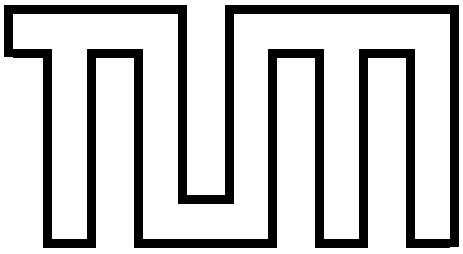


Technische Universität

ERROR: Das Ausführen von Systembefehlen ist deaktiviert (siehe Einstellungen)

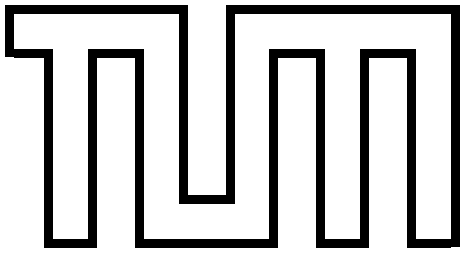
München

Institut für Werkzeugmaschinen und Betriebswissenschaften

Interdisziplinäres Projekt

A Post-Processor for industrial multifunctional robots for a task-oriented automation process

Thomas Schultz



Technische Universität München

Institut für Werkzeugmaschinen und Betriebswissenschaften

Interdisziplinäres Projekt

A Post-Processor for industrial multifunctional robots for a task-oriented automation process

Ein Post-Prozessor für Multifunktions-Industrieroboter innerhalb eines aufgabenorientierte Automatisierungsprozesses

Author: Thomas Schultz

Supervisor: Dipl.-Ing. Julian Backhaus

Submission: DD.MM.2015

Inhaltsverzeichnis

1 Einleitung **3**
- 1.1 Ausgangssituation . . . 3
- 1.2 Zielsetzung . . . 3
- 1.3 Post-Prozessor . . . 4

2 Modellierung **5**
- 2.1 Das Blackboard . . . 6
  - 2.1.1 Der AutomationML Standard . . . 6
  - 2.1.2 Die Blackboard-Struktur . . . 7
  - 2.1.3 Ressourcen . . . 7
  - 2.1.4 Skills . . . 8
  - 2.1.5 Funktionen . . . 9
- 2.2 Das SFC-Format . . . 13
  - 2.2.1 SFC-Objekte . . . 13
  - 2.2.2 Skill-Steps . . . 14
  - 2.2.3 Transitionen . . . 15
- 2.3 IRL-Dateien . . . 16
  - 2.3.1 Funktionsdefinition . . . 16
  - 2.3.2 Funktionsaufruf . . . 16
  - 2.3.3 Erweiterungen . . . 17
- 2.4 Mapping . . . 18
  - 2.4.1 SFC-Mapping . . . 18
  - 2.4.2 Funktions-Mapping . . . 19

3 Implementierung **22**
- 3.1 Klassen . . . 23
  - 3.1.1 aml-Namespace . . . 23
  - 3.1.2 sfc-Namespace . . . 24
  - 3.1.3 process-Namespace . . . 25
- 3.2 Entwicklungsumgebung . . . 26
- 3.3 Einschränkungen . . . 26
  - 3.3.1 Namensgebung . . . 26
  - 3.3.2 Dateipfade . . . 27

3.3.3 Groß- und Kleinschreibung 27
3.4 Optionen 28

4 Zusammenfassung **29**

Abschnitt 1

Einleitung

1.1 Ausgangssituation

Global wirkende Veränderungen wie steigende Variantenvielfalt, kürzere Produktlebenszyklen und steigender Kostendruck, zwingen Firmen ihre Entwicklungsprozesse und ihre Produktion darauf einzustellen. Um diesen gerecht zu werden, wurden in der Vergangenheit verschiedene Ansätze entwickelt, die den Entwicklungsprozess und die Produktion effizienter und flexibler zu gestalten. Vor allem die Anpassung von automatisierten Systemen und die dazu notwendige Anpassung der Konfiguration und der Programmierung schränkt deren Flexibilität ein. Zur Lösung der Problematik, wurden in der Vergangenheit mannigfache Konzepte entwickelt. Eine Möglichkeit die Programmierung effizienter zu gestalten, stellt die aufgabenorientierte Programmierung dar. Am iwb wird aktuell ein aufgabenorientiertes Programmiersystem entwickelt, mit dem, unter Berücksichtigung der industriellen Anforderungen, eine Programmierung auch durch Nichtexperten ermöglicht wird.

1.2 Zielsetzung

In mehreren studentischen Arbeiten wurden die grundsätzlichen Modellierungskonzepte als auch Algorithmen für Teilfunktionen des flexibel einsetzbaren aufgabenorientierten Programmiersystems erarbeitet. In einem interdisziplinären Projekt soll der aktuelle Stand der Prototypen für die Programmierung bis zur Einsetzbarkeit vorangetrieben werden. Dazu sind Arbeiten in verschiedenen Themengebieten notwendig, die über

Schnittstellen zusammenhängen, aber kein direkt überschneidenden Inhalte haben.
Die Generierung des herstellerspezifischen Codes (Postprozessor) z. B. für einen KUKA Roboter, wurde während der Konzeption in der Modellierung vor gedacht, jedoch nur als kleines Beispiel in Python umgesetzt. Der Test des erzeugten Robotersteuerungscodes mit einer virtuellen Robotersteuerung ist im System aktuell nicht möglich.
Im Rahmen dieser Arbeit soll daher das Konzept des Postprozessors überarbeitet und das Konzept anschließend implementiert werden. Für die Umsetzung soll dabei auf C++ zurückgegriffen werden. Als Eingangsgrößen stehen zwei XML-Dateien zur Verfügung, eine abstrakte Beschreibung des Steuerungscodes und eine Verknüpfung der abstrakten Beschreibung mit den herstellerspezifischen Befehlen. Zum Test des Steuerungscodes stehen verschiedene Simulationssysteme wie KUKA.Sim oder Prozesssimulate am iwb zur Verfügung, welche über eine API angesprochen werden müssen.

1.3 Post-Prozessor

Im Rahmen dieses Projekts ist ein funktionsfähiger PostProzessor implementiert worden, der aus den Eingangsinformationen, der SFC-Datei und dem Blackboard, ein funktionsfähiges KUKA-KRL Programm erzeugt. Dieser wurde in GNU C++ ohne Microsoft .NET realisiert und ist somit sowohl auf Windows und Linux lauffähig. Es wurde als eigenständiges Programm ohne Abhängigkeiten von vorhanden Bibliotheken erzeugt und wird über die Kommando-Zeile aufgerufen. Der Quellcode ist offen und darf von Fremden verwendet oder verändert werden. Im Rahmen der Evaluierung sind zusätzliche Validierungsfunktion integriert worden, um Eingabefehler zu erkennen.

Abschnitt 2

Modellierung

Durch vorangegangene Arbeiten wurde ein Modellierungskonzept entwickelt um eine weitreichende Automatisierung zu ermöglichen. Am Anfang steht dabei das gewünschte Produkt und das Zielsystem für die Produktion. Die Beschreibung des Zielsystem erfolgt in AutomationML und wird in dem Blackboard (siehe 2.1) gespeichert. Dieses wird nur einmal pro Zielsystem teil-automatisiert erstellt und dient für die Erzeugung alle Produkte auf diesen System.

Nach Spezifizierung des Produkt wird es anschließend schrittweise in der Planungsphase verarbeitet. Hier kommen je nach Anforderung mehrere Module zum Einsatz, wichtige Beispiele wären hierfür Bahnplanung und Schweißplanung. Für das Produkt wird eine eigene Datenstruktur in Form einer SFC-Datei (siehe 2.2) erstellt. Jedes der Module hat sowohl Zugriff auf die SFC-Datei, als auch auf das Blackboard. Nach Durchlauf eines Moduls schreibt dieses seine Änderungen in die SFC-Datei. In Abbildung 2.1 ist das bestehende Konzept im und die Schnittstelle für den Post-Prozessor dargestellt.

Da der Post-Prozessor das letztes Modul in der Automatisierungskette darstellt, erhält diese eine ausgeplante SFC-Datei. Im Rahmen des bisherigen Konzepts mussten kleine Anpassungen (siehe 2.4) vorgenommen werden, sodass dem Post-Prozessor alle nötigen Informationen zur Verfügung stehen.

