Wir planen die Produktion - mit Expertenwissen!

Storyboard

In der Technikum Digital Factory stellen wir zurzeit Roboter für Forschung- und Ausbildungszwecke her. Da dabei mehrere Robotertypen in der gleichen Produktionsumgebung gefertigt werden, ändert sich der Ablauf in der Fabrik regelmäßig. Bei jeder Änderung des Produktionsablaufs muss aber erneut geplant werden, wie die einzelnen Werkstücke die Fabrik durchlaufen müssen, damit die Roboter gefertigt werden können. Dabei muss die Dauer jedes Bearbeitungsschritts, die Verfügbarkeit mehrerer gleicher Maschinen und die Reihenfolge der Bearbeitungsschritte berücksichtigt werden. Da diese Planungsaufgabe für jede Änderung der Produktion erneut durchgeführt werden muss, wollen wir sie automatisch lösen lassen.

Abbildung 1 zeigt eine der Produktionsstraßen, in der die ankommenden Teile an vier Stationen verarbeitet werden. Die zu verplanenden Stationen sind eine Fräse, eine Borhstation, eine Station zum Zusammenbau und eine Polierstation. Dabei dauern die Bearbeitungsschritte an den Stationen unterschiedlich lang und die Fräse ist mehrmals vorhanden. Da das vorliegende Problem eine Planungsaufgabe ist, wollen wir Logikprogrammierung anwenden. Im Zuge von Logikprogrammierung formulieren wir ein Problem in Form von Einschränkungen, damit es automatisch gelöst werden kann. Diese Art der Problemformulierung ist speziell geeignet, um solche Planungsaufgaben zu lösen (siehe AIAV Video Heutige Anwendungen - Fokus Logikprogrammierung).

Dabei verbinden wir Logikprogrammierung mit einer Ontologie, um aus Wissen über die einzelnen Produktionsschritte automatisch den Produktionsablauf zu planen. Diese Vorgehen erlaubt es uns, bei einem sich ändernden Produktionsumfeld die Ontologie anzupassen und den Ablauf automatisch neu planen zu lassen.

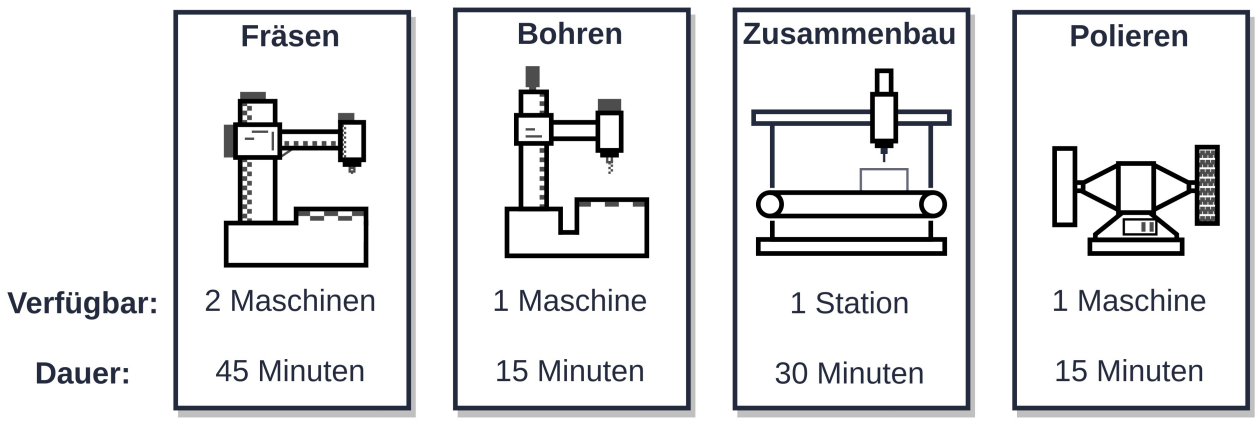

Abbildung 1: Der zu planende Produktionsablauf umfasst eine Fräse, eine Bohrstation, eine Station zum Zusammenbau und eine Poliermaschine. Dabei dauert jeder Bearbeitungsschritt unterschiedlich lang und manche Maschinen, wie z.B. die Fräse, sind mehrmals vorhanden.

