technische universität
dortmund

Bachelorarbeit

Verwendung von Antwortmengenprogrammierung in einem Multiagentensystem autonomer Fahrzeuge

Oliver Tüselmann
Oktober

Gutachter:
Prof. Dr. Gabriele
Kern-Isberner
M.Sc. MS SCE (USA) Steffen
Schieweck

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhlbezeichnung (LS-1)
http://ls1-www.cs.tu-dortmund.de

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation und Aufgabenstellung

In der Logistik etablieren sich aufgrund ihrer Flexibilität und Schnelligkeit zellulare Fördertechnik (ZFT) Systeme. Um diese Systeme weiter zu optimieren, sollen die Fördereinheiten in dem System intelligenter werden. In der Informatik existieren im Bereich der künstlichen Intelligenz (KI) verschiedene Ansätze, die den Einheiten mehr Intelligenz verschaffen können. Ein Ansatz ist zum Beispiel die Antwortmengenprogrammierung (AWM).

Schieweck et al. [24] haben bereits eine Planungsaufgabe mit Hilfe von AWM für ein ZFT System realisiert. Es wurden bei der Anwendungsaufgabe im Vergleich zur vorherigen Vorgehensweise deutlich bessere Ergebnisse erzielt. Deshalb werden weitere Planungsaufgaben für AWM in diesem Kontext gesucht. Des Weiteren besteht die Frage, wie AWM in die Architektur der ZFT Systeme integriert werden kann.

Um weitere Planungsaufgaben für AWM im ZFT System zu finden, sollen in dieser Arbeit geeignete Aufgaben für AWM im ZFT System identifiziert, implementiert und gegenüber dem bereits bestehenden Ansatz evaluiert werden. Die Integration von AWM in ein ZFT System soll mit Hilfe einer Systemarchitektur realisiert werden. Aus diesem Grund wird in dieser Arbeit eine Systemarchitektur zur Nutzung von Wissensrepräsentation innerhalb eines Multiagentensystems entwickelt und umgesetzt, welche die Verwendung von unterschiedlichen Implementationen aus dem Bereich der Wissensrepräsentation ermöglicht. Mit dieser Architektur soll es dann möglich sein, Aufgaben mit verschiedenen Ansätzen der KI, wie zum Beispiel AWM oder Prolog, im ZFT Kontext zu implementieren und zu evaluieren. Abschließend soll der Mehrwert der Antwortmengenprogrammierung gegenüber dem bereits bestehenden Ansatz ermittelt werden.

1.2 Aufbau der Arbeit

Am Anfang werden die für das Thema notwendigen Grundlagen im Kapitel 2 aufgearbeitet. Dabei wird das Konstrukt Multiagentensystem vorgestellt, um eine einheitliche Definition und Arbeitsweise des Konstrukts vorzugeben. Neben dem Multiagentensystem ist die Antwortmengenprogrammierung der wichtigste Bestandteil dieser Arbeit, weshalb auch hier

detailliert auf die Grundlagen und die Arbeitsweise eingegangen wird. Da sich mehrere Aufgaben in dieser Arbeit auf die zellularen Fördertechnik Systeme beziehen, werden die Grundlagen dieses Systems im Kapitel 2 erklärt. Nachdem die Grundlagen erarbeitet wurden, wird im Kapitel 3 eine Systemarchitektur zur Nutzung von Wissensrepräsentation innerhalb eines Multiagentensystems vorgestellt und die Vorgehensweise der Implementierung dieser Architektur in ein vorhandenes Framework beschrieben. Aufbauend darauf wird am Ende vom Kapitel 3 eine Verifizierung des entwickelten Systems durchgeführt. Im Kapitel 4 werden mehrere identifizierte AWM Aufgaben für ein ZFT System vorgestellt. Bei der Vorstellung einer Aufgabe wird zunächst das zu lösende Problem beschrieben und anschließend ein Encoding des Problems angegeben, welches zudem ausführlich validiert wird. Da nicht alle identifizierten Aufgaben aus zeitlichen Gründen umgesetzt werden konnten, wird im Kapitel 4 zur rationalen Entscheidungshilfe eine Nutzwertanalyse durchgeführt. Im Kapitel 4 werden zudem die Evaluierungsergebnisse der ausgewählten Aufgaben analysiert. Abschließend wird im Kapitel 5 der Mehrwert der Antwortmengenprogrammierung gegenüber dem bereits bestehenden Ansatz ermittelt.

Kapitel 2

Grundlagen

In diesem Kapitel werden die für die Arbeit notwendigen Grundlagen aufgearbeitet. Ein zentraler Bestandteil dieser Arbeit ist die Antwortmengenprogrammierung, die den Fördereinheiten im ZFT System mehr Intelligenz verschaffen soll. Deshalb werden sowohl wichtige Eigenschaften als auch die Arbeitsweise von AWM im Abschnitt 2.1 detailliert betrachtet. Außerdem wird im Abschnitt 2.2 der Ablauf und die Arbeitsweise von zellularen Fördertechnik Systemen vorgestellt. Dabei ist ein zentrales Merkmal von einem ZFT System, dass für die Kommunikation und Steuerung des Systems ein dezentraler Ansatz verwendet wird [19]. Dieser dezentrale Ansatz wird mit Hilfe von einem Multiagentensystem realisiert. Aus diesem Grund werden im Abschnitt 2.3 die Grundlagen von Multiagentensystemen und Agenten aufgearbeitet.

2.1 Antwortmengenprogrammierung

Die Antwortmengenprogrammierung ist ein Ansatz der deklarativen Programmierung. In der traditionellen Programmierung werden Probleme gelöst, indem einem Computer genau gesagt wird, wie er das Problem lösen soll. Im Gegensatz dazu wird bei der deklarativen Programmierung das Problem beschrieben und dem Computer überlassen, wie er das Problem löst. AWM trennt stark zwischen Logik und Kontrolle und erfüllt damit die ursprüngliche Motivation der logischen Programmierung. Zudem stehen bei der Modellierung zwei verschiedene Negationsoperatoren zur Verfügung, wodurch Unsicherheit, fehlendes Wissen und definitives Unwissen modelliert werden kann. Dabei bietet AWM eine Kombination aus einer mächtigen und leichtverständlichen Problemmodellierung und effizienten Solvern [16, S. 1-2].

Bei der klassischen logischen Programmierung gibt es eine Restriktion der Syntax, sodass nur Regeln, die in ihrem Bedingungsteil ausschließlich Atome enthalten und deren Folgerungsteil aus höchstens einem Atom besteht, zugelassen sind. Diese Restriktion hat starke Auswirkungen auf das Antwortverhalten [9, S.272]. Diese syntaktische Einschränkung auf Regeln wird bei der erweiterten logischen Programmierung aufgehoben. Damit können Regeln neben der Defaultnegation (*not*) auch die strenge, logische Negation ($\neg$) enthalten. Mit dieser Erweiterung kann unvollständige Information angemessen dargestellt und verarbeitet werden und wird nicht mit definitiven Nichtwissen verwechselt [9]. Sei B ein Atom, dann können durch die beiden Negationsoperatoren *not* und $\neg$, die Fälle „es gibt

keinen Nachweis für dieses Atom“ (*not* B) und „es gibt einen Nachweis, dass das Atom nicht wahr ist“ ($\neg$ B) unterschieden werden. Mit der Erweiterung um die Negation entsteht das Problem, dass logische Programme, welche Negation enthalten, möglicherweise mehrere Modelle oder auch gar kein sinnvolles Modell haben. Diese Problem entsteht, da ein Auftreten von Negation im Allgemeinen zur Folge hat, dass nicht mehr ein eindeutiges Modell die Semantik eines logischen Programms bestimmt [9, S.283].