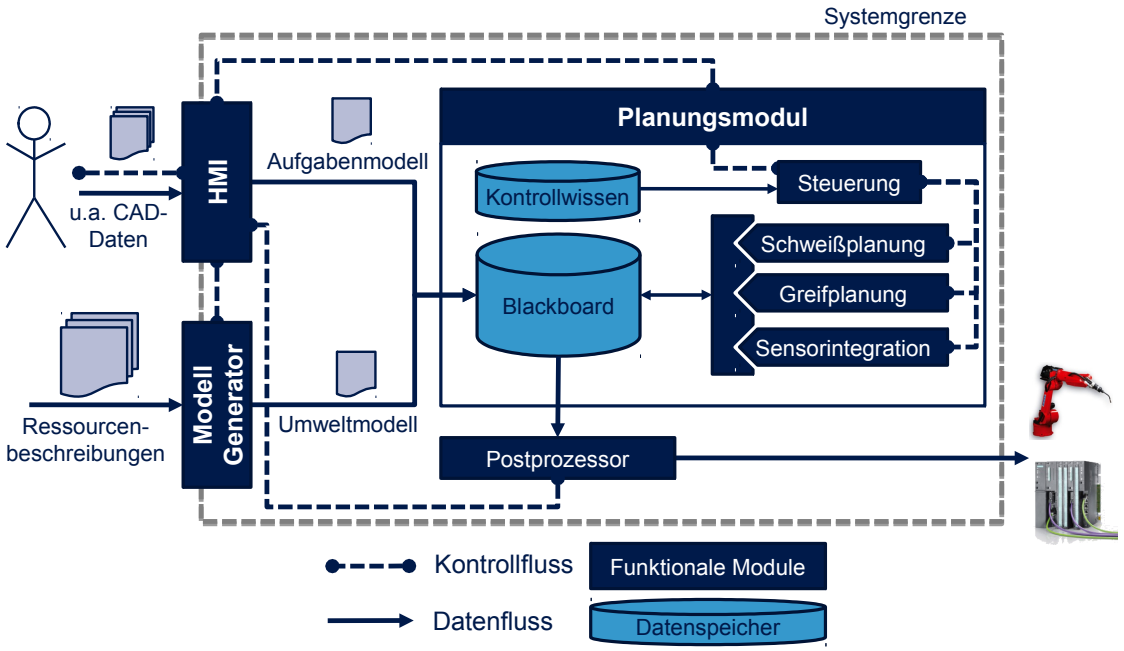


Abbildung 2.1: Automatisierungskonzept von User-Eingabe bis Maschinenprogramm

2.1 Das Blackboard

Das Blackboard basiert auf dem AutomationML Standard und dient als Container für die Eingangsgrößen sowie für Informationen die während der Planung erzeugt werden.

2.1.1 Der AutomationML Standard

Der Begriff steht für Automation Markup Language und ist eine XML basierte Datenstruktur. Es ist ein offener Standard und wird von dem AutomationML e.V und dessen Mitgliedern, darunter Daimler ABB, KUKA, Rockwell Automation und Siemens, betrieben und weiterentwickelt[1]. Innerhalb einer AML-Datei können komplexe herstellerübergreifende anlagenspezifische Informationen gespeichert werden. Dadurch soll das Zusammenspiel von heterogenen Strukturen in der Industrie erleichtert werden.

Zur Erstellung und Editieren von AML-Dateien gibt es einen eigenen Editor, der auf der Website herunterladen werden kann. Zusätzlich gibt es im Downloadbereich vorkompilierte Programm-APIs auf Basis von Microsoft .NET DLL-Dateien. Da der Post-Prozessor jedoch platformunabhängig konzipiert wurde, kamen diese DLLs nicht zur

[1] AutomationML e.V `https://www.automationml.org/o.red.c/home.html`

Verwendung. In den späteren Abbildungen zur Veranschaulichung werden Ausschnitte des Editors zu sehen sein. Die Informationen zu AML-spezifischen Punkten in dieser Arbeit wird sich auf das nötige Minimum beschränken, weitere Details können auf der Website und den diversen Whitepapern nachgelesen werden.

Das Blackboard ist die wichtigste Datenstruktur für den Post-Prozessor. Sie vereint alle nötigen Informationen über die Roboterzelle und dessen Fähigkeiten. Sie unterteilt sich in verschiedene Elemente. Im wesentlichen für den Post-Prozessor sind folgende Komponenten: Ressourcen, Fähigkeiten und Funktionen. Eine Ressource entspricht einer Abbildung eines realen Gerätes. Dabei besitzt jedes Gerät Fähigkeiten, im weiteren nur noch Skills genannt, die das Gerät unterstützt. Eine Funktion ist eine genau Anweisung des Gerät und stellt in der Modellierung den kleinsten Baustein dar.

2.1.2 Die Blackboard-Struktur

Das Blackboard besitzt neben der *Instanzhierarchie*, wie in Abbildung 2.2 zu sehen ist, drei weitere Elemente, die für den Post-Prozessor nicht von Belang sind und deshalb hier nicht näher erklärt werden. Im Abschnitt *Process Sequence* wird die Position des SFC-Files als relativer Pfad bezogen auf die Position des Blackboards hinterlegt. Nach einem erfolgreichen Durchlauf des Post-Prozessors wird der Pfad der Ausgabedatei unter *Code* gespeichert.

2.1.3 Ressourcen

Ressourcen sind die oberste Hierarchiestufe innerhalb einer Blackboard-Struktur. In einer Datei dürfen beliebig viele Einträge existieren, solange der Name unterscheidbar ist. Jede Ressource besitzt mehrere untergeordnete Elemente. Für den Post-Prozessor sind nur die Skills und die Funktionen, im Blackboard unter den Namen *FunctionDescription* zu finden, von Bedeutung.

Wie in Abbildung 2.3 zu sehen ist, beinhaltet eine Ressource noch weiter Elemente neben den *Skills* und den *FunctionDescriptions*. Pro Ressource werden alle Skills und Funktionen ausgelesen. Um sie eindeutig zuzuordnen, wird jeweils der Name verwendet. Daraus folgt, dass pro Ressource keine doppelten Skill-Namen und Funktionen definiert sein dürfen. Im Regelfall gilt das aber sowieso zu Vermeiden, um die Lesbarkeit zu erhalten.

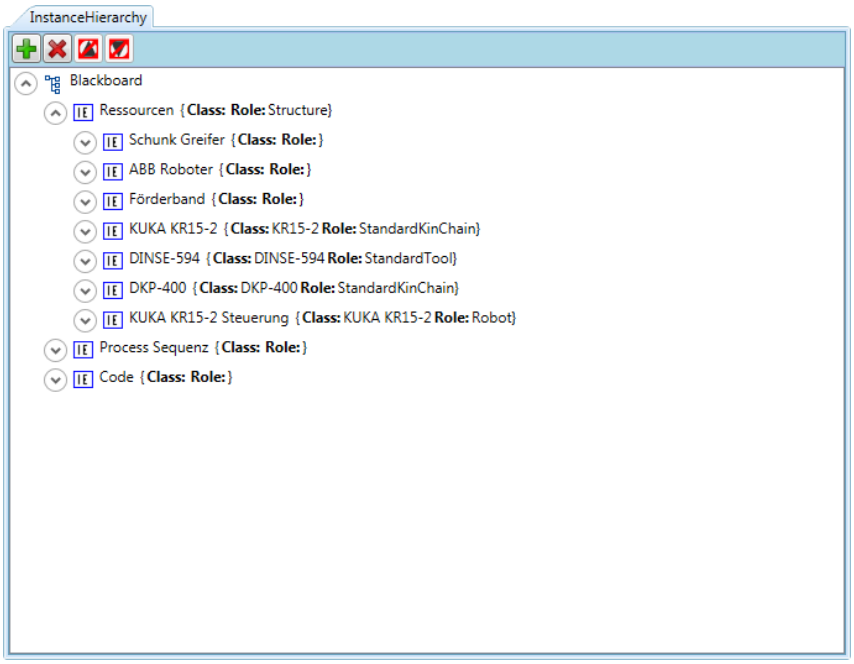


Abbildung 2.2: Blackboard mit unterschiedlichen Ressourcen

2.1.4 Skills

Ein Skill im Blackboard ist das äquivalent zu einem Skill-Schritt in der SFC-Struktur (dazu später mehr). Er beschreibt eine abstrakte Fähigkeit der speziellen Ressource bzw. Roboters. Ein Skill stellt eine oder mehrere Funktionen wie eine Schnittstelle als Fähigkeit nach außen dar.

Innerhalb des Skills gibt es sogenannte Interfaces. Abbildung 2.4 zeigt einen *Move PTP* Skill mit zwei Interfaces. Diese Interfaces sind für das Verknüpfung von Skill zu Funktionen wichtig. Das Mapping spielt für den Post-Prozessor eine wichtige Rolle und wird in Abschnitt 2.4.1 genauer erklärt. Da ein Skill eine abstrakte Fähigkeit der Ressource darstellt, greift sie auf eine oder mehrere Funktionen zu, wobei jede Funktion ihr eigens Interface benötigt.