Ontologien

Mittels Ontologien können wir Expertenwissen in eine maschinenlesbare Form bringen (siehe AIAV Video Ontologien), indem wir Vorgänge in der Umwelt mittels Konzepten und Instanzen beschreiben. Konzepte stellen dabei allgemeine Zusammenhänge dar, während Instanzen konkrete Beispiele für

diese Zusammenhänge sind. Ein Beispiel für so ein Konzept ist "*Künstler malen Kunstwerke.*", welches im allgemeinen Künstler mit Kunstwerken in Beziehung setzt (siehe Abbildung 2). Basierend auf so einem Konzept werden dann Instanzen, also konkrete Beispiele für die Konzepte, festgehalten. "*Da Vinci ist ein Künstler, Mona Lisa ein Kunstwerk.*", sind jeweils Instanzen der Konzepte von *Künstler* und *Kunstwerk*. Vom Wissen aus unserem Konzept können wir nun ableiten, dass Da Vinci die Mona Lisa gemalt hat.

Dieses Vorgehen erlaubt es uns mittels Konzepten und Instanzen eine zusammenhängende Wissensbasis zu erstellen, die einen bestimmten Wirkungsraum beschreibt und währenddessen auch einfach maschinell verarbeitet werden kann. Die Verarbeitung erfolgt dabei durch den sogenannten Reasoner, welcher Anfragen verarbeiten und Zusammenhänge zwischen Instanzen von der Wissensbasis ableiten kann.

Bekannte Konzepte und Instanzen werden in Form eines sogenannten Knowledge Graphs dargestellt, um die Wissensbasis einfach lesbar zu machen. Abbildung 2 stellt unser Beispiel als so einen Knowledge Graph dar.

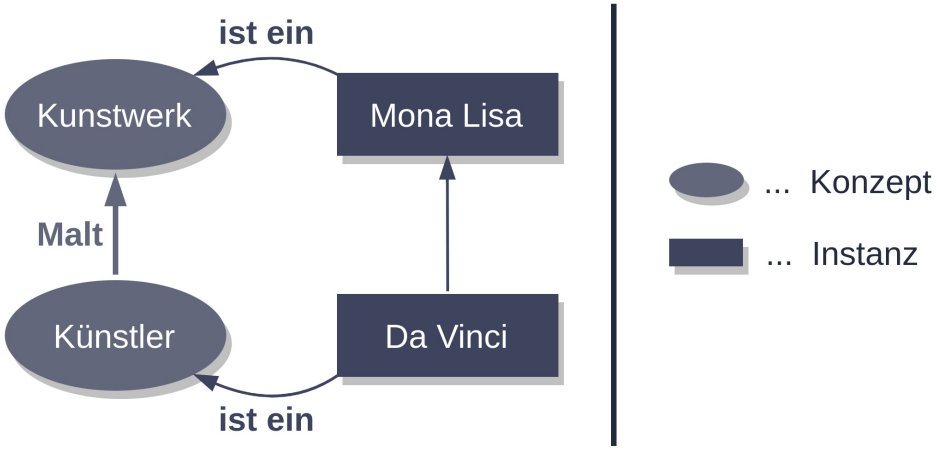

Abbildung 2: Wir stellen unsere Wissensbasis in form eines Knowledge Graphs dar, um sie einfach verständlich zu machen. Das Konzept ***Künstler malen Kunstwerke*** *ist der Wissensbasis dabei bekannt. Da Da Vinci ein Künstler und Mona Lisa ein Kunstwerk ist, können wir aus der Wissensbasis ableiten, dass Da Vinci die Mona Lisa gemalt hat.*

Logikprogrammierung

Python oder C sind gängige Beispiele für Programmiersprachen in denen auf künstlicher Intelligenz basierende Lösungen implementiert werden. Diese beiden Sprachen sind Teil der sogenannten imperativen Programmiersprachen, da man bei der Programmierung klare Anweisungen gibt, welche Aktionen das Programm durchführen soll. Dabei entscheidet der Programmierer wie das Programm vorgeht um ein Problem zu lösen.

Analog zu den imperativen Programmiersprachen steht die Logikprogrammierung (siehe AIAV Video Wissensrepräsentation und Logikprogrammierung). Dabei formulieren wir das Problem in Form von Vorgaben, bestehend aus logischen Operatoren. Beispiele für solche Operatoren sind UND und ODER, welche zwei oder mehrere Eingänge miteinander verknüpfen. Sie können sich diese Vorgaben wie eine Art Spezifikation vorstellen; diese gibt an was erfüllt sein muss, sodass eine Lösung für das Problem vorhanden ist. Dabei schränkt jede Vorgabe die Menge an möglichen Lösungen weiter ein, bis eine Lösung vorhanden ist, die unseren gewünschten Zweck erfüllt. Aufgrund dieses Vorgehens nennt man

die Vorgaben auch Einschränkungen oder Constraints.