Durch die Erhöhung der Ausdrucksfähigkeit logischer Programme wird eine wesentlich kompliziertere Semantik benötigt. Als Semantik für erweiterte logische Programme wird die *Antwortmengensemantik* verwendet [9, S.272]. Im Folgenden soll zuerst die Syntax von erweiterten logischen Programmen festgelegt und anschließend die formale Semantik beschrieben werden. Ein erweitertes logisches Programm P ist eine endliche Menge von Regeln der Form:

$$r : H \leftarrow A_1, ..., A_n, not\ B_1, ..., not\ B_m. \tag{2.1}$$

Wobei H, A_1,..., A_n, B_1,..., B_m Literale sind. Dabei ist ein Literal eine atomare Formel oder die Negation einer atomaren Formel. Sei $head(r) = \{H\}$ der Kopf, $pos(r) = \{A_1, ..., A_n\}$ die positiven Rumpfliterale und $neg(r) = \{B_1, ..., B_m\}$ die negativen Rumpfliterale der Regel r. Eine Regel r kann dann umgangssprachlich so gelesen werden, dass $head(r)$ wahr ist, wenn die Literale $A_1, ..., A_n$ wahr und die Literale $B_1, ..., B_m$ möglicherweise falsch sind [16, S. 13]. Ein Literal B_i mit $i \in \{1, ..., m\}$ ist möglicherweise falsch, solange nicht bewiesen wurde, dass B_i wahr ist. Für die Semantik wird festgelegt, dass Mengen von Grundliteralen für die Definition der Modelle erweiterter logischer Programme verwendet werden. Diese Mengen werden als Antwortmengen bezeichnet [9, S.284]. Dabei ist ein Grundliteral ein Grundatom oder die Negation eines Grundatoms. Ein Atom A ist durch die Formel $A \equiv P(t_1, ...t_n)$ definiert, wobei P ein n-stelliges Prädikatensymbol und $t_1, ..., t_n$ Terme sind [18]. Sind alle Terme eines Atoms Konstanten, dann wird es als Grundatom bezeichnet. Im Folgenden wird eine Antwortmenge sowie die dafür benötigten Eigenschaften und Konzepte formal definiert. Die beiden Literale $P(t_1, ...t_n)$ und $\neg P(t_1, ...t_n)$ werden *komplementär* genannt. Ist l ein Literal, dann wird das dazu komplementäre Literal mit $\bar{l}$ bezeichnet. Eine Menge von Grundliteralen heißt *konsistent*, wenn sie keine komplementären Literale enthält. Eine konsistente Menge von Grundliteralen wird als *Zustand* bezeichnet [9, S.284].

Für die Definition einer Antwortmenge wird die *Gelfond-Lifschitz-Reduktion* benötigt, welche das Ziel hat, die Defaultnegation in einem erweiterten logischen Programm zu eliminieren. Sei P ein erweitertes logisches Programm und S ein Zustand, dann ist P^S das Redukt von P bezüglich S und ist wie folgt definiert:

$$P^S \ = \ \{head(r) \leftarrow pos(r). \mid r \in P, neg(r) \cap S = \emptyset\} \tag{2.2}$$

Das Redukt P^S entsteht aus P in zwei Schritten. Im ersten Schritt werden alle Regeln, deren Rumpf ein negatives Literal *not* B mit $B \in S$ enthält, entfernt. Im zweiten Schritt werden aus allen übrig gebliebenen Regeln alle negativen Rumpfliterale entfernt. Damit ist P^S ein logisches Programm ohne Defaultnegation. Das Redukt hängt dabei stark vom Zustand S ab [9, S.286].

Im Folgenden wird der Konsequenzoperator $Cl(P)$, wobei P ein erweitertes logisches Programm ist, formal eingeführt. Der Konsequenzoperator spielt eine entscheidende Rolle in der Definition einer Antwortmenge. Sei P ein erweitertes logisches Programm, in dessen Regeln keine Defaultnegation vorkommt und S ein Zustand. Dann heißt S *geschlossen*

unter P, wenn für jede Regel r aus P gilt: Ist $pos(r) \subseteq S$, so ist $head(r) \cap S \neq \emptyset$. Der Durchschnitt zweier geschlossener Zustände ist wieder geschlossen. Zu einem erweiterten logischen Programm P ohne Defaultnegation gibt es also höchstens einen minimalen geschlossenen Zustand, welcher mit $Cl(P)$ bezeichnet wird, falls dieser existiert [9, S.286].

Nachdem alle Eigenschaften und Konzepte formal definiert wurden, welche für die Definition einer Antwortmenge benötigt werden, wird im Folgenden eine Antwortmenge formal definiert. Sei S ein Zustand und P ein erweitertes logisches Programm, dann heißt S *Antwortmenge* von P, wenn S minimal geschlossen unter dem Redukt P^S ist, das heißt wenn gilt $S = Cl(P^S)$ [9, S.291-292].

Für eine Antwortmenge S und ein Grundliteral A gibt es drei Möglichkeiten. Wenn $A \in S$, dann ist A wahr in S. Ist $\bar{A} \in S$, dann ist A falsch in S. Wenn sowohl $A \notin S$ und $\bar{A} \notin S$, dann wird dies als Nichtwissen interpretiert. Die *Antwortmengensemantik* wird im Folgenden durch die Inferenzrelation $\models$ definiert. Dafür sei P ein erweitertes logisches Programm und A ein Literal.

$$P \models A \textit{ genau dann wenn } A \textit{ wahr in allen Antwortmengen von } P. \tag{2.3}$$

Bei einer Anfrage Q, wobei Q ein Literal ist, an P antwortet die Antwortmengensemantik mit *yes*, falls $P \models Q$, mit *no* falls $P \models \bar{Q}$ und *unknown* in allen anderen Fällen [9, S.292]. Damit macht die Antwortmengensemantik eine Unterscheidung zwischen einer Anfrage die fehlschlägt, weil sie nicht bewiesen werden kann und einer Anfrage die fehlschlägt, weil ihre Negation bewiesen werden kann, möglich. Somit gibt es in der Semantik einen Unterschied zwischen sicherem Nichtwissen und purer Unwissenheit [9, S.291].

2.1.1 Antwortmengenprogrammierung Syntaxerweiterung

Im Folgenden wird die AWM Syntax durch mehrere Sprachkonstrukte erweitert, um die Modellierung zu erleichtern. Für die nachfolgenden Regeln wird keine neue Semantik erstellt, sondern die Regeln werden in erweiterte logische Regeln übersetzt.

Ein **Constraint** eliminiert unerwünschte Lösungskandidaten. Dafür werden alle Antwortmengen entfernt, die die Rumpfliterale des Constraints erfüllen. Ein Constraint hat die Form

$$\leftarrow A_1, ..., A_n, not\ B_1, ..., not\ B_m. \tag{2.4}$$

wobei $A_1, ..., A_n, B_1, ..., B_m$ Literale sind. Der Constraint (2.4) kann mit Hilfe der Regel:

$$X \leftarrow A_1, ..., A_n, not\ B_1, ..., not\ B_m, not\ X.$$

in ein erweitertes logisches Programm übersetzt werden. Dabei ist X ein neues Literal, welches in keiner Regel des Programms, in dem sich der Constraint befindet, vorkommt [16, S. 17].

Eine **Auswahlregel** liefert die Möglichkeit der Auswahl von Teilmengen aus einer Menge von Literalen. Jede Teilmenge von dem Kopf der Regel kann Teil einer Antwortmenge sein, wenn die Rumpfliterale erfüllt sind. Eine Auswahlregel hat die Form

$$\{H_1, ..., H_k\} \leftarrow A_1, ..., A_n, not\ B_1, ..., not\ B_m. \tag{2.5}$$

wobei $A_1, ..., A_n, B_1, ..., B_m, H_1, ..., H_k$ Literale sind. Die Regel (2.5) kann in $2k+1$ erweiterte logische Regeln übersetzt werden.

$$H' \leftarrow A_1, ..., A_n, not\ B_1, ..., not\ B_m.$$
$$H_1 \leftarrow H', not\ H'_1. \quad ... \quad H_k \leftarrow H', not\ H'_k.$$
$$H'_1 \leftarrow not\ H_1. \quad ... \quad H'_k \leftarrow not\ H_k.$$

Dabei sind $H', H'_1, ..., H'_k$ neue Literale, welche in keiner Regel des Programms, in dem sich die Auswahlregel befindet, vorkommen. Bei dieser Regel und den folgenden Regeln müssen die Antwortmengen, welche mit den übersetzten Regeln berechnet wurden, anschließend mit den Literalen des Programms ohne Übersetzung geschnitten werden [16, S. 18].