Neben den Elementen und Interfaces besitzt ein Skill auch noch Attribute. Diese werden im AML-Editor in einem extra Fenster dargestellt, wie es in Abbildung 2.5 zu sehen ist. Die Attribute eines Skills definieren zum einen, welche Eingangsgrößen der Skill benötigt und zum anderen, wo dieser Wert in einem SFC-Schritt zu finden ist. Auch diese Mapping wird gesondert in Abschnitt 2.4.1 behandelt. Besitzt ein solches Attribut Unterattribute, so wird intern ein globaler Name zugewiesen, der sich aus übergeordneten plus untergeordneten Namen zusammensetzt. Getrennt werden beide Namen durch einen

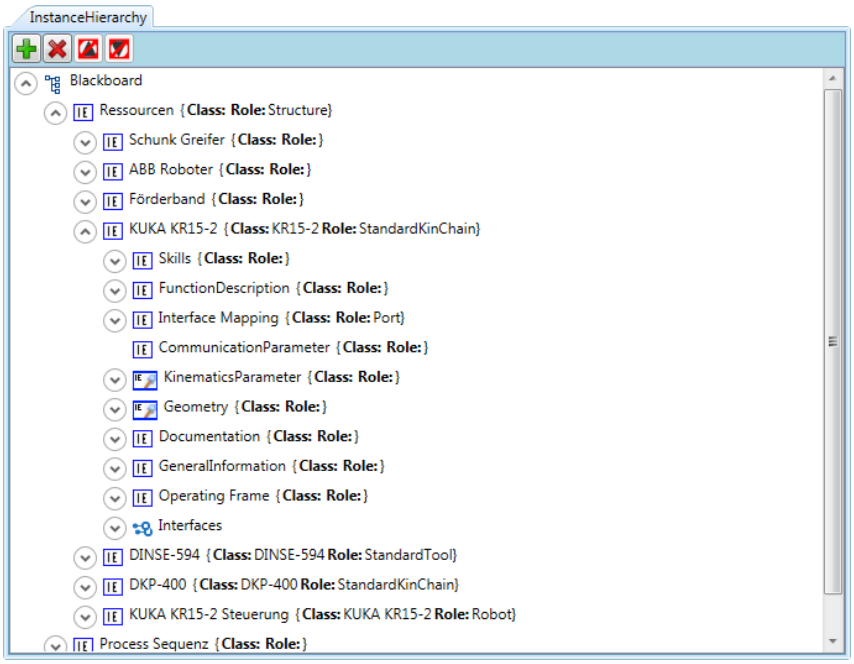


Abbildung 2.3: Inhalt einer Ressource

Punkt. Der *Move PTP* Skill besitzt die Größen *Velocity* und *End Position*, welches mehrere Unterattribute besitzt. Dadurch ergeben sich folgende globale Parameternamen: *Velocity*, *End Position.x*, *End Position.y* und *End Position.z*.

In der Theorie erlaubt dieses System beliebig tiefe Verschachtlungen. Pro Ebene müsste die Datenstruktur einer Variable eines SFC-Schritts um eine Dimension wachsen. Während ein einfache Parameter auf einen einzelnen Wert abbildet, entspricht ein einmal verschachtelter Parameter einer Liste. Dreimal verschachtelt würde somit einer zweidimensionalen Tabelle entsprechen. Aktuell sind allerdings nur Variablen als Wert oder als Liste vorgesehen.

2.1.5 Funktionen

Funktonen stellen die direkte Schnittstelle zum Gerät oder Roboter selber dar. Sie beschreiben wie ein einzelner Befehl des Roboters aussieht. Auch Funktionen besitzen Attribute, die analog zu denen des Skills, ihre nötigen Eingangsgrößen beschreiben. Abbildung 2.7 zeigt wie die Attribute einer Funktion auszusehen haben. Des weiteren besitzt jede Funktion genau ein Interface vom Typ *FunctionLogic*, in der Regel wird es *FunctionDataInterface* genannt, welches zwei Parameter besitzt: *Name* und *refURI*. Dabei steht der Name für den Befehl innerhalb der verwendeten Programmiersprache. Der andere

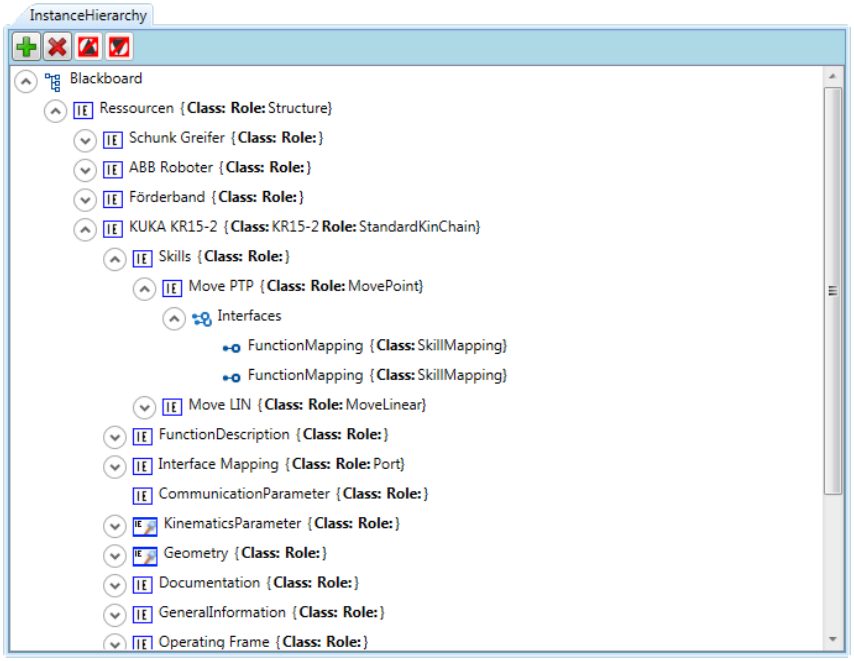


Abbildung 2.4: Aufbau eines Skills

verweist mittels Pfad zu einer Datei, in denen die jeweilige Funktionsdefinition und der Funktionsaufruf steht. Diese Daten werden für das Zuordnung (siehe 2.4.2) verwendet.

Es ist möglich für jede Funktion eine eigene Datei zu referenzieren. Zwecks der Übersichtlichkeit bietet es sich aber an, pro Ressource eine solche Datei anzulegen, in der alle Funktionen enthalten sind. Der genaue Syntax dieser Dateien ist in Abschnitt 2.3 erklärt.