Nach Festlegen der Einschränkungen werden diese in ein Lösungsprogramm, einem sogenannten Boolean Satisfiability (SAT) Solver, gegeben. Dieser vereinfacht zunächst die Gesamtheit der vorgegebenen Einschränkungen. Diese vereinfachte Form kann dann ausgegeben oder zur Suche nach spezifischen Lösungen verwendet werden. Ein gängiger SAT Solver ist Teil der Programmiersprache Prolog. Prolog erlaubt es, Einschränkungen in einer eigenen Sprache zu formulieren, zu vereinfachen und dynamisch Anfragen basierend auf den Einschränkungen zu verarbeiten. Neben dedizierten Logik Programmiersprachen, wie Prolog, gibt es aber auch Solver, welche die Formulierung der Einschränkungen in bestehenden Programmiersprachen erlauben. Ein Beispiel für so eine Bibliothek ist Z3, ein offener SAT Solver von Microsoft. Dieser Unterstützt unter anderem C, C++, Java und Python.

Praktische Implementierung

Für die praktische Implementierung wollen wir basierend auf einer Ontologie planen, wann welche Werkstücke in jeder Maschine bearbeitet werden. Die Ontologie soll die Grundlage für die Planung sein, da diese einfach verändert werden kann, um den Produktionsablauf anzupassen. Abbildung 3 zeigt dabei die für den Use Case definiterte Ontologie. Diese beinhaltet zwei Konzepte die beschreiben, dass Maschinen Werkstücke erzeugen und Werkstücke von Maschinen verarbeitet werden. Dabei definieren wir die einzelnen Maschinen als Instanzen des Konzepts *Maschine* und den Fertigstellungsgrad des Werkstücks als Instanzen des Konzepts *Werkstück*. Der Ablauf, in dem die Werkstücke bearbeitet werden, ergibt sich dann aus den Beziehungen der einzelnen Instanzen von Werkstücken und Maschinen.

Anhand der Ontologie in Abbildung 3 bearbeiten wir die Anfrage, in welcher Beziehung das unbearbeitete und fertige Werkstück zueinander stehen. Die Antwort auf diese Anfrage beinhaltet sowohl Werkstücke als auch Maschinen und repräsentiert den Produktionsablauf.