Eine **Kardinalitätsregel** bietet die Möglichkeit, dass eine Regel erfüllt ist, wenn bereits l Literale des Rumpfes wahr sind. Damit gehört das Kopfliteral zu dem stabilen Modell, wenn mindestens l Rumpfliterale erfüllt sind. Eine Kardinalitätsregel hat die Form

$$H \leftarrow l\{A_1, ..., A_n, not\ B_{n+1}, ..., not\ B_m\}. \tag{2.6}$$

wobei $H, A_1, ..., A_n, B_{n+1}, ..., B_m$ Literale sind und l ein positiver Integer mit $0 \leq l \leq m$ ist. Auch die Kardinalitätsregel kann in eine Menge von erweiterten logischen Regeln übersetzt werden. Für die Übersetzung wird ein neues Atom $ctr(i, j)$ eingeführt. Dabei repräsentiert dieses Atom den Fakt, dass mindestens j Literale, welche einen Index größer oder gleich i haben, in einer Antwortmenge enthalten sind. Dann wird eine Regel der Form (2.6) wie folgt ersetzt.

$$\begin{aligned}
H &\leftarrow ctr(1, l). \\
ctr(i, k+1) &\leftarrow ctr(i+1, k), A_i. \\
ctr(i, k) &\leftarrow ctr(i+1, k). && (1 \leq i \leq n) \\
ctr(j, k+1) &\leftarrow ctr(j+1, k), not\ B_j. \\
ctr(j, k) &\leftarrow ctr(j+1, k). && (n+1 \leq j \leq m) \\
ctr(m+1, 0).&
\end{aligned}$$

Dabei gilt $0 \leq k \leq l$ und es wird angenommen, dass das Atom $ctr/2$ in keiner Regel des Programms, in dem sich die Kardinalitätsregel befindet, vorkommt [16, S. 18-19].

Im Folgenden wird eine Regel betrachtet, welche erfüllt ist, wenn mindestens l und höchstens u Rumpfliterale erfüllt sind. Eine solche Regel hat die Form

$$H \leftarrow l\{A_1, ..., A_n, not\ B_{n+1}, ..., not\ B_m\}u. \tag{2.7}$$

wobei $A_1, ..., A_n, B_{n+1}, ..., B_m$ Literale sind. Außerdem sind l und u positive Integer mit $0 \leq l \leq u \leq m$. Eine Regel der Form (2.7) kann wie folgt übersetzt werden.

$$H \leftarrow B, not\ C.$$
$$B \leftarrow l\{A_1, ..., A_n, not\ B_{n+1}, ..., not\ B_m\}.$$
$$C \leftarrow u+1\{A_1, ..., A_n, not\ B_{n+1}, ..., not\ B_m\}.$$

Dabei wird angenommen, dass die Atome B und C in keiner Regel des Programms, in dem sich die Kardinalitätsregel befindet, vorkommen [16, S. 20].

Als nächstes werden Regeln betrachtet, welche die Kardinalitätsbeschränkung im Kopf der Regel haben. Die Regel sagt aus, dass jede Teilmenge von dem Kopf der Regel, welche mehr als $l-1$ und weniger als $u+1$ Literale enthält, Teil einer Antwortmenge sein kann, wenn die Rumpfliterale erfüllt sind. Eine solche Regel hat die Form

$$l\{C_1, ..., C_q, not\ D_{q+1}, ..., not\ D_p\}u \leftarrow A_1, ..., A_n, not\ B_{n+1}, ..., not\ B_m. \tag{2.8}$$

wobei $C_1, ..., C_q, D_{q+1}, ..., D_p, A_1, ..., A_n, B_{n+1}, ..., B_m$ Literale sind. Außerdem sind l und u positive Integer mit $0 \leq l \leq u \leq p$. Eine Regel der Form (2.8) kann wie folgt übersetzt werden.

$$\begin{aligned} E &\leftarrow A_1, ..., A_n, not\ B_{n+1}, ..., not\ B_m. \\ \{C_1, ..., C_q\} &\leftarrow E. \\ F &\leftarrow l\{C_1, ..., C_q, not\ D_{q+1}, ..., not\ D_p\}u. \\ &\leftarrow E, not\ F. \end{aligned}$$

Dabei wird angenommen, dass die Atome E und F in keiner Regel des Programms, in dem sich die Kardinalitätsregel befindet, vorkommen [16, S. 20].

2.1.2 Antwortmengenprogrammierung Prozess

Der vollständige AWM Prozess kann anhand der Abbildung 2.1 nachvollzogen werden.

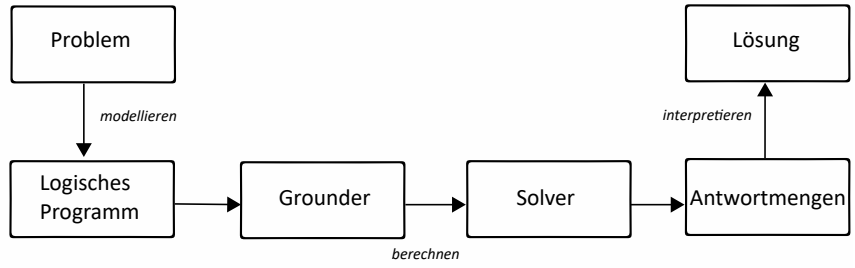


Abbildung 2.1: Der vollständige AWM Prozess [16, S. 3].

Dabei wird ein Problem mit Hilfe von erweiterten logischen Regeln zu einem Programm modelliert. Das entstandene logische Programm wird an einen Grounder übergeben. Die Hauptaufgabe des Grounders ist es, die Variablen in dem übergebenen logischen Programm P durch Konstanten zu ersetzen, sodass das Ergebnis ein äquivalentes Programm P' ist, welches nur aus Grundliteralen besteht. Die Programme P und P' sind äquivalent, falls sie dieselben Antwortmengen haben [4]. P' wird nun an einen Solver übergeben, welcher die Antwortmengen des Programms berechnet. Die Lösungen für das Problem werden durch die berechneten Antwortmengen repräsentiert und können durch eine geeignete Interpretation extrahiert werden.

2.1.3 Antwortmengenprogrammierung vs. Prolog

Auch wenn sich AWM und Prolog Programme stark ähneln, gibt es fundamentale Unterschiede in der Berechnung von Lösungen. Prolog berechnet seine Antworten mit Hilfe von einem top-down Ansatz, wohingegen AWM ein bottom-up Ansatz verwendet, um die Antwortmengen zu berechnen. Zudem ist Prolog eine vollständige Programmiersprache und gibt dem Benutzer Kontrolle darüber, wie das Programm ausgeführt wird. Dabei kann der Benutzer sowohl über die Reihenfolge der Regeln im Programm, als auch über die Reihenfolge der Bedingungen in einer Regel die Auswertung einer Anfrage beeinflussen [11]. Bei AWM ist die Logik stark von der Ausführung getrennt und der Benutzer kann sein Problem modellieren, hat dann aber keine Kontrolle darüber, wie die Lösungen berechnet werden [16, S. 1-2]. Außerdem bietet Prolog im Gegensatz zu AWM keine Möglichkeit der starken Negation bei der Modellierung [23, S. 2]. Prolog stellt einen *not*-Operator zur Verfügung, welcher eine Aussage als wahr annimmt, falls ihre Ableitung scheitert. Dabei wird die negative Information im Gegensatz zu AWM rein prozedural behandelt [9, S.283].

2.1.4 Lua

Lua ist eine mächtige, effiziente und eingebettete Skriptsprache [7]. Sie ist standardmäßig in der vierten Version von dem AWM Solver *clingo* verfügbar. Mit Hilfe von Lua ist es möglich, stärker in den Grounding Prozess einzugreifen [17]. Dabei können in Lua Funktionen entwickelt werden, die aus dem AWM Programm mit Hilfe von *@Funktionsname*(*Parameterliste*) aufgerufen werden können. Das Ergebnis der Funktion wird an diese Stelle zurückgeschrieben. Mit Hilfe von Lua können AWM Programme effizienter gestaltet werden.

2.2 Zellulare Fördertechnik Systeme

Wie in der Einleitung bereits erwähnt, etablieren sich aufgrund ihrer Flexibilität und Schnelligkeit zellulare Fördersysteme in der Logistik, da durch wechselnde Leistungsanforderungen der innerbetriebliche Materialfluss schnell und flexibel auf diese Anforderungen reagieren muss. Dabei stoßen die klassischen Fördertechniken an ihre Grenzen. Aus diesem Grund wurden zellulare Fördersysteme entwickelt, welche auf autonomen fördertechnischen Entitäten basieren [19, S. 1]. Das Fraunhofer IML in Dortmund hat ein ZFT System umgesetzt und verwendet als fördertechnische Entitäten autonome Transportfahrzeuge, die als *Multishuttle Move* bezeichnet werden [19, S. 3]. Dabei sind ZFT Systeme sehr flexibel, da sie in der Lage sind sich ohne manuelle Eingriffe an neue Gegebenheiten anzupassen. Außerdem kann jederzeit manuell die Anordnung der Transportentitäten, wie zum Beispiel das Streckennetz, geändert werden. Zudem arbeiten die autonomen Entitäten selbständig, durch die Kommunikation mit der Umgebung und anderen Entitäten des Systems, an der Erfüllung der vordefinierten Ziele. Die Kommunikation und die Steuerung erfolgt durch ein Multiagentensystem [19, S. 4]. Im Folgenden wird anhand des umgesetzten ZFT Systems des Fraunhofer IML in Dortmund der Aufbau und die Arbeitsweise eines ZFT Systems verdeutlicht.