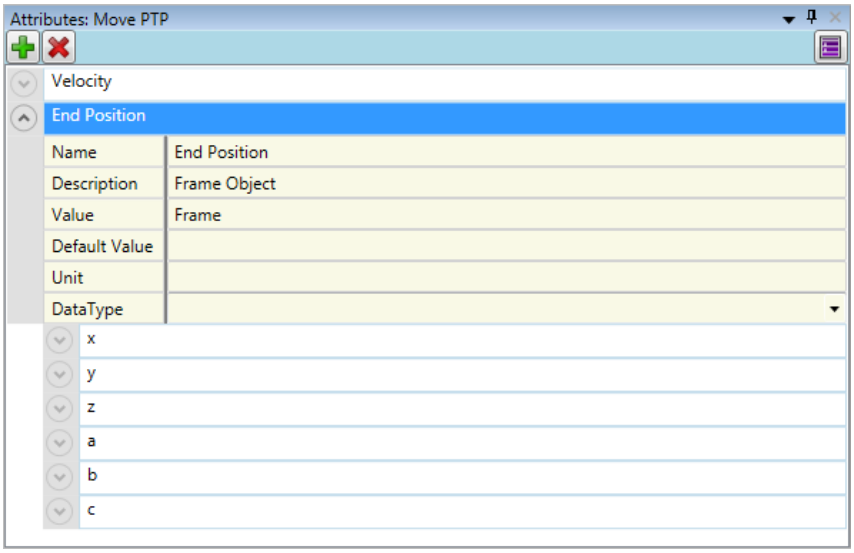


Abbildung 2.5: Attribute des Skill Move PTP

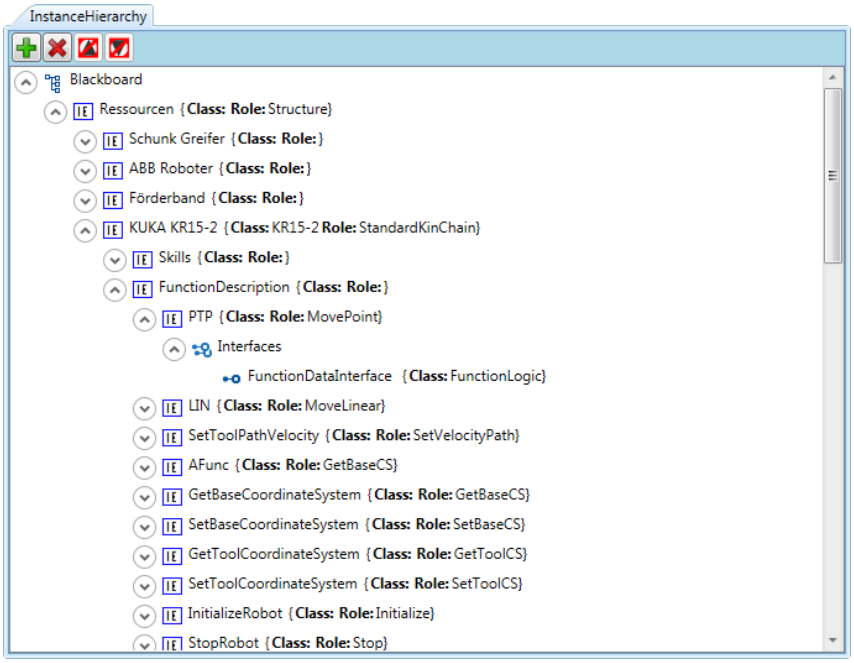


Abbildung 2.6: Aufbau einer Funktion

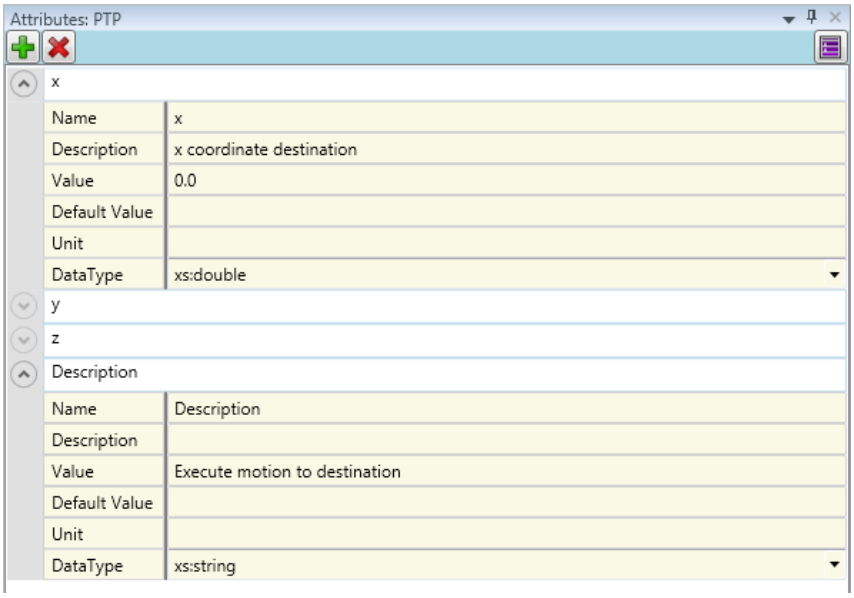


Abbildung 2.7: Attribute der Funktion am Beipsiel von PTP

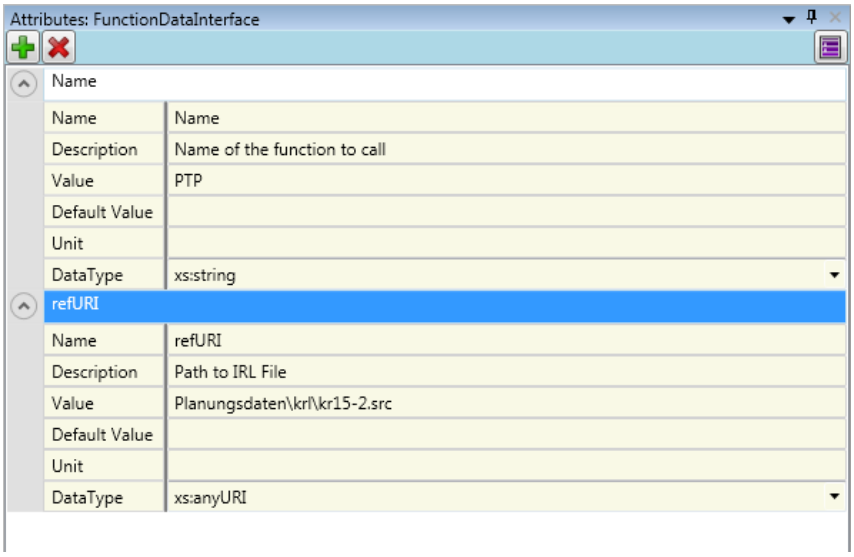


Abbildung 2.8: Attribute und deren Werte eines FunctionDataInterfaces

2.2 Das SFC-Format

SFC steht für Sequential Function Chart und ist eine Programmsprache basierend auf der IEC 61131 Norm[2]. Die SFC-Datei dient hier zur Speicherung und Verarbeitung von Prozessschritten. Sie liegt ebenfalls in einer XML-Struktur vor und beinhaltet alles Schritte zur Erzeugung des Produkts. Für diese Arbeit ist ausschließlich der Programm-Abschnitt innerhalb der Datei von Bedeutung, weswegen nur auf diesen eingegangen wird. In einem Programm-Block sind die einzelnen Arbeitsschritte hinterlegt. Im Prinzip kann eine SFC-Datei auch mehrere Programm-Blöcke enthalten, jedoch wurde dieser Fall nicht in der Aufgabenstellung vorhergesehen. Der Post-Prozessor verarbeitet immer den ersten Programm-Block.

Das Programm selber liegt in Form eine Flussdiagramms, vergleichbar mit einem Vorranggraph für die Montage, vor. Die Knoten stellen dabei Schritte, die Kanten Übergänge zwischen den Schritten dar. Jedes Programm besteht primär aus einer Abfolge von Prozess-Schritten (Process-Steps). Dies Prozess-Schritte werden im Laufe der Planungsphase in speziellere Fähigkeits-Schritte (Skill-Steps) zerlegt. Dadurch wird aus einer abstrakten Arbeitsanweisung ein konkreter Ablauf von unterschiedlichen Tätigkeiten.

Die Aufgabe des Post-Prozessors ist es diese Schritte, nachdem alle Prozess-Schritte ausgeplant wurden, in Maschinen-Code zu übersetzen, der von den verwendeten Robotern verstanden und ausgeführt werden kann. Die SFC-Datei beschreibt dabei das Produkt, bzw. dessen Herstellungsschritte. Die Position der Datei wird im Blackboard hinterlegt.

2.2.1 SFC-Objekte

Der Aufbau eines SFC-Objekt ist nach einem festen Schema definiert. Jedes Objekt besitzt eine eindeutige Identifikationsnummer und einen Namen. Weiterhin gibt es Verbindungspunkte für Ein- und Ausgänge. Dabei besitzen nur Eingänge eine Referenz auf das angeschlossene Objekt. Dafür wird die localId des Objekts als *refLocalId* gespeichert. Die Orts- und Positionsangaben an diesem Elementen beziehen sich auf die graphische Beschreibung des SFC und sind für diese Arbeit nicht von Belang.

In Auflistung 2.1 ist ein einfaches SFC-Objekt dargestellt. An dessen Eingangspunkt ist das Objekt mit der ID 42 verbunden. Es hat zwar einen Ausgangsverbindung, jedoch ist das Speichern des mit diesem verknüpften Objekts nicht möglich. Von einem beliebigen

[2]PLCopen Organsitation: `http://www.plcopen.org/pages/tc1_standards/iec_61131_3/`

```
<step localId="43" name="Deep Thought" height="23" width="38" xmlns="">
  <connectionPointIn>
        <relPosition x="40" y="0" />
        <connection refLocalId="42">
          <position x="362" y="383" />
          <position x="362" y="346" />
        </connection>
  </connectionPointIn>
  <connectionPointOut formalParameter="">
        <relPosition x="0" y="1" />
  </connectionPointOut>
  <addData>
   <data name="iwb" handleUnknown="preserve">
         <iwb>
           <time duration= "7.5 million years"/>
         </iwb>
  </data>
</step>
```

Auflistung 2.1: XML Beispiel eines SFC-Objekts

Objekt können so nur die Vorgänger ausgelesen werden, nie die Nachfolger. Für zusätzliche Informationen ist ein ein *addData* Block erlaubt, in dem beliebig Informationen angehängt werden können. In diesem Beispiel ist eine Zeitangabe gespeichert.

2.2.2 Skill-Steps

Skill-Steps sind eine der zwei Klassen von SFC-Objekten. Sie besitzen alle Eigenschaften eine SFC-Objekts, erweitern dieses aber um ein paar Informationen. So besitzt ein Skill-Step ein zusätzliches boolsches Attribute namens initialStep, welches bei Start- und Endknoten eines Graphen gesetzt ist. Um wichtige Daten innerhalb eines Schrittes zu hinterlegen gibt es Variablen. Wenn hier ein Werte gespeichert wird, ist dieser ohne zusätzliche Zeichen angegeben. Es besteht jedoch auch die Möglichkeit eine Liste von Daten anzugeben. Zusammenhängende Werte werden in runden Klammern geschrieben und durch Komma getrennt. Diese Schreibweise lässt auch mehrdimensionale Daten wie in Abschnitt 2.1.4 beschrieben zu, wird aber noch von keinem Modul unterstützt.