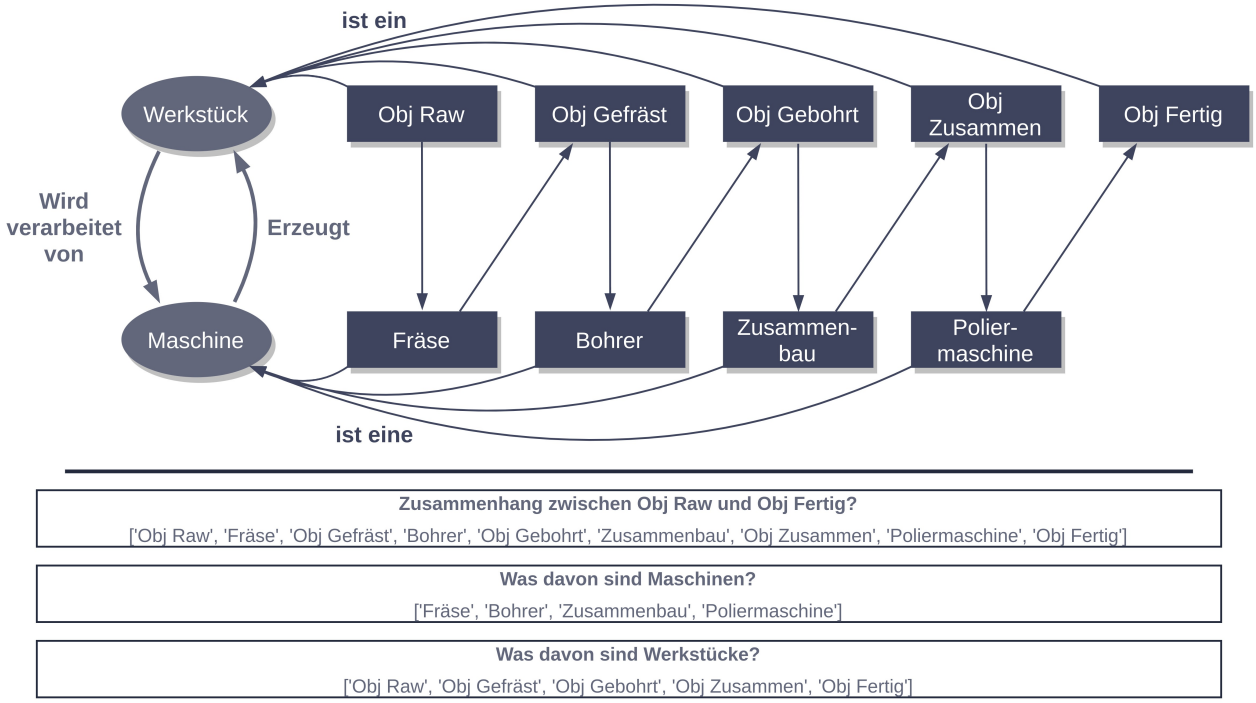

Abbildung 3: Wir erstellen eine Ontologie, anhand derer die Produktion geplant wird. Dabei wissen wir aus zwei Konzepten, dass Maschinen Werkstücke erzeugen und Werkstücke von Maschinen

verarbeitet werden. Durch Verknüpfung der verschiedenen Instanzen können wir so den Produktionsablauf beschreiben. Die benötigten Verarbeitungsschritte eines Werkstücks können mittels Anfragen ermittelt werden.

Nach Festlegen der Ontologie wenden wir nun Logikprogrammierung an, um den zeitlichen Verlauf der Produktion zu planen. Dafür benötigen wir die aus der Ontologie ermittelte Produktionsabfolge sowie die Dauer der Bearbeitungsschritte jeder Maschine und die Anzhal der verfügbaren Maschinen (siehe Abbildung 1). Wir teilen zunächst die verfügbare Zeit in Zeitslots mit einer fixen Länge (z.B. 15 Minuten) ein und definieren die benötigte Zeit für jeden Produktionsschritt in Form dieser Slots. Wir betrachten die verfügbare Zeit in Form dieser Slots, um die Anzahl der resultierenden Einschränkungen zu verringern und damit die benötigte Rechenleistung beim Ermitteln des Plans so gering wie möglich zu halten. Anschließend definieren wir die Anzahl der Teile, die wir in den verfügbaren Zeitslots verplanen wollen.

Die Einschränkung, dass allen Zeitslots keines oder ein einziges Teil zugeordnet werden muss, stellt die Grundlage der Einteilung dar und wird gespeichert. Sie legt die Zeitslots für jede Maschine fest. Anschließend werden die Einschränkungen pro zu verarbeitendem Teil aufgestellt. Dabei müssen wir beachten, dass jedes Teil nicht gleichzeitig an zwei Orten, also in mehreren Maschinen, sein kann und dass die Anzahl und Reihenfolge der Verarbeitungsschritte im Vorhinein nicht bekannt sind. Deswegen gehen wir wie folgt vor:

- Iteration über alle möglichen Zeitpunkte an denen die Produktion eines Teils starten kann:
  - Iteration über alle Verarbeitungsschritte:
  - Iteration über alle Maschinen im aktuellen Verarbeitungsschritt:
    - Aufstellen der Einschränkung, dass das aktuelle Teil zur aktuellen Zeit in der aktuellen Maschine ist
    - Aufstellen der Einschränkungen, dass das aktuelle Teil zur aktueleln Zeit in keiner anderen Maschine ist
    - Aufstellen der Einschränkungen, dass das aktuelle Teil nie in einer Maschine gleichen Typs ist (da jeder Verarbeitungsschritt nur einmal ausgeführt wird)

Einschränkungen werden anhand dieses Ablaufs so lange erstellt und gespeichert, bis alle zu verplanenden Teile zu allen möglichen Startzeitpunkten berücksichtigt wurden. Anschließend erstellen wir ein Z3 Modell mit allen gespeicherten Einschränkungen, welches durch den Solver gelöst wird. Die Lösung kommt dabei in Form einer Tabelle, die beschreibt welches Werkstück in welchen Zeitslots in welcher Maschine ist (siehe Abbildung 4). Dabei stellt jede Zahl (1 - 6) den Index eines Werkstücks dar, während ein Leeres Feld heiß, dass die jeweilige Maschine zu diesem Zeitpunkt leer ist.

