sind zwei generierte Entscheidungsbäume zu erkennen. Im ersten Iterator wird das Verhalten vom ersten Delta-Roboter modelliert und im zweiten Iterator das für Delta-Roboter zwei. Beim ersten Delta-Roboter wird zunächst danach entschieden, ob ein Gegenstand die Eigenschaft rot besitzt oder nicht. Daraufhin wird bei einen rotem Gegenstand geprüft ob dieses im oberen Arbeitsbereich liegt. Ist ein Gegenstand rot und ebenso im oberen Arbeitsbereich, so wird dieser gegriffen. Beim zweiten Delta-Roboter wird ein Gegenstand gegriffen wenn dieser rot und nicht eckig ist oder blau und nicht eckig ist.

Wurden nun die erlernten Algorithmen gezeichnet, kann nun daraus wieder Code generiert und simuliert werden.

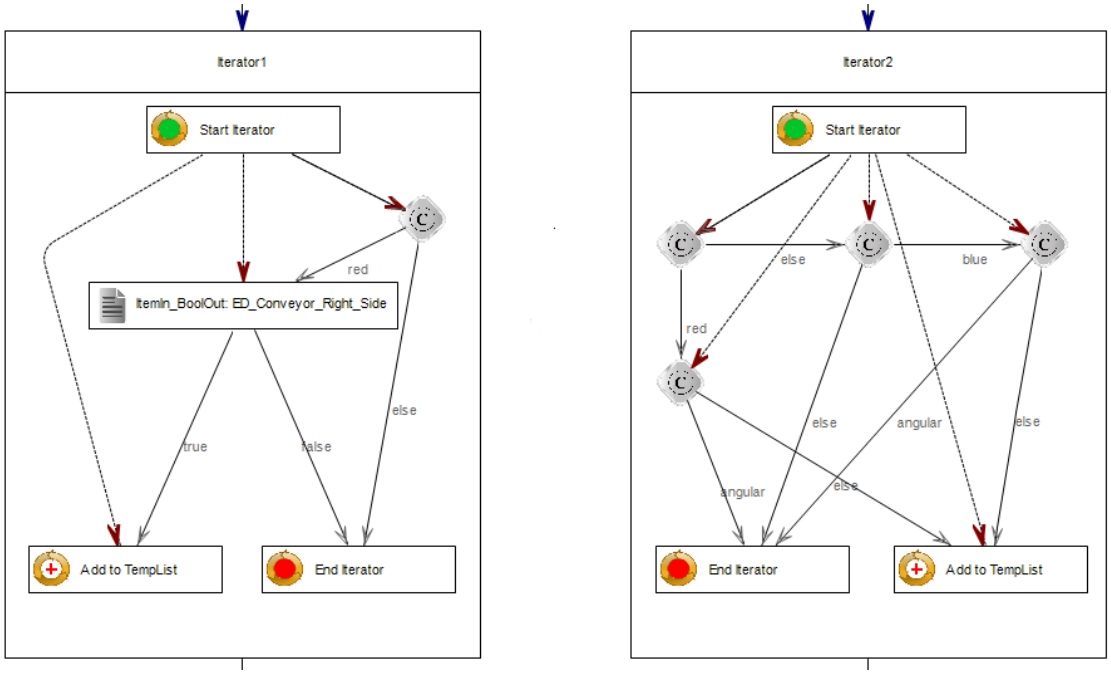


Abbildung 6.16: Iterator mit gelernten Entscheidungsbäumen für beide Delta-Roboter.

6.5 Evaluation des Lernverfahrens

Im folgenden Abschnitt wird das Lernverfahren evaluiert. Dazu wird zuerst der Aufbau der Testfälle beschrieben, auf dessen Grundlage dann Testfälle aufgestellt werden. Anschließend wird auf die Durchführung der Tests eingegangen. Abschließend werden die durchgeführten Testfälle mit Metriken bewertet. Mit Hilfe der Bewertung der Testfälle wird dann das Lernverfahren evaluiert.