Auflistung 2.2 zeigt ein vollständigen Skill-Step mit angedeuteten Variablenwerte. Skill-Steps dürfen immer nur einen Ein- und Ausgang besitzen. Wird diese Bedingung verletzt, kann der PostProcessor nicht richtig arbeiten. Die Anzahl der Variablen ist indes unbegrenzt. Eine Variable definiert sich im SFC-Format ausschließlich über den Namen. Informationen über Datentypen oder Anzahl an Werten sind nicht vorgesehen.

```
<step localId="0" name="P2P" initialStep="false" height="23" width="38">
<connectionPointIn>
  <relPosition x="40" y="0"/>
  <connection refLocalId="0">
        <position x="362" y="383"/>
        <position x="362" y="346"/>
  </connection>
</connectionPointIn>
<connectionPointOut formalParameter="">
  <relPosition x="0" y="1"/>
</connectionPointOut>
</addData>
<Variable name="Frame" Value="( X, Y ,Z)">
</Variable>
<Variable name="Velocity" Value="v">
</Variable>
<Variable name="JointConfig" Value="( a, b, c)">
</Variable>
</step>
```

Auflistung 2.2: XML Beispiel eines Punkt-zu-Punkt Schrittes

2.2.3 Transitionen

Transitionen stellen die zweite Klasse an SFC-Objekten dar. Sie sind nötig, um einzelne Skill-Steps zu einem Graphen zu verbinden. Dabei gilt die Regel, dass zwischen zwei Skill-Steps immer mindestens eine Transition vorhanden sein muss. Wird diese Bedingung verletzt, wird der PostProcessor vorzeitig mit einer Fehlermeldung abbrechen. Transitionen sind logisch in drei Varianten unterteilt: einfache 1-zu-1 Transitionen, 1-zu-N Divergencen und N-zu-1 Convergencen. Die vierte Möglichkeit von N-zu-M ist verboten und führt zu Fehlermeldungen. Die einzelnen Varianten unterscheiden sich nur in der Anzahl an Ein- bzw. Ausgängen.

```
<transition localId="0" height="2" width="20">
  <position x="300" y="10"/>
  <connectionPointIn>
        <relPosition x="10" y="0"/>
        <connection refLocalId="0">
          <position x="342" y="122"/>
          <position x="342" y="101"/>
        </connection>
  </connectionPointIn>
  <connectionPointOut>
        <relPosition x="10" y="2"/>
  </connectionPointOut>
</transition>
```

Auflistung 2.3: XML Beispiel einer 1-zu-1 Transition

In Auflistung 3.3 ist eine einfache 1-zu-1 Transition beschrieben. Obwohl Divergencen

und Convergencen jeweils eigene Objekte in der SFC-Datei sind, werden sie in der Modellierung für den Post-Prozessor zu dem Transitionen gezählt. Diese Abstraktion ist möglich, da der Post-Prozessor nur die Skill-Steps zu Code-Generierung nutzt und alle Transitionen ausschließlich zur logischen Verknüpfung dienen. Weiterhin ist es irrelevant wie viele Transitionen jeglicher Art zwischen zwei Skill-Steps liegen, solange es eine gültigen Verbindung mit definierten Anfangs- und Endknoten ist.

2.3 IRL-Dateien

IRL steht für Industrial Robot Language und eine herstellerunabhängige Programmiersprache für Industrieroboter. Sie ist nach DIN 66312 definiert und ähnlich üblicher High-Level Programmiersprachen aufgebaut. Die IRL-Dateien dienen zur Speicherung von Textbausteinen mit Programm-Code. Da der Post-Prozessor mit verschiedenen Programmiersprachen umgehen können soll, sind die Funktionen in diesen Format abzulegen. Der Unterschied zwischen IRL und KUKA KRL fällt jedoch sehr gering aus und da leider kein anderes Robotersystem für die Erprobung zu Verfügung stand, wird in der aktuellen Version noch KRL-Code verwendet.

IRL-Dateien werden in XML-Format gespeichert und müssen wie in Auflistung 2.4 aufgebaut sein. Unterteilt sind sie in drei Abschnitte: *info*, *def* und *src*. Während das Info-Feld nur für den Betrachter zur Lesbarkeit dient, müssen die beiden anderen vorhanden sein. In ihnen werden die Funktionsdefiniton und die -Aufrüfe beschrieben.

2.3.1 Funktionsdefinition

Eine Funktionsdefinition ist immer dann nötig, wenn eine Befehl mehrere Anweisung ausführen muss. Dadurch lassen sich Bausteine erstellen die beliebig viele Parameter und Anweisungen besitzen. Um die Funktion dann aufzurufen, benötigt es nur einen einzigen Funktionsaufruf. Bei einzelnen Anweisungen ist die Definition leer, oder ganz wegzulassen.

2.3.2 Funktionsaufruf

Ein Funktionsaufruf besteht aus einem Funktionsnamen und beliebiger Anzahl von Parametern. Diese sind jeweils separat als XML-Attribute anzugeben. Der eigentliche Aufruf wird dann direkt als Wert des Elements als Textbaustein angegeben. In Auflistung 2.4 seht man jeweils wie ein Funktionsaufruf auszusehen hat. Bei Programmiersprachen die

```
<irl>
        <info name="KUKA KR15-2"/>
        <def>
                <definition name="SetVelCP">
                        <parameter name="VALUE"/>
                        DEF SetVelCP(VALUE:IN)
                                DECL REAL VALUE
                                $VEL.CP = VALUE
                                RETURN
                        END
                </definition>
        </def>
        <src>
                <function-call name="SetVelCP">
                        <parameter name="VALUE"/>
                        SetVelCP (VALUE)
                </function-call>
                <function-call name="PTP">
                        <parameter name="X"/>
                        <parameter name="Y"/>
                        <parameter name="Z"/>
                        <parameter name="A"/>
                        <parameter name="B"/>
                        <parameter name="C"/>
                        PTP {X, Y, Z, A, B, C}
                </function-call>
                <function-call name="LIN">
                        <parameter name="X"/>
                        <parameter name="Y"/>
                        <parameter name="Z"/>
                        <parameter name="A"/>
                        <parameter name="B"/>
                        <parameter name="C"/>
                        LIN {X, Y, Z, A, B, C}
                </function-call>
        </src>
</irl>
```

Auflistung 2.4: Beispiel für IRL-Datei

unterschiedliche Klammerung nutzen, zum Beispiel KUKA KRL, müssen dieses korrekt eingetragen sein. In diesem Fall sind Positionsangaben (*PTP* oder *LIN*) in geschweiften und Funktionsaufrufe (*SetVelCP*) in runden Klammern anzugeben.

2.3.3 Erweiterungen

Die Modellierung in XML ermöglicht das einfache Erweitern des Funktionsumfangs. So lassen sich Funktionen mit gleichen Namen und Unterschiedlicher Anzahl an Parameter implementieren. Zu diesem Zweck hat jeder Funktionsaufruf eine Liste derer. Zusätzlich kann in Zukunft auch hier ein Datentyp spezifiziert werden, der dann ausgewertet werden kann. Diese Optimierung sind für weitergehende Arbeiten ausgelegt, der Post-Prozessor im Rahmen dieser Arbeit unterstützt lediglich das XML-Format mit den Textbausteinen.

Das Auslesen der Parameter erfolgt aktuell noch aus den Textbaustein selber.

2.4 Mapping

Um die Daten zwischen den verschiedenen Repräsentationen zuordnen zu können, braucht es Übersetzungsfunktionen. Für den Post-Prozessor müssen daher zwei Mappings vorliegen, zum einen die Zuordnung von Blackboard-Skill-Parametern zu den Variablen der SFC-Struktur und zum anderen die Zuordnung von Funktions-Parametern zu Skill-Parametern. Zu diesem Zweck werden im Blackboard beide Informationen in dem jeweiligen Skill hinterlegt.