In Abbildung 4 sehen wir, dass beim oben vorgestellten Problem ein Engpass in der Produktion beim Zusammenbau der Werkstücke entsteht. Hier kann der Durchsatz in der Produktion durch eine zweite Station zum Zusammenbau erhöht werden.

	Mill	Mill	Drill	Assembly	Polish
T0:		6			
T1:		6			
T2:	4	6			
T3:	4		6		
T4:	4	5		6	
T5:		5	4	6	
T6:	2	5		4	6
T7:	2		5	4	
T8:	2	1		5	4
T9:		1	2	5	
T10:	3	1		2	5
T11:	3		1	2	
T12:	3			1	2
T13:			3	1	
T14:				3	1
T15:				3	
T16:					3

Abbildung 4: Die anhand der Ontologie erstellten Einschränkungen werden in einen SAT Solver gegeben, welcher eine Lösung ermittelt, die alle Einschränkungen erfüllt. Für unser Problem kommt diese Lösung in Form einer Tabelle, welche die Werkstücke bestimmten Zeitslots in den Maschinen zuordnet. Hier wurden sechs Werkstücke über 16 Zeitslots für den in Abbildung 1 beschriebenen Produktionsablauf verplant.

Fazit

Ontologien erlauben es uns, Wissen über ein bestimmtes Umfeld effizient in eine für Maschinen verständliche Form zu bringen und dynamisch Anfragen zu verarbeiten. Umfangreiche Software Pakete zur Formulierung und Abfrage von Ontologien sind gratis verfügbar und ermöglichen die Verarbeitung von Ontologien mit nur wenig Rechenleistung.

Mittels Logikprogrammierung können wir Planungsprobleme in Form von Einschränkungen formulieren und automatisch lösen lassen. In diesem Use Case wurden Ontologien mit Logikprogrammierung gekoppelt, um ein Planungsproblem in der Produktion autonom, auf Basis von Expertenwissen über den Produktionsablauf, zu planen. Die Kombination von Logikprogrammierung und Ontologien erlaubt es uns, den Produktionsablauf direkt aus den Informationen über die Produktionsschritte zu planen. Ändern sich also Aspekte der Produktion, können diese in die Ontologie eingespielt werden. Anschließend wird der Produktionsplan autonom überarbeitet.

Codedokumentation

Die Implementierung basiert auf Python 3 und verwendet Z3 zur Logikprogrammierung, Numpy zur Datenverwaltung und Matplotlib zur Datenvisualisierung. Um die Lösung anschaulicher zu gestalten, wurden die Datenstrukturen zum Aufstellen der Ontologie und zum Verarbeiten von Anfragen manuell in Python Implementiert. Für komplexe Ontologien können hier alternativ der online Protege Ontologie Editor zur Erstellung der Ontologie und das Python Modul Owlready zum Laden und Verwenden der Ontologie verwendet werden.

Die Applikation besteht aus dem Python Skript *produktionskette.py*.

Formulierung der Ontologie

Zunächst importieren wir Z3, Numpy und Matplotlib und implementieren einige Hilfsfunktionen. Dabei halten die globalen Variablen *things* und *relations* alle bekannten Konzepte und Relationen zwischen Konzepten.

In [ ]:

```
from z3 import*
import numpy as np

import matplotlib.pyplot as plt

#################################################################
#####           Ontologie zur Ablaufplanung              #####

# Deklaration der Datenstruktur für die Ontologie und Implementierung von Hilfskla
# Die Ontologie wird als Liste von Dingen (Things) bestehend aus Konzepten und ihr
# Die Instanzen und Konzepte werden dann von separat gespeicherten Relationen verk

things = []
relations = []

class thing:
    """ Klasse, welche ein Konzept beschreibt """
    def __init__(self, concept, instances=[]):
        self.concept = concept
        self.instances = instances

def listConcepts(things):
    """ Gibt alle bekannten Konzepte aus """
    return [t.concept for t in things]

def listInstances(concept, things):
    """ Gibt alle Instanzen eines Konzeptes aus """
    for t in things:
        if t.concept == concept:
            return t.instances
    return []

class relation:
    """ Klasse, welche Relationen zwischen Konzepten beschreibt """
    def __init__(self, who, action, target):
        self.who = who
        self.action = action
        self.target = target

def getRelevantRelation(instance, relations):
    """ Gibt alle Relationen einer Instanz zurück """
    ret1 = [] # Zusammenhänge bei denen instance der Akteur ist
```

```
    ret2 = [] # Zusammenhänge bei denen instance das Ziel ist
    for r in relations:
        if r.who == instance: ret1.append(r)
        elif r.target == instance: ret2.append(r)
    return ret1, ret2

def plotThings(things):
    """ Gibt alle Konzepte und alle zugehörigen Instanzen aus """
    for t in things:
        print("{}".format(t.concept))
        print(" Instances:")
        for i in t.instances: print(" - {}".format(i))
        print(" ")

def queryReason(goal, start, things, relations):
    """ Bearbeitet Anfragen und gibt den Zusammenhang zwischen zwei Instanzen zurü
    node = goal
    path = []
    path.append(node)
    while node != start:
        _, rel = getRelevantRelation(node, relations)
        for r in rel:
            if r.target == node:
                node = r.who
                path.append(node)
                break
    return path
```