6.5.1 Aufbau der Testfälle

Die Testfälle sind alle nach dem selben Prinzip aufgebaut. Dem Anwender bzw. Tester wird mit einer Beschreibung vorgegeben, welche Aktionen er in der Entscheidungsphase tätigen soll. Das heißt es wird dem Anwender vorgeschrieben, welche Gegenstände er zu welchen Gegebenheiten greifen soll. In der Vetophase soll der Anwender für Verhalten, welches von der Beschreibung abweicht, ein Veto einlegen. Das erwartete Ergebnis ist, dass durch das Lernverfahren ein DSL-Programm bzw. ein Programm in ST, welches das vorgegebene Verhalten möglichst exakt beschreibt, erzeugt wird.

Die Beschreibung eines Testfalls bezieht sich dabei auf Eigenschaften, welche die Gegenstände direkt betreffen, wie die Form und Farbe oder auf Eigenschaften, welche sich auf die Lage eines Gegenstandes auf dem Fließband beziehen. Dabei handelt es sich um genau die Eigenschaften, welche zur Entscheidungsfindung innerhalb des Lernverfahren verwendet werden können (siehe auch 6.2). Damit ist es möglich direkt im DSL-Programm, welches

mit dem Lernverfahren erzeugt wurde, zu sehen, ob das beschriebene Verhalten erlernt wurde, weil in der DSL genau diese Eigenschaften abgebildet werden.

Weiterhin wird durch das Vorgehen, dass dem Anwender vorgeschrieben wird welche Aktionen er zu welchen Gegebenheiten durchführen soll, gewährleistet das die Testfälle reproduzierbar sind. Durch den menschlichen Einfluss in jedem Testfall, ist aber ein deterministischer Testablauf nicht sichergestellt und somit auch der Ausgang bzw. das Ergebnis eines Tests nichtdeterministisch. Dieser Nichtdeterminismus zeigt sich z.B. darin, dass der Anwender nicht immer genau bestimmen kann ob ein Gegenstand in der oberen Hälfte des Fließbandes liegt oder in der unteren Hälfte, wenn er mittig auf dem Fließband liegt. Der Nichtdeterminismus verfälscht aber in sofern die Testergebnisse nicht, weil auch in der späteren Anwendung des Lernverfahrens immer ein Mensch die Bedienung übernimmt. Des Weiteren sind die Testfälle so gewählt, dass alle Eigenschaften die erlernt werden können in den Testfälle enthalten sind.

6.5.2 Testfälle

Die folgenden Testfälle stellen Anweisungen an den Anwender bzw. Tester, welche er in der Entscheidungsphase durchführen soll. In der Vetophase soll der Anwender, wie oben schon beschrieben, für Verhalten das von der Beschreibung des Testfalls abweicht ein Veto einlegen.

Um die Testfälle realitätsnah zu gestalten, werden bei der Fließbandbelegung nur bestimmte Gegenstände erzeugt. Dazu gibt es zum Einen Testfälle, in denen alle Gegenstände aufgenommen werden und zum Anderen Testfälle, in denen Gegenstände bewusst nicht gegriffen werden. Der letztere Fall hat den Zweck zu simulieren, dass Ausschussgegenstände nicht verpackt werden. Dabei könnte es sich beispielsweise um gebrochene Kekse handeln.

1. Testfall

Der Nutzer lässt den ersten Delta-Roboter alle runden Gegenstande (Round) greifen und den zweiten Delta-Roboter alle eckigen (Angular) Gegenstände.

Gegenstände in Szenario: Alle verfügbaren

2. Testfall

Der Nutzer lässt den ersten Delta-Roboter in der oberen Hälfte des Fließbandes rote runde Gegenstände (RedRound) und in der unteren Hälfte des Fließbandes blaue runde Gegenstände (BlueRound) greifen. Den zweiten Delta-Roboter lässt der Nutzer in der oberen Hälfte des Fließbandes blaue runde Gegenstände (BlueRound) und in der unteren Hälfte des Fließbandes rote runde Gegenstände (RedRound) greifen. Alle eckigen (Angular) Gegenstände sollen von keinem Delta-Roboter gegriffen werden.