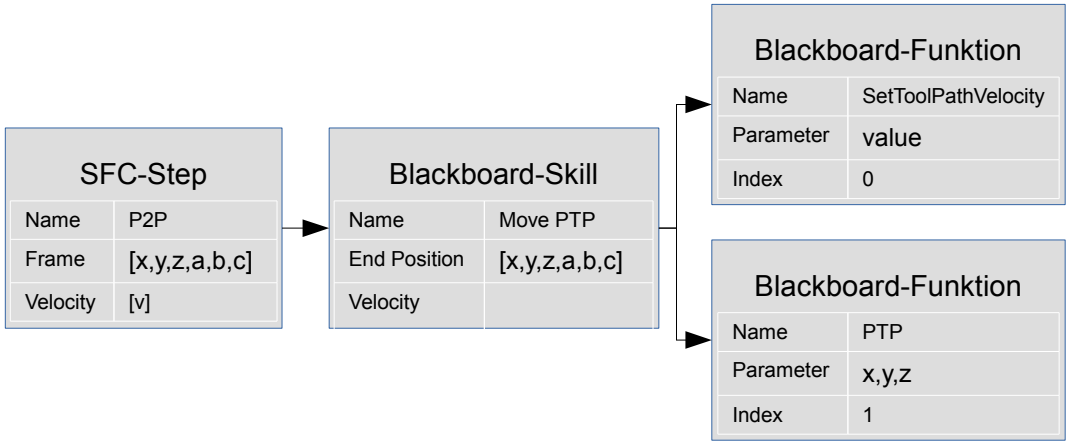


Abbildung 2.9: Überblick des Mapping-Verfahrens

2.4.1 SFC-Mapping

Nach dem Auslesen eines Skills besitzt dieser eine Liste von Parameter in globaler Notation. Um die Werte aus dem SFC-Schritt zu erhalten, benötigt man den entsprechenden Variablen Namen. Im bereits erwähnten Beispiel von *Move PTP* muss so *End Position.x* auf die Variable *Frame* des SFC-Schritts verweisen. Dieser Information wird im Blackboard-Skill im Feld *Value* hinterlegt. Liegt ein verschachtelter Parameter vor, enthält das Attribute auf der höchsten Ebene wie gewohnt den Namen der Variable, während die Unterattribute den Index des Wertes in der Variablenliste enthält.

Wären mehrdimensionale Variablen vorgesehen würde so jede weitere Ebene einen Index in der entsprechenden Dimension entsprechen. Mit diesem System ist die benötigte

Funktionalität bereits hinreichend gegeben und es muss bei Bedarf für mehrdimensionale Datenstrukturen nicht verändert werden.

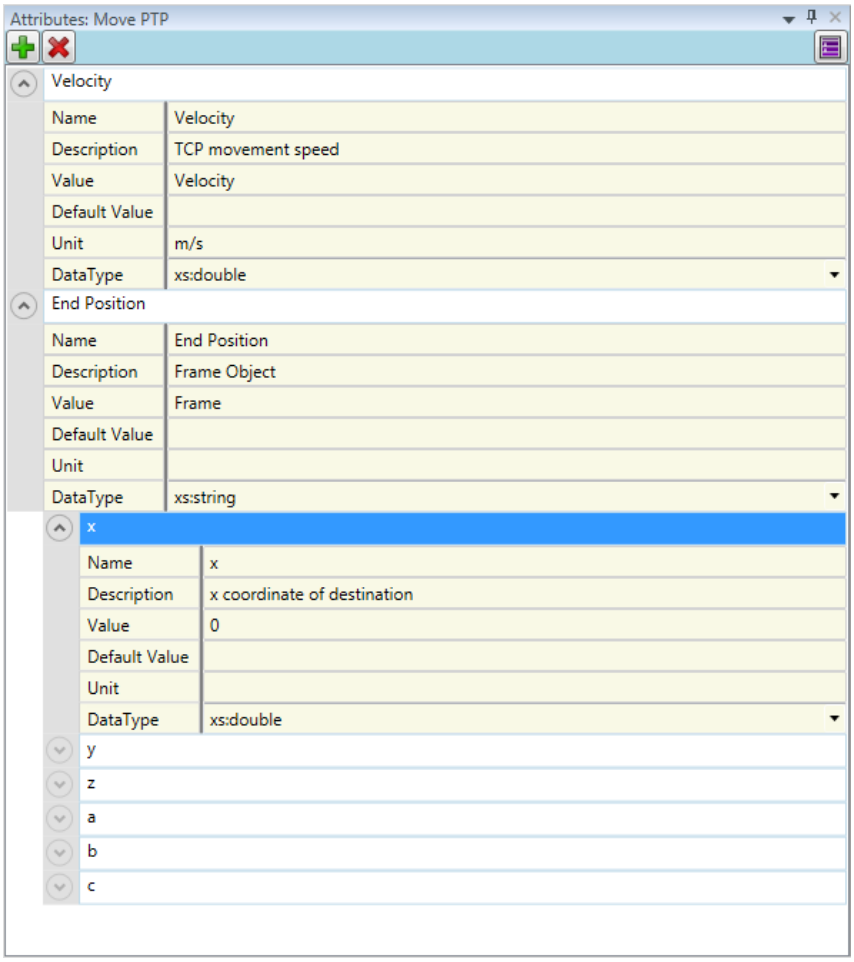


Abbildung 2.10: Beispiel für das SFC-Mapping

Um das Beispiel zu verdeutlichen, ist in Abbildung 2.10 das obige Beispiel gezeigt. *End Position.x* verweist auf das nullte Element innerhalb der Variablenliste *Frame*. Zu Beachten gilt es hierbei, dass die Adressierung bei Null und nicht Eins startet. Auch der Datentyp *xs:double* ist nur bei dem Subattribute x vorhanden, da sich dieser direkt auf das Attribut bezieht. Technisch ist es dadurch möglich innerhalb einer Variablenliste des SFC-Schrittes verschiedene Datentypen zu realisieren.

2.4.2 Funktions-Mapping

Damit Funktionen die richtigen Daten erhalten, müssen die Parameternamen korrekt zugewiesen werden. Dieses Mapping ist nötig, da Funktionen nur für sich

betrachtete Eindeutigkeit aufweisen. Eine Bewegungsfunktion wird in der Regel eine Koordinatenposition mit drei Werten: X,Y und Z erwarten. In einem Skill sind aber möglicherweise mehrere Positionen unter unterschiedlichen Namen, zum Beispiel *End Position* und *Intermediate Position* mit jeweils X,Y und Z-Werten gespeichert.

Um eine Zuordnung zu erreichen, muss pro verknüpfte Funktion ein FunctionMapping-Interface angelegt werden. Dieses beinhaltet drei Informationen: den Namen der Funktion innerhalb der Ressource, eine Index für die Reihenfolge und das Mapping. Bei mehreren Funktionen wird mit der Funktion mit dem kleinsten Index angefangen und in aufsteigenden Reihenfolge fortgefahren. Wird der Wert weggelassen, wird eine implizit Index gleich Null angenommen. Während das bei Skills mit einer einzigen Funktion ausreicht, ist das Ergebnis bei mehreren Funktionen dadurch undefiniert und es muss deshalb ein Wert gesetzt werden.

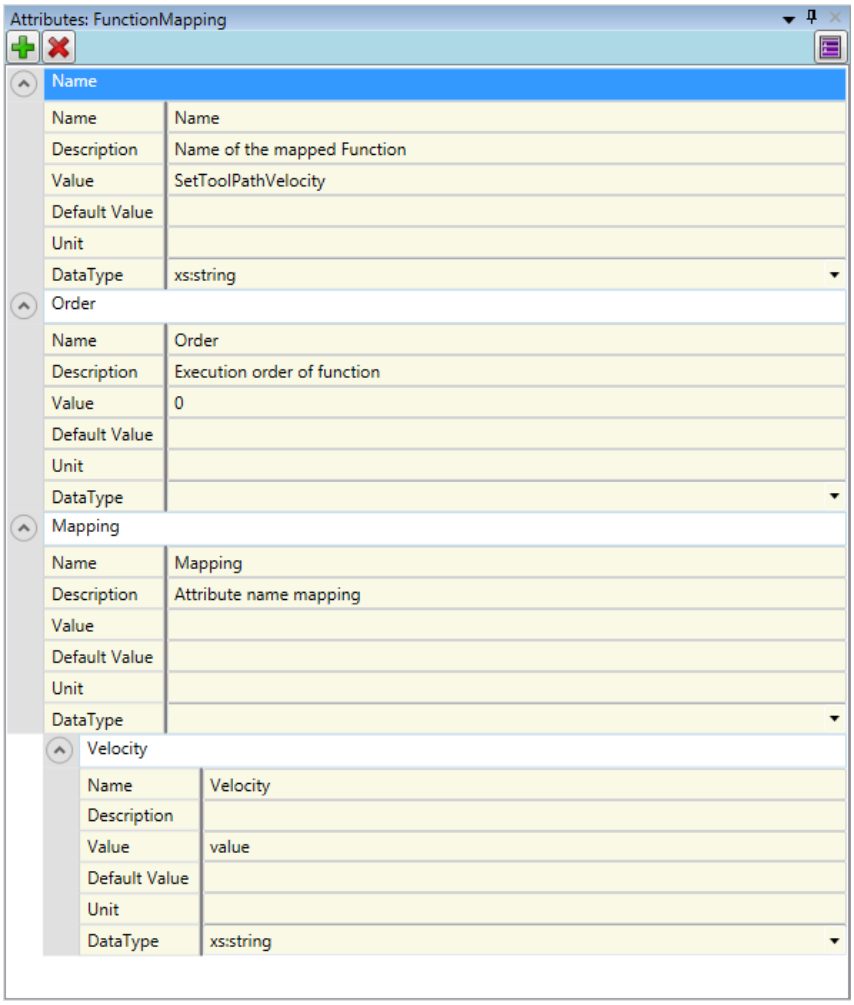


Abbildung 2.11: Beispiel für das Funktions-Mapping

Abbildung 2.11 zeigt eines der Funtions-Mapping des *Move PTP* Befehls im ausgefüllten Zustand. Der Parameter *Velocity* wird auf die Funktion *SetToolPathVelocity* unter dem Namen *value* abgebildet. Die Funktion besitzt einen Index von Null und wird somit als erstes ausgeführt. Danach würde das zweite Mapping die Funktion *PTP* analog zu diesem Beispiel mit den Parametern *End Position.x* etc. aufgerufen werden. Aus diesem Beispiel ist auch zu erkennen warum ein Index Wert nötig ist. Es wäre sinnlos die Bewegungsgeschwindigkeit nach dem Bewegungsbefehl zu setzen.