2.2.1 Aufbau der zellularen Fördertechnik Systeme

Die Hauptkomponenten eines ZFT Systems sind die Regale, die Transportsysteme und die Kommissionierstationen. Anhand der Abbildung 2.2 kann der Aufbau des ZFT Systems des Fraunhofer IML in Dortmund nachvollzogen werden. In einem Regal befindet sich ein Schienensystem, wodurch sich die Transportsysteme im Regal bewegen können. Des Weiteren besteht das Regal aus mehreren Ebenen, die durch Lifte angefahren werden können. Ein Regal hat zwei Regalseiten, welche durch eine Spur, für die Beförderung der Transportsysteme, getrennt sind. Jede Regalseite besteht aus Stellplätzen für die Ladungsträger, die Waren enthalten.

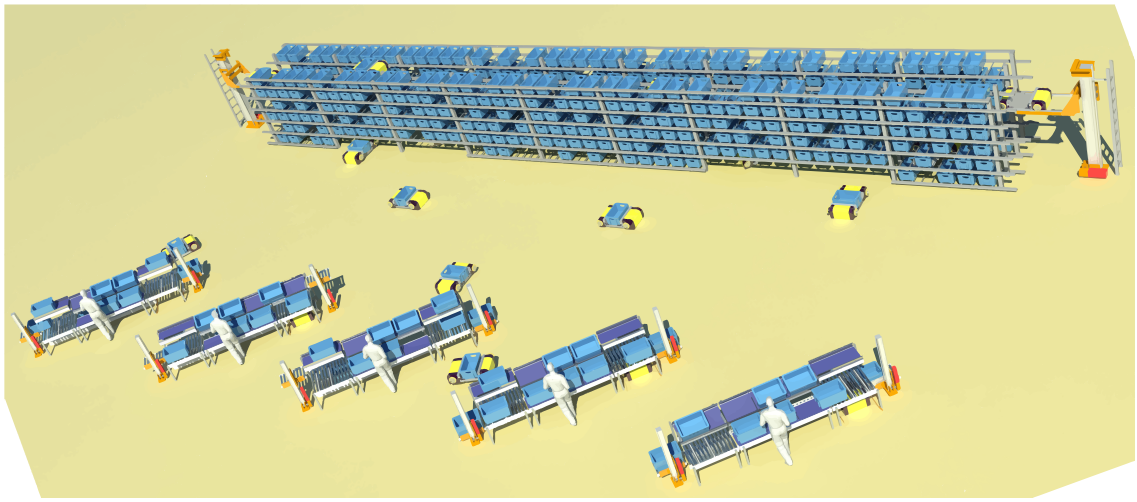


Abbildung 2.2: Ein Überblick über das System [8].

Die Transportsysteme sind im ZFT System für den Transport von Ladungsträgern verantwortlich. Ihre Aufgabe besteht im Speziellen darin, die Ladungsträger im Regalsystem aufzunehmen, zur Kommissionierstation zu befördern und den Ladungsträger anschließend zu seinem vorherigen Standort zurück zu befördern. Um diese Aufgaben ausführen zu können und sich dabei sowohl sicher als auch effizient in der Umgebung zu bewegen, verfügen sie über eine Vielzahl von Sensoren. Zudem sind die Transportsysteme in der Lage, mit anderen Entitäten der Umgebung zu kommunizieren.

An einer Kommissionierstation befindet sich ein Kommissionierer, der einen Auftrag zugewiesen bekommen hat. Dabei ist ein Auftrag eine Menge von Positionen. Eine Position enthält eine eindeutige Beschreibung von einem Artikel aus dem Sortiment und wie häufig dieser benötigt wird. Außerdem besteht ein Job aus einer Position und einer eindeutigen ID, welche als Job_ID bezeichnet wird und den Job eindeutig beschreibt. Die Transportsysteme befördern die Ladungsträger mit den Waren für die einzelnen Positionen des Auftrags zu den Kommissionierstationen. Der Kommissionierer entnimmt dann die benötigte Anzahl der Artikel aus dem Ladungsträger und arbeitet den Auftrag positionsweise ab.

2.2.2 Arbeitsweise der zellularen Fördertechnik Systeme

Im ZFT System liegen bei einem zentralen Verwalter alle Aufträge vor. Die Positionen der Aufträge werden zu Jobs umgeformt und mit Hilfe einer Auktion den jeweiligen Transportsystemen zugewiesen. Gewinnt ein Transportsystem eine Auktion, ist es dafür zuständig, dass der Ladungsträger, der den Artikel für den Job enthält, aus dem Regal zu der ihm zugewiesenen Kommissionierstation befördert wird. Zusätzlich ist es dafür verantwortlich, dass sich der Ladungsträger am Ende des Jobs wieder an seiner vorgesehenen Stelle im Regal befindet. Das Transportsystem ermittelt zu Beginn, an welchem Stellplatz im Regal sich der Ladungsträger befindet und kontaktiert den Lift des Regals, um auf die Ebene des Ladungsträgers zu gelangen. Im Regal fährt das Transportsystem zu dem Ladungsträger und lädt diesen auf. Anschließend fährt es zu dem zweiten Lift des Regals, um auf den Hallenboden zu gelangen. Von dort fährt es auf dem schnellsten Weg zu der zugewiesenen Kommissionierstation und liefert den Ladungsträger ab. Der Kommissionierer entnimmt die Ware und gibt den Ladungsträger wieder frei. Das Transportsystem wartet auf die Freigabe des Ladungsträgers und nimmt diesen dann erneut auf. Danach fährt es auf direktem Weg zum Regal, um den Ladungsträger an seinen vorherigen Ort zurück zu befördern. Mit dem Abladen des Ladungsträgers an dessen vorgesehenen Standort ist der Job des Transportsystems beendet. Der Auftrag ist jedoch erst fertig, wenn alle seine Positionen beendet wurden.

2.3 Multiagentensysteme

Ein Multiagentensystem ist ein System, in dem sich mehrere Agenten in derselben Umgebung befinden, die miteinander interagieren [27, S. 9]. Dabei sind Agenten Computersysteme, die im Interesse ihres Benutzers selbstständig Aktionen ausführen. Einem Agenten muss nicht zu jedem Zeitpunkt genau gesagt werden was er tun soll, sondern er wählt selbst seine Aktionen, um seine definierten Ziele zu erreichen [27, S.19]. Agenten befinden sich in einer Umgebung, welche sie in einer bestimmten Form wahrnehmen. Auf Basis dieser Wahrnehmung stehen ihnen eine Menge von Aktionen zur Verfügung, mit dessen Hilfe sie ihre Umgebung verändern können [27, S. 16].

2.3.1 Abstrakte Architektur für Agenten

Im Folgenden wird eine abstrakte Architektur für Agenten formalisiert. Eine **Umgebung** sei ein Element aus einer Menge $E = \{e, e', ...\}$, wobei E aus allen möglichen diskreten Zuständen der Umgebung, in der sich der Agent befindet, besteht [27, S. 31]. Bei einem Schachspiel würde E zum Beispiel aus allen möglichen Positionen der Schachfiguren bestehen. Agenten haben die Möglichkeit, über ihre **Aktionen** den Zustand der Umgebung, in der sie sich befinden, zu verändern. Dabei stehen dem Agenten eine endliche Menge von möglichen Aktionen zur Verfügung, welche durch die Menge $Ac = \{\alpha, \alpha', ...\}$ repräsentiert wird. Ein Agent kann in jedem Umgebungszustand genau eine Aktion ausführen und nach der Ausführung befindet sich die Umgebung wieder in genau einem Zustand [27, S. 31].