Gegenstände in Szenario: RedRound, BlueRound, RedAngular, BlueAngular

3. Testfall

Der Nutzer lässt den ersten Delta-Roboter in der oberen Hälfte des Fließbandes rote runde Gegenstände (RedRound) und in der unteren Hälfte des Fließbandes blaue eckige Gegenstände (BlueAngular) greifen. Den zweiten Delta-Roboter lässt der Nutzer in der oberen Hälfte des Fließbandes blaue eckige Gegenstände (BlueAngular) und in der unteren Hälfte des Fließbandes rote runde Gegenstände (RedRound) greifen.

Gegenstände in Szenario: RedRound, BlueRound, RedAngular, BlueAngular

4. Testfall

Der Nutzer lässt den ersten Delta-Roboter im vorderen Arbeitsbereich grüne runde Gegenstände (GreenRound) und im hinteren Arbeitsbereich schwarze eckige Gegenstände (BlackAngular) greifen. Den zweiten Delta-Roboter lässt der Nutzer im vorderen Arbeitsbereich blaue eckige Gegenstände (BlueAngular) und im hinteren Arbeitsbereich rote runde Gegenstände (RedRound) greifen. Die weiß eckigen Gegenstände (WhiteAngular) sollen von keine Delta-Roboter gegriffen werden.

Gegenstände in Szenario: GreenRound, RedRound, BlackAngular, BlueAngular, WhiteAngular

5. Testfall

Der Nutzer lässt den ersten Delta-Roboter in der oberen Hälfte des Fließbandes im vorderen Arbeitsbereich ein Gegenstand greifen, wenn es sich in einem 4-Cluster befindet. Den zweiten Delta-Roboter lässt der Nutzer in der unteren Hälfte des Fließbandes im hinteren Arbeitsbereich ein Gegenstand greifen, wenn es sich in einem 4-Cluster befindet.

Gegenstände in Szenario: Alle verfügbaren

6. Testfall

Der Nutzer versucht mit dem ersten Delta-Roboter alle 4er, 3er, 2er, 1er Cluster aufzulösen. Mit dem zweiten Delta-Roboter versucht der Nutzer, so weit noch vorhanden, Cluster aufzulösen bzw. versucht die restlichen Gegenstände zugreifen.

Gegenstände in Szenario: Alle verfügbaren

Das erwartete Ergebnis aus den Testfällen entspricht dem Verhalten, dass der Anwender im jeweiligen Szenario vorgibt.

6.5.3 Durchführung der Tests

Bei der Durchführung der Testfälle wird bis zu zwei Mal, alternierend, eine Entscheidungsphase und Vetophase nach dem Schema des jeweiligen Testfalles durchgeführt. Auf jede Phase folgt die Bewertung des Resultats in einem weiteren Durchlauf, indem durch

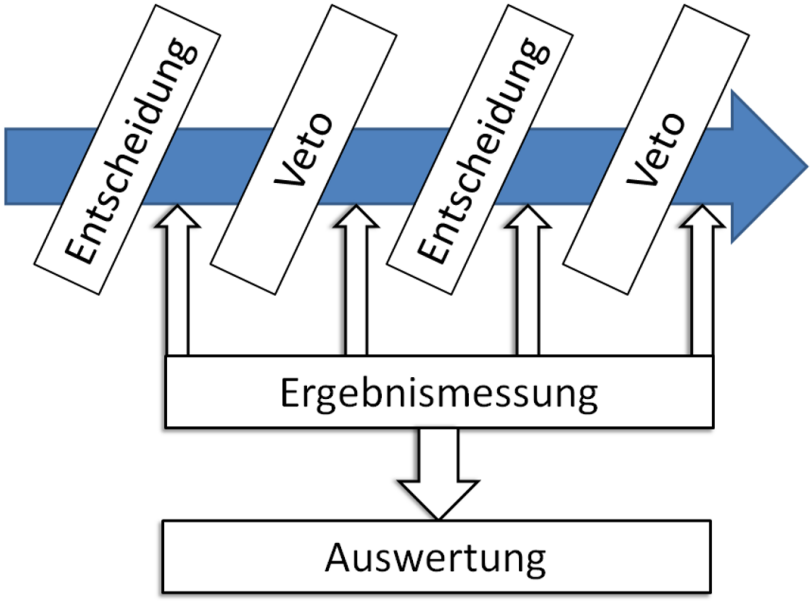


Abbildung 6.17: Ablauf des Tests.