Anschließend formulieren wir die Ontologie in Abbildung 3. Dabei werden alle Konzepte und zugehörigen Instanzen instanziert und die Relationen zwischen den Instanzen gespeichert. Wir verwenden dann die *queryReason* Funktion, um den Zusammenhang zwischen *Obj_Raw* und *Obj_Finished* zu finden. Dieser Zusammenhang stellt unseren Produktionsablauf dar und beinhaltet Instanzen von Maschinen und Werkstücken.

In [ ]:

```
# Konzept des Werkstücks mit verschiedenen Verarbeitungsstufen als Instanzen
things.append(thing(
    "Part", instances=["Obj_Raw", "Obj_Milled", "Obj_Drilled", "Obj_Assembled", "Ob
))

# Konzept der Maschine mit den einzelnen Stationen als Instanzen
things.append(thing(
    "Machine", instances=["Mill", "Drill", "Assembly", "Polish"]
))

# Festlegen der Relationen laut Ontologie
relations.append(relation("Obj_Raw", "used by", "Mill"))
relations.append(relation("Mill", "creates", "Obj_Milled"))

relations.append(relation("Obj_Milled", "used by", "Drill"))
relations.append(relation("Drill", "creates", "Obj_Drilled"))

relations.append(relation("Obj_Drilled", "used by", "Assembly"))
relations.append(relation("Assembly", "creates", "Obj_Assembled"))

relations.append(relation("Obj_Assembled", "used by", "Polish"))
relations.append(relation("Polish", "creates", "Obj_Finished"))

# Lösung = Zusammenhang (Relation) zwischen Obj_Raw und Obj_Finished
reason = queryReason("Obj_Finished", "Obj_Raw", things, relations)
reason.reverse()

# Ermittlung des Ablaufs der Maschinen
tmp = listInstances("Machine", things)
```

```
machines = []

for r in reason:
    if r in tmp: machines.append(r)

# Ermittlung der Abfolge der Bearbeitungsschritte des Werkstücks
tmp = listInstances("Part", things)
steps = []

for r in reason:
    if r in tmp: steps.append(r)
```

Erstellen des Produktionsplans mittels Logikprogrammierung

Um den Produktionsplan aufstellen zu können, teilen wir zunächst die verfügbare Zeit in Zeitslots von fixer Länge auf. Wir legen fest, wie viele Zeitslots zur Verfügung stehen und wie viele Teile zu verplanen sind. Anschließend speichern wir wie lange jeder Produktionsschritt dauert und wie viele Maschinen des jeweiligen Maschinentyps verfügbar sind.

Um die Erstellung der Einschränkungen zu vereinfachen, speichern wir die Informationen über die Zeitslots und Maschinen als globale Variablen.

In [ ]:

```
#################################################################
#####                LOGIKPROGRAMMIERUNG                    #####

# Wir teilen die zu verplanende Zeit in Zeitslots bei jeder Maschine auf
# Diesen Zeitslots wird später je ein Werkstück zugeordnet

# Anzahl der verfügbaren Zeitslots
numTimeSlots = 17

# Anzahl der zu verplanenden Teile
numParts = 6

# Info über verfügbare Maschinen
# Format der Liste: [ Maschinentyp, Anzahl der verfügbaren Maschinen des Typs, ben
machineTypes = np.array([
    ['Mill',     2, 3],
    ['Drill',    1, 1],
    ['Assembly',1, 2],
    ['Polish',   1, 1]
])

# Erstellen der Liste aller verfügbaren Maschinen
machineList = []
for m in machineTypes:
    for i in range(int(m[1])):
        machineList.append(m[0])

# Erstellen der Liste der frühesten Zeiten an denen jeder Produktionsschritt begin
machineDurations = []
tmpSum = 0
for m in machineTypes:
    machineDurations.append(tmpSum)
    tmpSum += int(m[2])

machineDurations.append(tmpSum)

# Speichern der Anzahl der Maschinen
numMachines = len(machineList)
```