Abschnitt 3

Implementierung

Der PostProzessor ist als Konsolenanwendung konzipiert. Um den Code für ein Projekt zu generieren, muss ihm die Blackboard-Datei übergeben werden. In ihr muss der Pfad der SFC-Datei hinterlegt sein. Nach der Verarbeitung legt der Post-Prozessor zwei Dateien neben dem Blackboard an: eine .src-Datei und eine .dat-Datei. Der Name für das zu entstellende Programm muss ebenfalls als Parameter beim Aufruf übergeben werden.

```
prompt>PostProcessor.exe Blackboard.aml output
Parsing completed
Programm was written to 'output.src'

prompt>
```

Auflistung 3.1: Ausgabe bei erfolgreicher Aufruf

Auflistung 3.1 zeigt die Ausgabe auf der Konsole, nach erfolgreicher Generierung. Falls die Eingabe nicht korrekt war, wird wie in Auflistung 3.2 eine Hinweis über die richtige Schreibweise ausgegeben. Sollten während des Vorgangs Fehler auftreten, wird dies durch Fehlernachrichten angezeigt. Bei der Generierung von Fehlermeldungen wurde darauf geachtet, möglichst genau anzugeben wo der Fehler aufgetreten ist. Die Meldungen müssen aber nicht zwangsläufig anzeigen wo die Ursache hierfür lag.

```
Usage: PostProcessor [options] <blackboard-file> <program-name>
```

Auflistung 3.2: Ausgabe bei fehlerhaften oder unvollständigen Aufruf

Alle hier genannten Informationen beziehen sich auf Version 1.0.3 des Post-Prozessors. Durch weitere Änderungen Dritter oder mich selber kann es zu Abweichungen in aktuelleren Versionen kommen.

3.1 Klassen

Um eine gute Codequalität zu erhalten und gute Lesbarkeit zu erlauben, ist das Projekt in zahlreiche Klassen unterteilt. Hauptsächlich sind sie in drei Namespaces unterteilt: aml, sfc und process. Zusätzlich gibt es diverse Hilfefunktionen ohne Zugehörigkeit.

Die Quell-Dateien sind entsprechend ihrer Zugehörigkeit in einer Ordnerstruktur abgelegt. Details zu Implementierung sind im Code selber zu finden. Die Informationen in diesem Abschnitt dienen nur dem gröberen Überblick.

3.1.1 aml-Namespace

Alle Klassen, die mit dem Blackboard oder den Daten daraus arbeiten, sind hier zusammengefasst. Die wichtigste Klasse hier ist die *Blackboard* Klasse, die für das Auslesen der AML-Datei und erzeugen der Objektstruktur zuständig ist. In ihr wird für jede gefunden Ressource ein Objekt der Klasse *Resource* angelegt und gespeichert.

Innerhalb des Namespaces gibt es die *Attribute* Klasse. Diese Klasse wird immer da verwendet wenn auf die XML-Struktur der Attribute aus dem Blackboard zugegriffen wird. Alle Felder aus dem Attribute-Fenster des AML-Editors lassen sich somit direkt über Funktionen erhalten. Solange geworfene Exceptions abgefangen werden, kann auch auf nicht vorhandene Attribute zugegriffen werden.

In einer Instanz der Klasse *Resource* werden alle vorhanden Skills und Funktionen gespeichert. Da mehrere Funktionen auf die selbe IRL-Dateien verweisen können, werden die IRL-Datei Objekte hier gespeichert. Damit werden Objekt-Duplikate für die selben Datei vermieden.

Für Skills im Blackboard steht die Klasse *Skill* zur Verfügung. Dort werden alle Eingangsgrößen unter den globalen Namensschema gespeichert. Des weiteren werden die Objekte für das Mapping hier erzeugt. Für die Code-Generierung wird die Funktion *setParameters()* verwendet. Mithilfe eines SFC-SkillSteps werden hier die Werte aus den Variablen den globalen Namen des Blackboard-Skills zugeordnet.

Die Klasse *Function* repräsentiert ein Funktions-Element aus dem Blackboard. Hier wird das FunctionLogic-Interface ausgelesen, um den Namen der Funktion und den Pfad zur IRL-Datei auszulesen.

Für das Mapping zwischen Blackboard und SFC gibt es die Klasse *SFCMapping*. Dieses liest die Werte des SFC-Mapping (siehe 2.4.1) aus. Die Klasse *FunctionMapping* ist für das

Mapping zwischen Skillnamen und Funktionsparametern (siehe 2.4.2) zuständig. Beide Mappings werden von einem Skill-Objekt erzeugt und gespeichert.

Obwohl die IRL-Dateien logisch nicht zum Blackboard gehören, befinden sie sich ebenfalls in diesem Namespace. Sollte die Funktionalität erweitert werden, so dass eine beliebige Programmiersprache statt nur KUKA KRL implementiert wird, kann dieser Teil auch ausgelagert werden. In aktueller Version gibt es nur zwei Klassen hierfür: *IRLFile* und *FunctionCall*. Pro referenzierte IRL-Datei wird eine Instanz von *IRLFile* erzeugt. Mithilfe der *FunctionCall* Klasse wird bei der Code-Generierung der Textbaustein aus der IRL-Datei mit Werten gefüllt und zurückgegeben.

3.1.2 sfc-Namespace

Der sfc-Namespace beschränkt sich auf das Auslesen und speichern von Objekten aus der SFC-Datei. In der Implementierung existiert hierfür eine Superklasse *FCObject*. In dieser Klasse ist alles zusammengefasst, was Skill-Steps und Transitionen gemeinsam haben. Um die Verarbeitung zu vereinfachen, besitzt diese Datenstruktur, im Gegensatz zu den Objekten in der SFC-Datei, auch Verweise auf die Nachfolger. Nach dem Einlesen wird für jede Verknüpfung beim Vorgänger die ID des jeweiligen Partners eingetragen. Dadurch kann bei der Code-Generierung Schritt für Schritt bearbeitet werden.

Von dieser Klasse abgeleitet, gibt es die Klassen *SkillStep* und *Transition*. In letztere werden alle Arten von Transitionen zusammengefasst. Anhand der Anzahl Ein- und Ausgänge wird jedoch gespeichert, um welche Art der Transition es sich handelt. Wird beim Verbinden von Objekten eine Verletzung der in Abschnitt 2.2.3 genannten Bedingung festgestellt, wird eine Fehlermeldung generiert. Dies deutet sehr stark auf eine Fehlerhafte SFC-Datei hin. Weiter Informationen die in der Transitionen hinterlegt sind, zum Beispiel das *condition* Element, sind für den Post-Prozessor nicht von Bedeutung und werden deshalb ignoriert.

Für die Verwaltung der Skill-Steps gibt es eine Hilfsklasse namens *StepSelector*, die anhand der XML-Struktur die nötigen *SkillStep*-Instanzen erzeugt. Hier sind alle unterstützen Skill-Step Typen hinterlegt. Um neue Varianten hinzuzufügen benötigt man nur einen weiteren Aufruf der Register-Funktion. Sollte beim Auslesen der SFC-Datei benötigte Variablen nicht vorhanden sein, wird eine Fehlermeldung ausgegeben. Überschüssige Variablen werden ignoriert. Ein Skill-Step darf immer nur einen Ein- und einen Ausgang besitzen. Wird diese Bedingung verletzt, kann der Post-Prozessor nicht richtig arbeiten und gibt eine Fehlermeldung aus. Auch hier ist höchst wahrscheinlich die SFC-Datei

fehlerhaft.

```
registerStep("P2P", vector<String>{"Frame", "Velocity", "JointConfig"});
```

Auflistung 3.3: Aufruf zum Hinzufügen eine neuen Skill-Step Typen

Auflistung 3.3 zeigt ein Beispiel um den Typen *P2P* mit den nötigen Variablen *Frame*, *Velocity* und *JointConfig* zu definieren. Alle Werte der Variablen werden dann automatisch ausgelesen und gespeichert. Dabei ist es unwichtig ob die Variablen Listen enthalten oder nicht. Auch der Datentype ist nicht von Bedeutung. Da im SFC keinerlei Informationen darüber angegeben wird, werden alle Werte intern nur als String abgelegt.