Beobachten alle falschen und richtigen Aktionen gezählt werden. Die Entscheidung ob eine Aktion richtig war wird dabei vom Tester nach dem jeweiligen Schema des Testfalls entschieden. Für jede Phase des Tests wird eine neue zufällige Bandbelegung mit 50 Gegenständen verwendet.

Abbildung 6.17 verdeutlicht den Ablauf des Tests. Der Test beginnt mit einer Entscheidungsphase nach der das Ergebnis gemessen wird. Drauf folgt eine Vetophase nach der eine weitere Messung durchgeführt wird. Dieser Prozess wird im Anschluss noch einmal wiederholt, so dass vier Messerwerte mit einem Wertebereich zwischen 0 und 100 Prozent richtiger Entscheidungen entstehen. Dadurch lässt sich nachvollziehen wie schnell und wie korrekt das System die jeweiligen Testfälle lernen konnte. Ergibt sich bereits vor Ende der letzten Phase ein Testergebnis von 100 Prozent richtigen Entscheidungen, so wird der Lernprozess nicht weiter fortgesetzt und die weiteren Messpunkte mit 100 Prozent bewertet.

6.5.4 Metriken

Als Metrik zur Messung der Effizienz für den jeweiligen Testfall wird die relative Häufigkeit der richtigen Entscheidungen herangezogen. Diese Metrik zielt auf die Messung der Lerneffizienz ab und gibt an, wie hoch der Anteil korrekter Entscheidungen nach Durchführung des Tests ist. Zu jedem Testfall entstehen vier dieser Werte die nachfolgend in einer Tabelle und einem Diagramm dargestellt werden.

6.5.5 Bewertung des Lernverfahrens

Die oben beschriebenen Testfälle wurden nach dem vorgegebenen Ablauf getestet. Dabei ergaben sich die folgenden Ergebnisse die in Tabelle 6.1 und Abbildung 6.18 abgebildet sind. Die Ergebnisse der Tests zeigen, dass für die ersten beiden Testfälle nach der Phase *2:Veto* bereits ein Ergebnis von 100% erzielt werden konnte. Das Lernergebnis nach der Entscheidungsphase war noch nicht vollständig, da noch Negativ Beispiele aus der Vetophase nötig waren um das Lernbild zu komplettieren. Weiter haben die beiden Testfälle, die sich insbesondere in der Anzahl der Itemtypen unterschieden gezeigt, dass es für das Lernverfahren Problemlos möglich ist mit vielen Itemtypen zu Lernen, auch wenn nur einige davon aufgenommen werden sollen.

In den Testfällen drei und vier konnte über alle vier Phasen hinweg ein Ergebnis von 60% erzielt werden, was in der Praxis noch nicht ausreichend wäre. Eine schwäche des Lernverfahrens sorgt in dieser Beispielkonstellation dafür, dass der Baum sehr komplex wird. Dadurch läuft der Lernvorgang in ein falsche Richtung und kann auch durch weitere Lernphasen nur schwer verbessert werden. Dieses Problem zeigt spiegelt sich in der Ergebnissen darin wieder, dass in den Testfällen drei und vier die Lernergebnisse aller Phasen zwischen 50% und 60% schwanken.

In den Testfällen fünf und sechs wurde insbesondere das erlernen von Clustern getestet. Hier hat sich in beiden Fällen bereits nach der ersten Entscheidungsphase ein Ergebnis von 100% eingestellt. Der Lernerfolg liegt besonders an der Eindeutigkeit der Beispiele, da alle Gegriffenen Objekte die entsprechende Clusterzahl besaßen.