Nach dem Festlegen der Zeitslots als Z3 Variablen, itierieren wir über jedes Teil und jeden Zeitslot an dem die Produktion des Teils starten kann. Die Verschachtelung der Iterationen erfolgt dabei so:

- Iteration über alle möglichen Zeitpunkte an denen die Produktion eines Teils starten kann:
  - Iteration über alle Verarbeitungsschritte:
  - Iteration über alle Maschinen im aktuellen Verarbeitungsschritt:
    - Aufstellen der Einschränkung, dass das aktuelle Teil zur aktuellen Zeit in der aktuellen Maschine ist
    - Aufstellen der Einschränkungen, dass das aktuelle Teil zur aktueleln Zeit in keiner anderen Maschine ist
    - Aufstellen der Einschränkungen, dass das aktuelle Teil nie in einer Maschine gleichen Typs ist (da jeder Verarbeitungsschritt nur einmal ausgeführt wird)

In [ ]:

```
# Erstellen der Zeitslots, damit diese von Z3 ein Teil zugeordnet bekommen
# Format der Slots: num_time_machine
machineSlots = [ [ Int("num_%s_%s" % (t+1, m+1)) for m in range(numMachines) ] for
machineCellConstraints  = [ And(0 <= machineSlots[t][m], machineSlots[t][m] <= numM

# Hilfsfunktion für das Aufstellen der Einschränkungen
def flatten(t):
    return [ item for sublist in t for item in sublist ]

## Erstellen der Einschränkungen
allConstraints = []
partConstraints = []

# Iteration über alle zu verplanenden Teile
for part in range(1,numParts+1):
    # Iteration über alle möglichen Startzeitpunkte
    tstartConstraints = []
    for tstart in range(numTimeSlots):
        # Erstellung der Einschränkungen pro Startzeitpunkt pro Teil
        #
        # Überprüfung, ob ein zu tstart angefangenes Teil fertig produziert werden
        # Wenn nicht, wird dieser und alle folgenden Startzeitpunkte übersprungen
        if numTimeSlots-tstart < machineDurations[len(machineTypes)]:
            break
        #
        # tmp speichert alle Einschränkungen für das aktuelle Teil zum aktuellen S
        tmp = []
        # Festlegen des Objektzustandes am Beginn der Produtkion
        obj = steps[0]
        #
        # Loop bis das Teil alle Produktionsstufen abgeschlossen hat
        while obj != steps[len(steps)-1]:
            #
            # Ermittlung, welche Maschine als nächstes benötigt wird
            reqMachine = reason[reason.index(obj)+1]
            # Ermittlung der Maschineninfo aus machineTypes
            idx = np.where(machineTypes[:,0] == reqMachine)[0][0]
            _, availMachines, reqTime = machineTypes[idx]
            reqTime = int(reqTime)
            availMachines = int(availMachines)
            # Ermittlung, in welchem Zeitslot der Verarbeitungsschritt relativ zu
            tmachine = machineDurations[idx]
            # Ermittlung des Index der aktuellen Maschine auf dem Zeitplan
            machineIndex = np.where(np.array(machineList) == reqMachine)[0][0]
            #
            # Prüfung, ob der Produtionsschritt noch in der verfügbaren Zeit durch
            # Falls nicht mehr genug Zeitslots frei sind, wird abgebrochen
            if reqTime > numTimeSlots-(tstart+tmachine):
```

```
                    break
                #
                # Prüfung, ob auch Maschinen dieses Typs verfügbar sind, wenn nein, da
                if availMachines < 1:
                    break
                #
                # Unterscheidung, ob die Maschine mehrmals verfügbar ist
                if availMachines == 1:
                    #
                    # Hinzufügen der aktuellen Maschine
                    tmp.append(And(flatten([
                        # Setzen der Slots welches ein Teil belegt
                        # slot = part bei -> t: tstart+tmachine - tstart+tmachine+reqT
                        [ machineSlots[t][machineIndex] == part for t in range(tstart+
                        ## Setzen aller anderer Slots als nicht vom jeweiligen Teil be
                        # Alle Slots in der gleichen Maschine vor der Bearbeitung
                        [ machineSlots[t][machineIndex] != part for t in range(tstart+
                        # Alle Slots in der gleichen Maschine nach der Bearbeitung
                        [ machineSlots[t][machineIndex] != part for t in range(tstart+
                        # Alle Maschinen vor der verwendeten Maschine während das Teil
                        [ machineSlots[t][m] != part for t in range(tstart+tmachine, ts
                        # Alle Maschinen nach der verwendeten Maschine während das Tei
                        [ machineSlots[t][m] != part for t in range(tstart+tmachine, ts
                    ])))
                else:
                    #
                    # Hinzufügen der aller möglichen Ableger derselben Maschine
                    tmp2 = []
                    # Iteration über alle verfügbaren Maschinen des gleichen Typs
                    for ver in range(availMachines):
                            tmp2.append(And(flatten([
                                # Setzen der Slots welches ein Teil belegt
                                # slot = part bei -> t: tstart+tmachine - tstart+tmach
                                [ machineSlots[t][machineIndex+ver] == part for t in ra
                                ## Setzen aller anderer Slots als nicht vom jeweiligen
                                # Alle Slots in der gleichen Maschine vor der Bearbeit
                                [ machineSlots[t][machineIndex+ver] != part for t in ra
                                # Alle Slots in der gleichen Maschine nach der Bearbei
                                [ machineSlots[t][machineIndex+ver] != part for t in ra
                                # Alle Maschinen vor der verwendeten Maschine während
                                [ machineSlots[t][m] != part for t in range(tstart+tma
                                # Alle Maschinen nach der verwendeten Maschine während
                                [ machineSlots[t][m] != part for t in range(tstart+tma
                            ])))
                    tmp.append(Or(tmp2))
                #
                # Bestimmen des aktuellen Zustands des Teils
                obj =  reason[reason.index(reqMachine)+1]
            #
            # Hinzufügen aller Constraints für den aktuellen Startzeitbunkt des aktuel
            tstartConstraints.append(And(tmp))
        #
        # Hinzufügen aller Constraints für des Aktuelle Bauteil
        partConstraints.append(Or(tstartConstraints))

allConstraints.append(And(partConstraints))
allConstraints.append(And(machineCellConstraints))
```