Im Basismodell eines Agenten läuft die Interaktion eines Agenten mit der Umgebung, in der er sich befindet, wie folgt ab. Die Umgebung hat einen Startzustand, aufgrund des-

sen der Agent eine Aktion ausführt, die den Zustand der Umgebung verändert. Dann wählt der Agent auf Basis des veränderten Zustandes eine neue Aktion aus und die Umgebung ändert sich erneut. Dieser Prozess setzt sich endlos fort und wird formal als Lauf definiert. Ein **Lauf** r von einem Agenten in einer Umgebung ist eine abwechselnde Folge von Umgebungszuständen und Aktionen:

$$r : e_0 \xrightarrow{\alpha_0} e_1 \xrightarrow{\alpha_1} e_2 \xrightarrow{\alpha_2} ... \xrightarrow{\alpha_{l-1}} e_l ... \tag{2.9}$$

Sei $\mathcal{R}$ die Menge aller Läufe über E und Ac, $\mathcal{R}^{Ac}$ die Menge aller Läufe, die mit einer Aktion enden und $\mathcal{R}^{E}$ die Menge aller Läufe, die mit einem Umgebungszustand enden [27, S. 31]. Der Effekt, den eine Aktion eines Agenten auf eine Umgebung hat, wird durch die **Zustandstransformationsfunktion**

$$\tau : \mathcal{R}^{Ac} \rightarrow 2^{E} \tag{2.10}$$

definiert [27, S. 32]. Die Funktion ordnet jedem Lauf, der mit einer Aktion endet, eine Menge von allen möglichen Umgebungszustände zu, die nach der Ausführung der Aktion entstehen können. Formal wird eine **Umgebung** Env dann als ein Tripel $Env = \langle E, e_0, \tau \rangle$ definiert, wobei E eine Menge von Umgebungszuständen, e_0 der Startzustand der Umgebung und τ eine Zustandstransformationsfunktion ist [27, S. 32]. Nach der formalen Definition einer Umgebung wird eine formale Definition für Agenten benötigt, die mit der Umgebung interagieren. Ein **Agent** Ag wird als eine Funktion definiert, die einen Lauf, der mit einem Umgebungszustand endet, auf eine Aktion abbildet:

$$Ag : \mathcal{R}^{E} \rightarrow Ac \tag{2.11}$$

Damit treffen Agenten die Entscheidungen welche Aktion sie ausführen aufgrund ihrer Vergangenheit [27, S. 34]. Ein **System** enthält ein Agenten und eine Umgebung, dabei bezeichnet $\mathcal{R}(Ag, Env)$ alle möglichen Läufe, die der Agent Ag in der Umgebung Env generieren kann. Eine Folge $(e_0, \alpha_0, e_1, \alpha_1, e_2, ...)$ repräsentiert den Lauf eines Agenten Ag in der Umgebung $Env = \langle E, e_0, \tau \rangle$, wenn e_0 der initiale Zustand von Env, $\alpha_0 = Ag(e_0)$ sowie für alle $l > 0$ gilt, dass $e_l \in \tau((e_0, \alpha_0, ..., \alpha_{l-1}))$ und $a_l = Ag((e_0, \alpha_0, ..., e_{l-1}))$ ist [27, S. 32].

2.3.2 Verfeinerung der abstrakten Architektur

Im Folgenden wird die abstrakte Architektur für intelligente Agenten verfeinert, da die starke Abstraktion keinerlei Informationen bezüglich der Umsetzung eines Agenten bietet. Ein Agent wählt seine Entscheidung aufgrund seiner Wahrnehmung der Umgebung. Die Möglichkeit für den Agenten seine Umgebung zu beobachten wird mit Hilfe seiner *see* Funktion realisiert. Sei Per eine nicht leere Menge von Wahrnehmungen, dann bildet die *see* Funktion

$$see : E \rightarrow Per \tag{2.12}$$

ein Umgebungszustand auf eine Wahrnehmung ab. Die *see* Funktion kann für einen Agenten, welcher sich in der physikalischen Welt befindet, zum Beispiel durch eine Videokamera oder einen Infrarotsensor realisiert werden [27, S. 34].

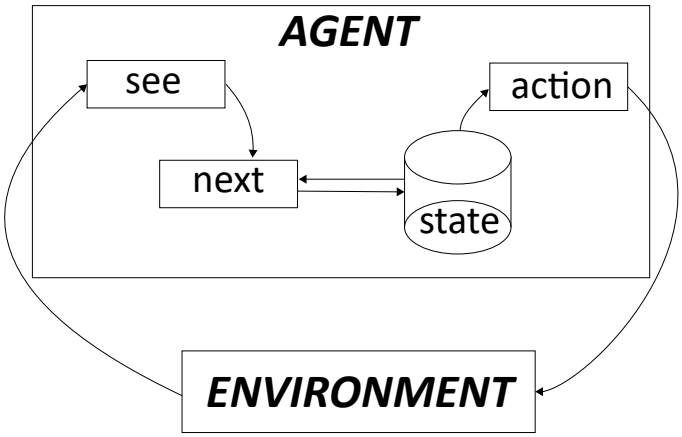


Abbildung 2.3: Agent mit Zustand [27, S. 36].

In einem verfeinerten Agentenmodell haben die Agenten einen internen Zustand. Dieser Zustand verarbeitet die Wahrnehmungen von außen und trifft Entscheidungen darüber, welche Aktion ausgeführt wird. Der Aufbau eines solchen Agenten kann mit Hilfe von Abbildung 2.3 visualisiert werden. Sei I die Menge aller internen Zustände eines Agenten, dann wird die nächste Aktion über die folgende *action* Funktion bestimmt [27, S. 36].

$$action : I \rightarrow Ac \tag{2.13}$$

Um die Wahrnehmungen von außen zu verarbeiten, wird der nächste Zustand mit Hilfe der *next* Funktion bestimmt.

$$next : I \times Per \rightarrow I \tag{2.14}$$

Dabei wird mit dem aktuellen Zustand des Agenten und der aktuellen Wahrnehmung ein neuer Zustand des Agenten berechnet [27, S. 36]. Ein zustandsbasierter Agent verhält sich zyklisch, dabei kann sein Verhalten wie folgt festgehalten werden. Sei $i0$ der Startzustand des Agenten. Der Agent beobachtet seine Umgebung e, welche er mittels $see(e)$ wahrnimmt. Dann aktualisiert er mit der Funktion $next(i0, see(e))$ seinen internen Zustand und berechnet seine nächste Aktion, indem er $action(next(i0, see(e)))$ ausführt. Die ausgeführte Aktion führt zu einem neuen Umgebungszustand und der Prozess startet erneut [27, S. 36].

2.3.3 Umgebungen

Jede Umgebung hat bestimmte Merkmale, durch die sie beschrieben und klassifiziert werden kann. Nach Stuart Russel und Peter Norvig können Umgebungen nach folgenden Eigenschaften klassifiziert werden [22, S. 42-44].

- **Zugänglich vs. unzugänglich**
  In einer zugänglichen Umgebung kann der Agent zu jeder Zeit vollständige und konkrete Informationen über den Zustand der Umgebung erhalten.
- **Deterministisch vs. nicht-deterministisch**
  In einer deterministischen Umgebung hat jede Aktion nur genau einen garantierten Effekt. Damit besteht keine Unsicherheit über den Zustand nach der Ausführung einer Aktion.

- **Statisch vs. dynamisch**
  Eine statische Umgebung kann nur durch Aktionen des Agenten verändert werden, wohingegen sich in einer dynamischen Umgebung die Umgebung auch ohne eine Aktion des Agenten ändern kann.

- **Diskret vs. kontinuierlich**
  Eine Umgebung ist diskret, wenn die Anzahlen der Aktionen und Wahrnehmungen in der Umgebung endlich sind.

2.3.4 Intelligente Agenten

Ein einfaches Thermostat ist bereits ein Agent, jedoch wird dieses im Allgemeinen nicht als intelligent betrachtet. Ein Agent muss deshalb nach Wooldridge und Jennings die folgenden Eigenschaften erfüllen, um als intelligent betrachtet zu werden [27, S. 23].

- **Reaktivität**
  Der Agent ist in der Lage, Veränderungen in der Umgebung wahrzunehmen und in einer angemessenen Zeit auf diese Änderung zu reagieren, um seine Ziele zu erreichen.

- **Proaktivität**
  Der Agent übernimmt selbst die Initiative, um seine Ziele zu erreichen. Er arbeitet somit aktiv an der Erfüllung seiner Ziele.

- **Soziale Kompetenz**
  Der Agent ist in der Lage, mit anderen Agenten in Interaktion zu treten.