Insgesamt entsprechen die Ergebnisse den Erwartungen. Vier von sechs Tests konnten während der vier Durchläufe die 100% Quote erreichen. Eine Schwäche hat des Verfahrens hat sich in den Testfällen drei und vier gezeigt. Auffällig während der Durchführung des Verfahrens war es, dass sich teilweise sehr große Lernbäume gebildet haben. An dieser Stelle könnte noch Entwicklungsbedarf in Folgeprojekten bestehen. Weiter ist es im aktuellen Verfahren nicht möglich Reihenfolgen zu erlernen, wie zum Beispiel, dass erst alle roten Objekte und danach alle Blauen Objekte verpackt werden sollen.

6.5.6 Auswertung der Testfälle

Testfall	1:Entsch.	2:Veto	3:Entsch.	4:Veto
1	59,38	100,00	100,00	100,00
2	46,15	100,00	100,00	100,00
3	53,13	42,86	54,55	72,41
4	61,29	51,85	67,74	53,85
5	100,00	100,00	100,00	100,00
6	100,00	100,00	100,00	100,00

Tabelle 6.1: Ergebnisse der Testevaluation.

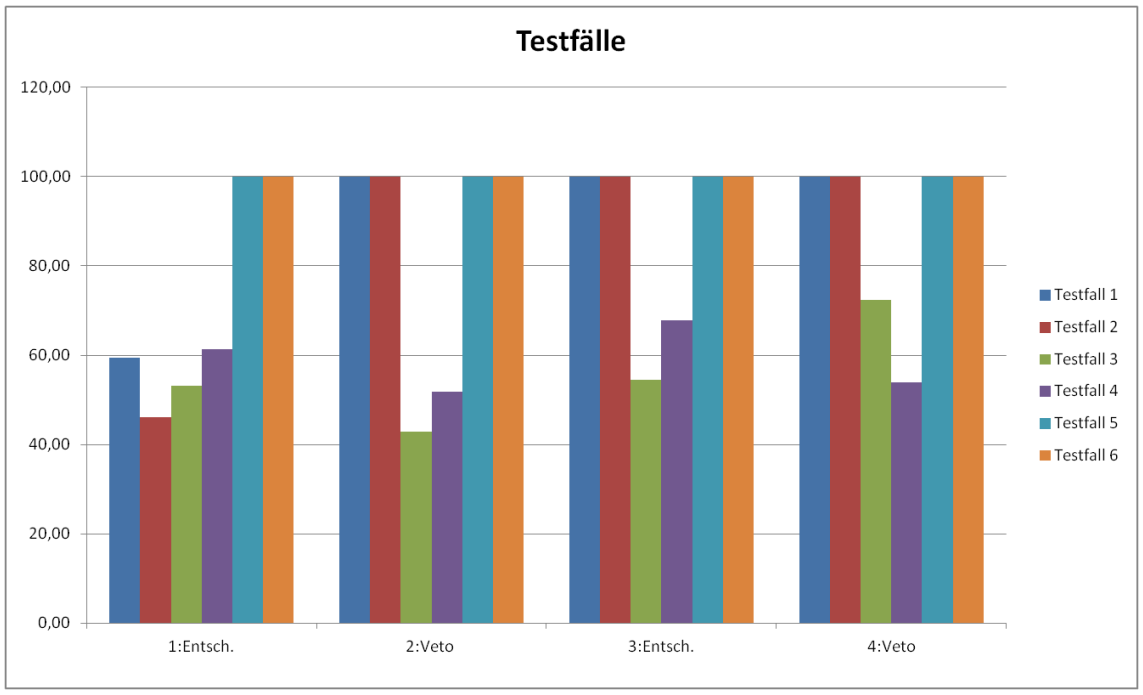


Abbildung 6.18: Ergebnisse der Testevaluation.

Kapitel 7

Zusammenfassung

Im Rahmen dieser Projektgruppe wurde das Konzept „Programming by Example“[5] umgesetzt. In einem iterativen Verfahren wurden zwei Delta-Roboter durch das Aufzeichnen von Benutzerinteraktionen programmiert.