3.1.3 process-Namespace

Dieser Namespace beinhaltet alle Klassen, die zur Verarbeitung und schließlich zur Code-Generierung nötig sind. Die Klasse *Flow* vereint dabei die beiden Eingangsdateien und erzeugt einen sequentiellen Ablauf der einzelnen Schritte mit Hilfe der *Sequence* Klassen. Das Befehle für das zu generierende Programm werden in der Klasse *Program* gesammelt und am Ende als einen ausführbaren KUKA Code in die .src- und .dat-Datei gespeichert. Damit die verwendete Sprache nicht völlig fest programmiert ist, ist diese in der Klasse *Language* gekapselt. Die Implementierung reicht allerdings noch bei Weitem nicht um allgemeine Sprachen zu unterstützen.

Die eigentliche Code-Generierung des Post-Prozessors findet in der *process()* Methode in *Flow* statt. In einer Schleife wird hierbei jeder Schritt der sequentiellen Ablaufs mit Werten versehen und übersetzt. Dabei entstehende Probleme werden ausgegeben, was zu einem Abbruch der Generierung führt. Um einen SFC-Step erfolgreich zu verarbeiten müssen mehrere Punkte durchgeführt werden.

1. Erfassen des Ressource- und Skill-Objekts anhand der Namen.
2. Befüllen von der Variablen durch Übergabe des SFC-Steps an den Blackboard-Skill. Dadurch wird implizit das SFC-Mapping verwendet.
3. Übersetzung der im Skill hinterlegten Funktionen in korrekter Reihenfolge.
   (a) Erfassen der zugehörigen IRL-Datei.
   (b) Auslesen der Funktionsdefiniton, falls vorhanden, Registrierung im Programm.
   (c) Erstellen und Befüllen des Funktionsaufrufs mit den korrekten Parametern. Hinzufügen im Programm.
4. Fortfahren mit Punkt 1 oder beenden.

Der PostProzessor unterstützt für Punkt 2 eine Überprüfung des Datentyps, sofern im Blackboard an dem Attribut der niedrigsten Ebene ein Wert für *DataType* angegeben ist. Nur wenn ein Wert vorhanden ist und nicht dem Datentyp des Wertes aus dem SFC-Schritt entspricht, wird mit einer Fehlermeldung abgebrochen. Unterstütze Datentypen sind im Moment: *xs:double*, *xs:decimal* und *xs:boolean*.

3.2 Entwicklungsumgebung

Als Entwicklungsumgebung diente Ecplise. Um unter Windows das C++-Projekt zu importieren wird ein C++-Compiler benötigt. Für Windows existiert MinGW und Cygwin, die beide den GNU Compiler mitbringen. Der Post-Prozessor wurde mit Hilfe von MinGW geschrieben, da die so erzeugte Executable (.exe) mit zwei DLL-Dateien auch auf Rechnern ohne MinGW-Installation lauffähig ist. Unter Linux sind nur entprechende Compiler aus dem repository zu installieren, hier existieren keine Abhängigkeiten bei der Ausführung.

3.3 Einschränkungen

Die erfolgreiche Generierung von Code ist an ein paar Voraussetzungen gekoppelt. Die meisten essentiellen Punkte sind in Abschnitt 2 erwähnt. Um Komplikationen vorzubeugen, sind hier noch wichtige Punkte hinzuzufügen.

3.3.1 Namensgebung

Aus Gründen der Plattformunabhängigkeit ist bei der Wahl von Namen Vorsicht geboten. Gerade bei Sonderzeichen kann es zu systembedingten Auslesefehlern kommen. Der Post-Prozessor unterstützt zwar explizit Umlaute und das scharfe S, andere Sonderzeichen sollten aber vermieden werden. Es wird empfohlen, sich auf ASCII-Schriftzeichen zu beschränken. Ein Besonderheit stellen die Attribute im Blackboard dar. Da hier der Punkt als Trennzeichen für die globalen Namen verwendet wird, darf dieser nicht für die Namen der Parameter verwendet werden.

3.3.2 Dateipfade

In früheren Implementierungen wurde oftmals ein absoluter Pfad zu Dateien verwendet. Dadurch ist zwar immer eine eindeutige Stelle beschrieben, bringt aber auch diverse Nachteile. So kann ein Projekt-Ordner mit Blackboard und IRL-Dateien nicht verschoben oder gar auf andere System übertragen werden. Aus diesem Grund arbeitet der Post-Prozessor immer mit relativen Pfaden in Bezug auf die Blackboard-Datei. Dies gilt auch bei der Erstellung des Mappings im Blackboard zu beachten. Da in der Unix-Welt Dateipfade, anders als in Windows, statt '\' mit '/' geschrieben werden, verarbeitet der Post-Prozessor beide Zeichen. Leerzeichen im Pfad oder den Dateinamen sollten nach Möglichkeiten vermieden werden, um eine korrekte Ausführung zu garantieren.

3.3.3 Groß- und Kleinschreibung

In der Implementierung des Post-Prozessors wurde darauf geachtet, das Matching-Verfahren die auf Namen, sprich strings, beruhen nicht auf Groß- und Kleinschreibung achten. Um dennoch Fehlerquellen zu vermeiden, sollte ein einheitliches Schema in Blackboard und IRL-Dateien gewählt werden. Sollten dennoch Fehlermeldungen vorkommen, gilt es die Schreibweise nochmal zu überprüfen.

3.4 Optionen

Um bei Fehlern nähere Informationen zu erhalten, gibt es zusätzliche Optionen. Diese Liste der Optionen lässt sich auch beim Ausführen mit `--help` anzeigen.

`--help`
Gibt die Liste an Optionen aus.

`--info`
Gibt die Versionsnummer und das Build-Date aus. Zusätzlich wird die Liste der unterstützen Skill-Steps ausgegeben.

`--warn`
Gibt alle Warnungen aus. Als Warnung wird alles klassifiziert, was den weiteren Ablauf des Programms nicht verhindert.

`--validate`
Startet den PostProzessor im Validierungs-Modus. Das bedeutet es werden alle Eingaben gelesen und überprüft. Beinhaltet die Option `--warn`. Führt keine Code-Generierung aus.

`--sfc`
Überprüfungs-Modus für SFC-Dateien. Mit dieser Option wird statt einer Blackboard-Datei eine SFC-Datei erwartet. Beinhaltet die Option `--validate` und `--warn`.

`-pB`
Gibt alle ausgelesenen Informationen aus dem Blackboard aus.

`-pS`
Gibt alle ausgelesenen Informationen zu den SFC-Objekten aus.

`-pF`
Gibt ein pseudographischen Ablauf der SFC-Schritte aus.

`-pO`
Gibt die chronologische Abfolge der Schritte aus, wie sie der PostProzessor bearbeitet.

Abschnitt 4

Zusammenfassung

Mit dem Post-Prozessor wurde die Machbarkeit von automatisierten Herstellungsverfahren weiter demonstriert. Solange das Robotersystem im Blackboard ausreichend beschrieben ist, ist der Post-Prozessor in der Lage eine beliebige Abfolge von Arbeitsschritten in Maschinen-Code zu übersetzen. Obwohl das Projekt grundsätzlich auf allgemeine Systeme ausgelegt ist, wurde ausschließlich mit KUKA KRL gearbeitet und getestet. Die Abstraktion ist in vielen Bereichen angedacht oder teilweise Implementiert, wodurch viele Punkte fest einprogrammiert sind, die flexibel sein müssten. Hier sind weitere Ansätze nötig um alle Feinheiten, Strukturen und den Syntax von verschiedenen Sprachen in einem Post-Prozessor zu vereinen.

Mit seinen Validierungsfunktion bietet der Post-Prozessor auch die Möglichkeit Fehler in den Eingabedateien aufzuspüren und bei der Behebung dieser zu unterstützen.

Während der Durchführung sind einige Änderungen am Blackbaord-Konzept vorgenommen worden. Deshalb müssen viele Dinge noch per Hand zu erledigt werden. Das trifft hauptsächlich das Mapping-Verfahren. Hier können automatisierte Prozesse dem Anwender nochmal ein großes Stück an Arbeit erleichtern. Ebenso benötigt das Hinzufügen von Skill-Steps eine, wenn auch kleine, Änderung im Quellcode. Durch Auslagerung dieser Information in eine externe XML-Datei ließe sich die Flexibilität weiter steigern.

An dieser Stelle möchte ich mich auch nochmal bei meinem Betreuer Dipl.-Ing. Julian Backhaus für die gute Zusammenarbeit bedanken. Des Weiteren gilt auch mein Dank Max Hujber, der die nötigen Änderungen an den Planungsmodulen vorgenommen hat, um eine automatisierte Code-Generierung überhaupt möglich zu machen.