Die gespeicherten Einschränkungen werden dann einem Z3 Modell hinzugefügt. Z3 ermittelt dann eine spezifische Lösung, die alle Einschränkungen erfüllt.

In [ ]:

```
# Lösung mittels SAT Solver
s = Solver()
s.add(simplify(And(allConstraints)))
```

```
# Lösen des Systems
if s.check() == sat:
    mod = s.model()
    # Auslesen der Resultate des Modells
    results = [ [ mod.evaluate(machineSlots[t][m]) for m in range(numMachines) ] fo
    print('Lösung berechnet, hier ist das Ergebnis:')
    print('-----------------------------------')
    print(machineList)
    for idx, row in enumerate(results):
        print('T{}:  {}'.format(idx, row))
    print('-----------------------------------')
else:
    print("Nicht alle Teile konnten verplant werden.")
    print("Bitte entfernen Sie einige der angegebenen Teile!")
```

Die vom SAT Solver ermittelte Läsung wird anschließend mittels Matplotlib dargestellt.

In [ ]:

```
######################################################################
##### Visualisierung des Ergbnisses als Matplotlib Tabelle #####

# Konvertierung der Dateneinträge zu Text
cellText = []

for row in results:
    formatRow = []
    for entry in row:
        if entry == 0: formatRow.append(' ')
        else: formatRow.append('{}'.format(entry))
    cellText.append(formatRow)

# Colormaps für die Tabellenbeschriftungen
rcolours = plt.cm.BuPu(np.full(numTimeSlots, 0.1))
ccolours = plt.cm.BuPu(np.full(len(machineList), 0.1))

# Plot als Tabelle
the_table = plt.table(cellText=cellText,
                      rowLabels=[ 'T{}:'.format(i) for i in range(numTimeSlots) ],
                      colLabels=machineList,
                      colColours=rcolours,
                      rowColours=rcolours,
                      rowLoc='right',
                      loc='center')

## Säuberung der Matplot Table
# Ausblenden der Achsenbeschriftungen
ax = plt.gca()
ax.get_xaxis().set_visible(False)
ax.get_yaxis().set_visible(False)

# Ausblenden der Box
plt.box(on=None)

# Vergrößern der Zellen
the_table.scale(1, 1.5)

# Speichern des Plots
plt.savefig('timetable.png',
            bbox_inches='tight',
            dpi=300
            )

plt.show()
```