Zunächst wurde mit 3PL eine graphische DSL für Verpackungs- und Sortieralgorithmen erstellt. In Cinco können mit 3PL, ohne tiefgreifende Kenntnisse der Programmierung von Delta-Roboter, per Drag and Drop Algorithmen entwickelt werden. Durch die Verwendung von Prime Referenzen2.3.2 ist 3PL durch zusätzliche Fallunterscheidungen erweiterbar. In Cinco implementierte Programme können mit der eigens erstellten Codegenerierung in die Programmiersprache ST übersetzt werden. ST wird von Sysmac Studio (eine IDE zur Erstellung für Roboterprogramme der Firma OMRON) genutzt. In Sysmac Studio kann der Quellcode in speziell entwickelte Rahmenprogramme eingefügt und kompiliert werden. Die erstellten Programme sind auf einer NJ (ein Programmable Logic Controller der Firma OMRON) deploybar.

Zur Demonstration der Programme wurde eine Visualisierungskomponente bestehend aus V-REP, V-REP Communicator und NJCom implementiert. In V-REP erfolgt die Darstellung der Programmabläufe. Der V-REP Communicator ermöglicht die Kommunikation zwischen V-REP und der NJCom. Die NJCom kann randomisiert Bandbelegungen generieren, Programmabläufe abspielen (die in V-REP angezeigt werden) und Daten aus der NJ auslesen bzw. aufzeichnen.

Das Eingangs erwähnte iterative Verfahren (Lernkonzept) erfolgt in zwei Phasen. In der ersten Phase (Entscheidungsphase) wird die Beförderung der generierten Gegenstände über das Fließband in der Visualisierungskomponente dargestellt. Die Delta-Roboter sind in dieser Phase noch nicht aktiv. Dem Benutzer werden zu wohl definierten Zeitpunkten die auf dem Band befindlichen Gegenstände zum Greifen angeboten. Die Entscheidung des Benutzers wird in der Visualisierungskomponente protokolliert. Nach dem alle Gegenstände abgearbeitet wurden (entweder wurden sie verpackt oder landeten in einer Sammelbox am Ende des Fließbandes), werden die protokollierten Entscheidungen des Benutzers in eine

Lernkomponente exportiert. Die Lernkomponente benutzt RapidMiner (eine Umgebung für maschinelles Lernen und Data-Mining). Sie erstellt ein Programm in 3PL für Cinco. Dieses Programm wird zu ST-Code übersetzt, anschließend in dem Rahmenprogramm in Sysmac Studio eingefügt und auf der NJ deployed.

Die Visualisierungskomponente triggert die zweite Phase (Vetophase) des Lernkonzepts. In dieser Phase wird eine neue Bandbelegung generiert und auf der NJ eingespielt. Die Gegenstände werden in dieser Phase durch das Fließband befördert und entsprechend der Programmierung von den Delta-Robotern gegriffen. Dieser Programmablauf erfolgt auf der NJ und wird von dem NJCom aufgezeichnet. Beim Abspielen kann der Benutzer nach jedem Greifvorgang die Aktion des Delta-Roboters revidieren (Veto). Diese Veto-Entscheidungen des Benutzers werden zu den bereits vorhandenen Entscheidungen aus der vorhergehenden Phase hinzugefügt. Die protokollierten Daten können anschließend wieder in die Lernkomponente exportiert werden und ein neues Programm kann in Cinco generiert werden. Dieses kann dann wieder deployed, abgespielt und bewertet werden. Diese Iterationen können fortgeführt werden bis der Benutzer ein zufriedenstellendes Ergebnis erreicht.

Durch dezidierte und reproduzierbare Tests wurden mehrere erfolgreich gelernte Programme generiert und die Machbarkeit des Verfahrens nachgewiesen.