2.3.5 Java Agent Development Framework

Java Agent Development Framework oder kurz JADE, ist ein Software Framework für die erleichterte Implementierung von Multiagentensystemen in Java. JADE ist ein open-source Projekt, welches ursprünglich von der Telecom Italia entwickelt wurde. Dabei folgt JADE den Standards der *Foundation for Intelligent Physical Agents (FIPA)*. Die Implementierung in JADE wird durch vorgegebene und erweiterbare Agenten Modelle vereinfacht. Zudem stehen dem Entwickler mehrere Werkzeuge für die Verwaltung und für das Testen von den implementierten Multiagentensystemen zur Verfügung [10].

Kapitel 3

Eine Systemarchitektur zur Verwendung von Wissensrepräsentation innerhalb eines Multiagentensystems

In diesem Kapitel wird eine entwickelte Systemarchitektur vorgestellt, mit welcher verschiedene Arten der Wissensrepräsentation in einem Multiagentensystem verwendet werden können. Diese Architektur ist eine Antwort auf die offene Fragestellung bezüglich der Integration von AWM in das ZFT System. Dabei beschreibt eine Systemarchitektur die Struktur des Systems durch Systemkomponenten und ihre Schnittstellen untereinander [5]. Die entwickelte Systemarchitektur ist dabei variabel im Bezug auf den verwendeten Ansatz der Wissensrepräsentation. Das bedeutet, dass mit dieser Architektur neben AWM Aufgaben auch zum Beispiel Prolog Aufgaben im ZFT System gelöst werden können. Diese Variabilität ermöglicht es dann, verschiedene Ansätze der Wissensrepräsentation im ZFT System zu verwenden und zu vergleichen. In diesem Kapitel werden die einzelnen Komponenten der entworfenen Systemarchitektur beschrieben und die Schnittstellen zwischen den Komponenten definiert. Die entworfene Systemarchitektur wurde mit dem Framework aus der Bachelorarbeit [13] umgesetzt. Im Abschnitt 3.3 wird die Umsetzung der Architektur beschrieben.

3.1 Vorstellung der entwickelten Systemarchitektur

Da die zu entwickelnde Systemarchitektur mit verschiedenen Ansätzen der Wissensrepräsentation genutzt werden soll, wird zunächst der abstrakte Ablauf zur Lösungsberechnung für die verschiedenen Ansätze untersucht. Dabei stellt sich heraus, dass alle Ansätze der Wissensrepräsentation dem Ablauf aus Abbildung 3.1 folgen. Bei dem Ablauf wird zunächst das Problem in die benötigte Repräsentation für den verwendeten Ansatz der Wissensrepräsentation modelliert. Bei der Antwortmengenprogrammierung wäre die Repräsentation zum Beispiel ein erweitertes logisches Programm. Anschließend wird mit der Repräsentation eine Lösung für das modellierte Problem berechnet. Man erhält die Lösung für das

ursprüngliche Problem, wenn die berechnete Lösung geeignet interpretiert wird.

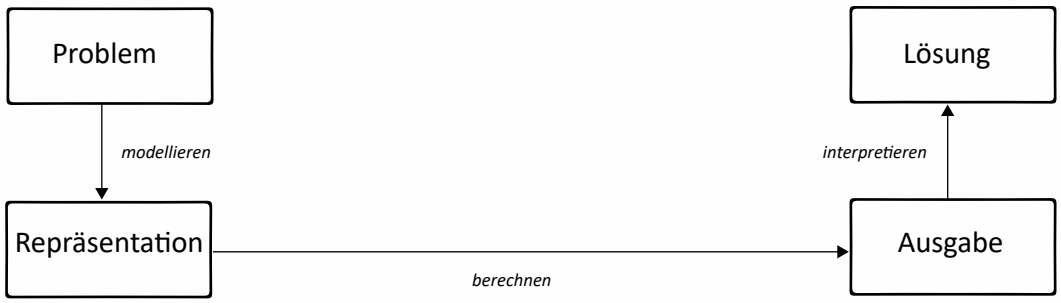


Abbildung 3.1: Der abstrakte Ablauf zur Lösungsberechnung für die verschiedenen Ansätze der Wissensrepräsentation.

Mit der abstrakten Arbeitsweise von den verschiedenen Ansätzen der Wissensrepräsentation wird im Folgenden die benötigte Architektur entwickelt. Zunächst muss dafür die Frage geklärt werden, was in diesem Kontext ein Problem ist. Dabei kann ein Problem als eine zu erledigende Aufgabe in einer Situation betrachtet werden, wobei die Aufgabe durch ein vom Benutzer modelliertes Programm beschrieben und die Situation durch die Eingabedaten für das Programm repräsentiert wird. Bei einem Programm zur Deadlock-Erkennung würde die Aufgabe zum Beispiel durch das vom Benutzer modellierte logische Programm beschrieben. Die Eingabedaten für das Programm, wie zum Beispiel die Positionen der Fahrzeuge im Weggraphen, repräsentieren die relevanten Informationen aus der Situation.

Für die Verwendung der Architektur wird vorausgesetzt, dass jeder Agent im System ein Modul besitzt, in dem alle Aktionen vorhanden sind, die dem Agenten zur Verfügung stehen. Dieses Modul wird im Folgenden als *Aktions-Modul* bezeichnet. Zudem muss jeder Agent über ein Modul für das Senden von Nachrichten an andere Agenten im System und ein Modul zum Empfangen von Nachrichten von anderen Agenten aus dem System verfügen. Diese Module werden im Folgenden als *Sende-Modul* und *Empfangs-Modul* bezeichnet. Die Abbildung 3.2 visualisiert die entwickelte Systemarchitektur, die anhand des Ablauf aus Abbildung 3.1 entworfen wurde.

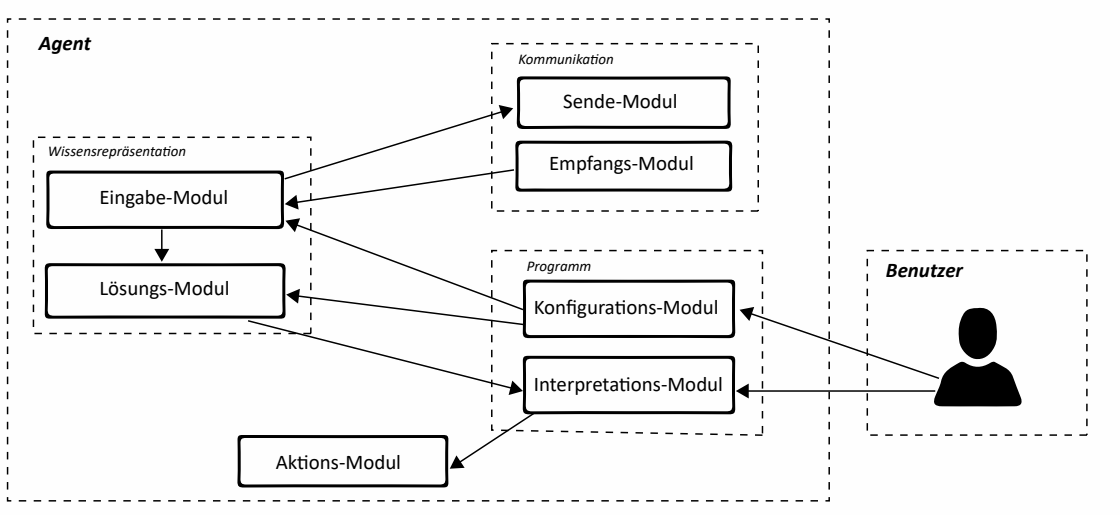


Abbildung 3.2: Die entwickelte Systemarchitektur.

Zunächst wird abstrakt beschrieben, wie mit Hilfe der Architektur das Problem in die Repräsentation modelliert wird. Dabei werden die benötigten Eingabedaten mit dem *Eingabe-Modul* zusammengetragen. Dafür legt der Benutzer im *Konfigurations-Modul* fest, welche Daten benötigt werden und bei welchen Agenten die Daten angefragt werden können. Da es sich um Multiagentensysteme handelt, liegen in den meisten Fällen dem Agenten die Daten nicht direkt vor und der Agent muss diese bei anderen Agenten im System anfragen. Die Anfragen werden über das *Sende-Modul* gesendet und die Antworten über das *Empfangs-Modul* empfangen. Der Benutzer modelliert die zu erledigende Aufgabe in ein Programm und gibt den Pfad zu der Programmdatei im *Konfigurations-Modul* an. Damit wurde der erste Schritt zur Lösungsberechnung aus der Abbildung 3.1 realisiert. Das Problem wird mit Hilfe der angegebenen Module in die benötigte Repräsentation des verwendeten Ansatzes der Wissensrepräsentation modelliert.