Kapitel 8

Ausblick

In dieser Projektgruppe wurden Lösungen anhand eines konkreten Verpackungszenarios entwickelt. Dieses Szenario unterlag diversen Einschränkungen, die es ermöglichten ein realistisches Ziel für die Projektgruppe zu definieren. Aus diesen Einschränkungen lassen sich weitere Arbeiten ableiten, die an der Generalisierung der vorgestellten Lösungen ansetzen, um nicht nur ein konkretes sondern beliebige Verpackungsszenarien mit Delta-Roboter unterstützen zu können. Eine Auswahl möglicher Ansätze zur Generalisierung sind wie folgt:

- Unterstützung einer beliebigen Anzahl von zu verpackenden Objekten
- Verwendung von beliebigen Verpackungen
- Nutzung einer beliebigen Anzahl an Delta-Roboter
- Erweiterung der Anzahl der Objekttypen
- das Hinzufügen weiterer Fließbänder

Bei einer Generalisierung müsste die Lernkomponente (siehe Kapitel 6) auch angepasst werden, da sie den Einschränkungen, die hinsichtlich des konkreten Beispielszenarios entstehen, unterliegt. So wird unter anderem nur eine eingeschränkte Menge an Entscheidungsattribute, die speziell für das konkrete Szenario ausgelegt sind, verwendet. Denkbar wäre hier die Entwicklung von generischen Entscheidungsattributen für die Lernkomponente, um beliebige Verpackungszenarien zu unterstützen.

Durch die Verwendung von RapidMiner als Backend der Lernkomponente, welches eine Vielzahl unterschiedlicher Lernverfahren unterstützt, bietet es sich an, neben dem bestehenden Lernverfahren (siehe Kapitel 6.1) weitere Verfahren zu unterstützen. Dies erhöht die Flexibilität der Lernkomponente hinsichtlich verschiedener Verpackungsszenarien.

Aber nicht nur eine Erweiterung des Lernkonzeptes sollte in Erwägung gezogen werden. Auch eine Einschränkung der zur Zeit implementierten Entscheidungskriterien gilt

es zu evaluieren. Derzeit ist ein Nutzen der Eigenschaft Cluster oder vorderer/hinterer Arbeitsbereich fraglich. Hier besteht noch Testbedarf.

Für einen produktiven Einsatz des entwickelten Lernverfahrens könnte die Benutzerfreundlichkeit verbessert werden. Derzeit gibt es noch manuell durchzuführende Schritte. Diese stellen eine Fehlerquelle dar und verkomplizieren die Verwendung der Lernverfahrens unnötig. Eine mögliche Optimierung wäre eine API von Sysmac Studio für den direkten Import von ST-Code. Dadurch würde das manuelle kopieren und einfügen von dem generierten ST-Code durch den Benutzer entfallen. Eine weitere Automatisierung wäre die Zusammenführung der Benutzeraktionen für die Steuerung der Visualisierung und die Steuerung der Generierung des 3PL Programmes in der NJCom Benutzeroberfläche. Zur Zeit werden die Entscheidungsvektoren aus NJCom exportiert und der Benutzer muss nach Cinco wechseln um das 3PL Programm zu erstellen. Durch das Zusammenfassen der aufgeführten Schritte könnte die Durchführung des Lernverfahrens erheblich vereinfacht werden.

Die Initialisierung der verwendeten Programme wäre ein nächster Schritt für die Verbesserung Benutzerfreundlichkeit. Im Augenblick werden die Programme aus verschieden IDEs heraus gestartet und dabei ist für eine erfolgreiche Inbetriebnahme des Lernverfahrens eine vorgegebene Initialisierungsfolge einzuhalten. Im Hinblick auf die Benutzergruppe und das Ziel, die Programmierung von Delta-Robotern zu vereinfachen, sollte die Verbesserung der Benutzerfreundlichkeit keine untergeordnete Rolle spielen.

Das in dieser Projektgruppe erstellte Lernkonzept ist für die Veranschaulichung des Konzeptes „Programming by Example“ sehr gut geeignet. Über die Leistungsfähigkeit dieses Konzeptes könnte eine Gegenüberstellung der drei Verfahren: „Programming by Example“, Programmieren mit 3PL und Programmieren in ST von Interesse sein. Diese Evaluation war in dieser Projektgruppe zeitlich nicht mehr möglich.