Nachdem das Problem geeignet modelliert wurde, soll mit dem verwendeten Ansatz der Wissensrepräsentation eine Lösung für das modellierte Problem berechnet werden. Dafür wird das *Lösungs-Modul* erstellt, das die Eingabedaten aus dem *Eingabe-Modul* sowie die verwendete Art der Wissensrepräsentation und den Pfad zum modellierten Programm aus dem *Konfigurations-Modul* nimmt und damit eine Lösung berechnet.

Um eine Lösung für das ursprüngliche Modul zu erhalten, muss die berechnete Lösung des *Lösungs-Moduls* noch geeignet interpretiert werden. Dafür legt der Benutzer im *Interpretations-Modul* eine geeignete Interpretation für das Programm an. Die Interpretation bestimmt zusammen mit der berechneten Lösung des *Lösungs-Moduls* eine Liste von Aktionen, die der Agent ausführen soll. Die Liste wird an das *Aktions-Modul* weitergegeben, welches diese Aktionen ausführt. Nachdem alle Aktionen umgesetzt wurden, wurde das ursprüngliche Problem gelöst.

Die Architektur besteht aus mehreren Modulen, die über Schnittstellen miteinander verbunden sind. Die Funktionalität der Module wird im Folgenden kurz beschrieben. Die detaillierteren Funktionsweisen und die Schnittstellen der Komponenten werden im Verlaufe des Kapitels genauer betrachtet.

- **Eingabe-Modul**
  Ein Programm benötigt normalerweise Eingabedaten. Da es sich bei dem ZFT System um ein Multiagentensystem handelt, kann es sein, dass der Agent nicht alle benötigten Informationen besitzt und diese bei anderen Agent im System anfragen muss. In diesem Modul werden die benötigten Eingabedaten gesammelt und geeignet zusammengestellt.

- **Lösungs-Modul**
  Ein Agent soll mit Hilfe der Architektur in einer Situation eine Aufgabe lösen. Dabei wird die Situation durch die Eingabedaten und die Aufgabe durch ein vorliegendes Programm repräsentiert. Dieses Modul ermöglicht die Berechnung einer Lösung für die Aufgabe in der Situation und das für verschiedene Ansätze der Wissensrepräsentation.

- **Interpretations-Modul**
  Die berechnete Lösung muss interpretiert werden, sodass der Agent geeignete Aktionen ausführt, um eine Aufgabe in einer vorliegenden Situation zu lösen. In diesem Modul wird die Lösung in eine Liste von Aktionen umgewandelt, mit welcher der Agent die vorgegebene Aufgabe lösen kann.

- **Sende-Modul**
  Ein Agent besitzt eine Sendeeinheit, mit welcher er Nachrichten an andere Agenten aus dem System senden kann. Dieses Modul repräsentiert die Sendeeinheit des Agenten.

- **Empfangs-Modul**
  Ein Agent besitzt eine Empfängereinheit, mit welcher er Nachrichten von anderen Agenten aus dem System empfangen kann. Dieses Modul repräsentiert die Empfängereinheit des Agenten.

- **Aktions-Modul**
  Einem Agenten steht eine Menge von möglichen Aktionen zur Verfügung, mit denen er den Zustand der Umgebung, in der er sich befindet, verändern kann. Alle Aktionen die dem Agenten dabei zur Verfügung stehen, sind in diesem Modul vorhanden.

- **Konfigurations-Modul**
  Die Architektur benötigt mehrere Eingaben vom Benutzer, um variabel und korrekt zu funktionieren. In dem Modul werden die Benutzereingaben entgegengenommen und geeignet gespeichert.

Die einzelnen Module der Architektur werden im Verlauf dieses Kapitels noch detaillierter betrachtet. Für das Verständnis dieser Architektur wird im Folgenden der Ablauf der Architektur anhand eines Beispiels visualisiert, sodass die Funktionsweise der Architektur verdeutlicht wird. Es handelt sich um ein Beispiel aus dem Kontext von ZFT Systemen. In diesem Beispiel bekommt der Fahrzeug Agent $fahrzeug_0$ vom Auktionator eine Aufforderung zum Bieten. Die Aufgabe des Fahrzeugs ist es, für alle verfügbaren Jobs ein Gebot beim Auktionator abzugeben. Für die Berechnung der Gebote kann das Fahrzeug auf das AWM Programm *AWMBieten.lp* zurückgreifen. Dafür benötigt *AWMBieten.lp* die Positionen der Fahrzeuge im System, alle verfügbaren Jobs und die Höhe des Regals als Eingabedaten. Das Programm gibt das Prädikat $biete(J, G)$ aus, wobei J die ID von einem Job ist und G der Betrag der für den Job geboten werden soll. Das Beispiel wird

im Folgenden als *AWMBieten* bezeichnet. In den Beschreibungen der Module wird für das Beispiel der Ablauf von der Berechnung bis hin zur Abgabe der Gebote mit Hilfe der Systemarchitektur beschrieben.

3.1.1 Konfigurations-Modul

Diese Modul nimmt die Eingabedaten für die Architektur vom Benutzer entgegen und speichert diese geeignet ab. Im Folgenden werden die Eingabedaten der Architektur mit der benötigten Syntax angegeben. Ein Programm benötigt normalerweise Eingabedaten. Dabei ist es häufig der Fall, dass der Agent nicht alle benötigten Informationen besitzt und diese bei anderen Agent im System anfragen muss. Dafür muss der Agent wissen, welche Informationen benötigt werden und bei welchen Agenten er diese Informationen anfragen kann. Dafür legt der Benutzer fest, welche Informationen das auszuführende Programm benötigt und welcher Agententyp diese Informationen bereitstellt. Der Benutzer muss diese Eingabe in der folgenden Form angeben:

$$\{InfoId_1 : AgentenId_1, ..., InfoId_n : AgentenId_n\}$$

Dabei ist $InfoId_k$ die ID von einer benötigten Information und $AgentenId_k$ die ID von einem Agententyp der diese Information bereitstellt. Diese Information wird im Folgenden als *InfoAgententyp* bezeichnet. Da die Architektur variabel im Bezug auf den verwendeten Ansatz der Wissensrepräsentation ist, muss der Benutzer am Anfang festlegen, welche Art der Wissensrepräsentation für die Berechnung einer Lösung verwendet werden soll. Der Benutzer muss diese Eingabe in der folgenden Form angeben:

$$ArtId$$

Dabei ist *ArtId* die ID für die verwendete Art der Wissensrepräsentation. Diese Information wird im Folgenden als *VerwendeteArt* bezeichnet. Zudem benötigt die Architektur die Pfade zu den einzelnen Programmdateien. Dabei ist eine Programmdatei das auszuführende Programm für eine bestimmte Art der Wissensrepräsentation. Diese Information wird in folgender Form erwartet:

$$\{ArtId_1 : Pfad_1, ..., ArtId_n : Pfad_n\}$$

Dabei ist $ArtId_k$ die ID von einer Art der Wissensrepräsentation und $Pfad_k$ der Pfad zu der Programmdatei für die Art der Wissensrepräsentation. Diese Information wird im Folgenden als *ArtProgrammpfad* bezeichnet. Für eine effiziente Interpretation einer Lösung ist es wichtig, dass die Lösung unabhängig von der verwendeten Art der Wissensrepräsentation ist. Dafür muss die berechnete Lösung des Programms in eine solverunabhängige Form gebracht werden. Dabei ist ein zentraler Bestandteil, dass die für die Interpretation benötigten Prädikate aus der berechneten Lösung jeweils eine eindeutige ID zugewiesen wird. Dafür legt der Benutzer Abbildungen in folgender Form fest:

$$\{RueckgabeId_1 : Name_1; Stelligkeit_1, ..., RueckgabeId_n : Name_n; Stelligkeit_n\}$$

Dabei ist $RueckgabeId_k$ die ID für ein Prädikat p der Lösung, $Name_k$ der Name von p und $Stelligkeit_k$ die Stelligkeit von p. Diese Information wird im Folgenden als *Loesung-Praedikat* bezeichnet. Die Abbildung 3.1 zeigt wie die Benutzereingabe für das Beispiel *AWMBieten* aussieht. Die *RueckgabeId* wird im weiteren Verlauf auch als Rückgabe ID bezeichnet.

```
{
  InfoAgententyp      →     {
                              FAHRZEUG_POS : Lokalisator,
                              JOB : Auktionator,
                              REGALEBENEN : Inventarmanager
                            }

  VerwendeteArt       →       AWM

  ArtProgrammpfad     →     {
                              AWM : 'AWMBieten.lp'
                            }

  LoesungPraedikat    →     {
                              BIETEN_JOB : 'bieten;2'
                            }
}
```

Listing 3.1: Die Benutzereingaben im *Konfigurations-Modul* für das Beispiel *AWMBieten*.