Für eine Weiterentwicklung von „Programming by Example“ für Delta-Roboter sollten die realen Einsatzszenarien von Delta-Robotern genauer untersucht werden. Hier erwiesen sich die Gespräche mit den Mitarbeitern der Firma OMRON, welche im Rahmen dieser Projektgruppe geführt worden, als aufschlussreich. In einem Szenario könnte ein Delta-Roboter die Gegenstände auf dem Fließband stapeln, während ein weiterer Delta-Roboter die gestapelten Gegenstände mit einem Zugriff komplett vom Fließband abgreift und verpackt. Weitere Szenarien könnten sich durch Delta-Roboter mit unterschiedlichen Effektoren (Arbeitswerkzeugen) ergeben. Da zum Zeitpunkt, als die Gespräche mit den Mitarbeitern der Firma OMRON geführt wurden, die Entwicklung bereits sehr weit Fortgeschritten war, konnten die oben genannten Hinweise nicht mehr in diesem Projekt berücksichtigt werden.

Als Fazit dieser Projektgruppe wurde nachgewiesen das „Programming by Example“ für Delta-Roboter umgesetzt werden kann. Die in diesem Kapitel diskutierten offen Fragen zeigen jedoch auch, dass weiterer Forschungsbedarf bezüglich der Einsatzmöglichkeiten, der Weiterentwicklung und der Wettbewerbsfähigkeit des „Programming by Example“besteht.

Abbildungsverzeichnis

Literaturverzeichnis

[1] Coppelia Robotics. Virtual Robot Experimentation Platform USER MANUAL. http://www.coppeliarobotics.com/helpFiles/index.html. Zuletzt besucht am 18. September 2014.

[2] Ierusalimschy, R. *Programmieren in Lua*, 3 ed. Open Source Press, 2013.

[3] International Electrotechnical Commission. International Electrotechnical Commission. http://www.iec.ch/. zuletzt besucht am 18. Spetember 2014.

[4] Kern-Isberner, G. Darstellung, Verarbeitung und Erwerb von Wissen, 2012.

[5] Lieberman, H. *Your wish is my command: Programming by example*. Morgan Kaufmann, 2001.

[6] Naujokat, S., Lybecait, M., Kopetzki, D., and Steffen, B. CINCO: A Simplicity-Driven Approach to Full Generation of Domain-Specific Graphical Modeling Tools. to appear.

[7] Naujokat, S., Lybecait, M., and Steffen, B. Projektgruppenantrag - inprox: Industrial programming by example. Technische Universität Dortmund, Faktultät Informatik, Lehrstuhl 5, 2014.

[8] Naujokat, S., Traonouez, L.-M., Isberner, M., Steffen, B., and Legay, A. Domain-Specific Code Generator Modeling: A Case Study for Multi-faceted Concurrent Systems. In *Proc. of the 6th Int. Symp. on Leveraging Applications of Formal Methods, Verification and Validation, Part I (ISoLA 2014)* (2014), no. 8802 in LNCS, Springer, pp. 463–480.

[9] Omron Electronics LL. Sysmac studio. http://industrial.omron.eu/en/products/catalogue/motion_and_drives/machine_automation_controllers/software/sysmac_studio/default.html. zuletzt besucht am 18. September 2014.

[10] Omron Electronics LLC. FA Communications Software CX-Compolet / SYSMAC Gateway Flexible & High Speed PL C-Accessing Softwares, 2012.

[11] RapidMiner GmbH. RapidMiner, a tool for Machine Learning and Data-Mining. https://rapidminer.com/. zuletzt besucht am 13. Februar 2015.

[12] Steffen, B., Margaria, T., Nagel, R., Jörges, S., and Kubczak, C. Model-driven-development with the jabc, 2007.

[13] The Eclipse Foundation. Xtend Documentation @ONLINE. http://www.eclipse.org/xtend/documentation.html, Oct. 2014. zuletzt besucht am 13. Februar 2015.

[14] Universität Passau. Struktur und Implementierung von Programmiersprachen I. http://www.infosun.fim.uni-passau.de/cl/lehre/sips1-ss06/folien/10.pdf, 2006. zuletzt besucht am 12. März 2015.

[15] Wiese, H. remoteapinetwrapper. https://github.com/SeveQ/remoteApiNETWrapper. zuletzt besucht am 17. Oktober 2014.

[16] Xtext. https://eclipse.org/Xtext/. Zuletzt besucht am 14. Mai 2015.