3.1.2 Eingabe-Modul

Ein Programm benötigt normalerweise Eingabedaten. Da es sich bei dem ZFT System um ein Multiagentensystem handelt, kann es sein, dass der Agent nicht alle benötigten Informationen besitzt und diese bei anderen Agent im System anfragen muss. Deshalb werden in diesem Modul die benötigten Eingabedaten für das Programm gesammelt und geeignet zusammengestellt.

Eingabe:
Das Modul bekommt über das *Konfigurations-Modul* die Daten, welche Informationen der Agent zum Ausführen des Programms benötigt und welche Agententypen diese Informationen bereitstellen.

Ausgabe:
Das Modul gibt die gesammelten Eingabedaten in folgender Form zurück:

$$\{InfoId_1 : [wert_{1,1}, ..., wert_{1,m}], ..., InfoId_n : [wert_{n,1}, ..., wert_{n,k}]\}$$

Dabei ist $InfoId_i$ die ID von einer Information und $wert_{i,j}$ ein Eingabedatum für diese Information.

Ablauf

Im Folgenden soll der Ablauf in dem Modul vorgestellt werden. Damit wird die Arbeitsweise dieses Moduls verdeutlicht. Die verwendeten Methoden in Abbildung 3.3 werden im Anschluss genauer betrachtet.

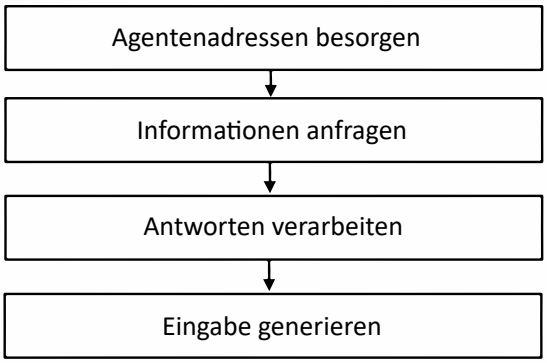


Abbildung 3.3: Der Ablauf von dem *Eingabe-Modul*.

Agentenadressen besorgen:

Jeder Agent in einem Multiagentensystem hat eine eindeutige Adresse, die für eine zuverlässige Kommunikation im System nötig ist. Da die Agenten im ZFT System sehr flexibel sind, weil sie zum Beispiel hinzugefügt oder entfernt werden können, gibt es eine zentrale Instanz zur Agentenverwaltung. Diese zentrale Instanz hat zu jeder Agentenart eine Liste, welche alle Agentenadressen für die aktiven Agenten dieser Art enthält. Diese zentrale Instanz wird im Folgenden als *Verteiler* bezeichnet. Agenten können diese Listen anfragen. Nachdem ein Agent die aktuellen Agentenadressen hat, kann er direkte Anfragen an die jeweiligen Agenten stellen, ohne mit der zentralen Instanz zu interagieren. In Abbildung 3.4 wird für das Beispiel *AWMBieten* die Besorgung der Agentenadressen visualisiert. In dem Beispiel werden die Agentenadressen vom *Auktionator, Lokalisator* und *Inventarmanager* benötigt.

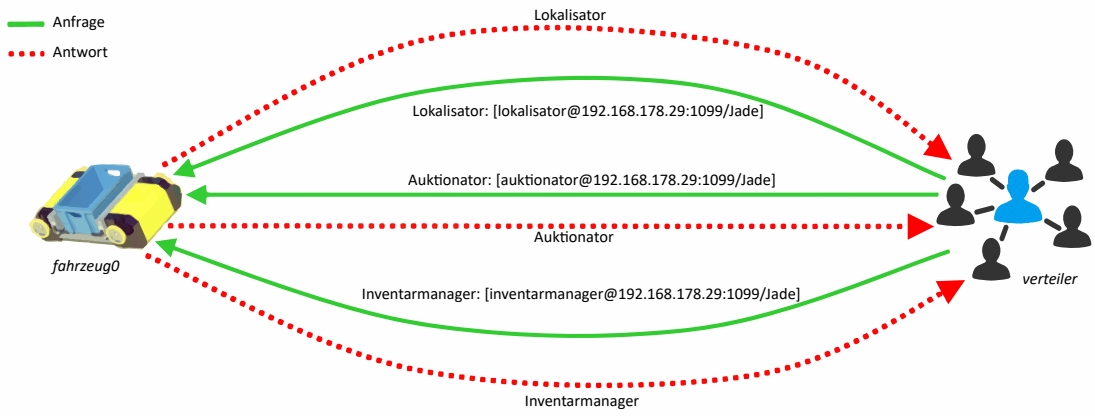


Abbildung 3.4: Die Methode *Agentenadressen besorgen* für das Beispiel *AWMBieten*.

Informationen anfragen:

Nachdem der Agent die Agentenadressen für alle benötigten Informationen hat, kann er die benötigten Informationen bei den jeweiligen Agenten anfragen. Die Anfragen werden

über das *Sende-Modul* gestellt. Jeder Information ist dabei eine eindeutige ID zugeordnet, wodurch ein geregelter Austausch von Informationen zwischen Sender und Empfänger gewährleistet wird. Anhand der Abbildung 3.5 können die Anfragen für das Beispiel *AWMBieten* nachvollzogen werden.

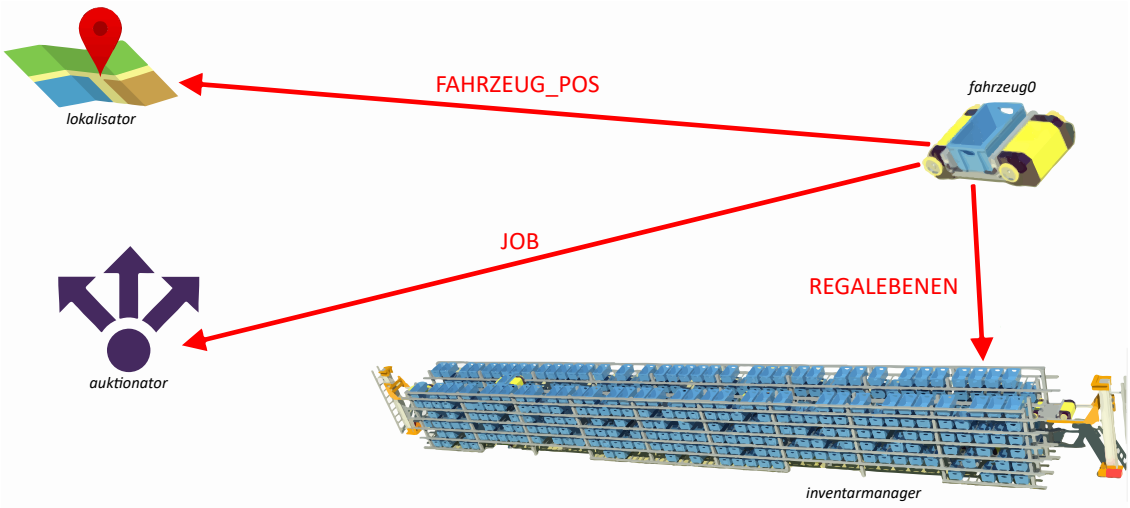


Abbildung 3.5: Die Methode *Informationen anfragen* für das Beispiel *AWMBieten*.

Antworten verarbeiten:

Das ZFT System ist ein Multiagentensystem und die Kommunikation ist asynchron. Dabei kann es vorkommen, dass Anfragen aus dem vorherigen Schritt nicht rechtzeitig ankommen. Deshalb wird in dieser Methode auf die angefragten Informationen gewartet. Wenn eine Antwort über das *Empfangs-Modul* eintrifft, dann wird diese verarbeitet. Zudem arbeitet diese Methode mit einem Timeout, um einen Fortschritt im Ablauf zu garantieren. Außerdem ist es wichtig, dass die Antworten eine fest definierte Syntax haben. Denn mit einer festen Syntax können die Antworten unabhängig von der Information verarbeitet werden. Anhand der Abbildung 3.6 kann die Idee der Methode für das Beispiel *AWMBieten* nachvollzogen werden.

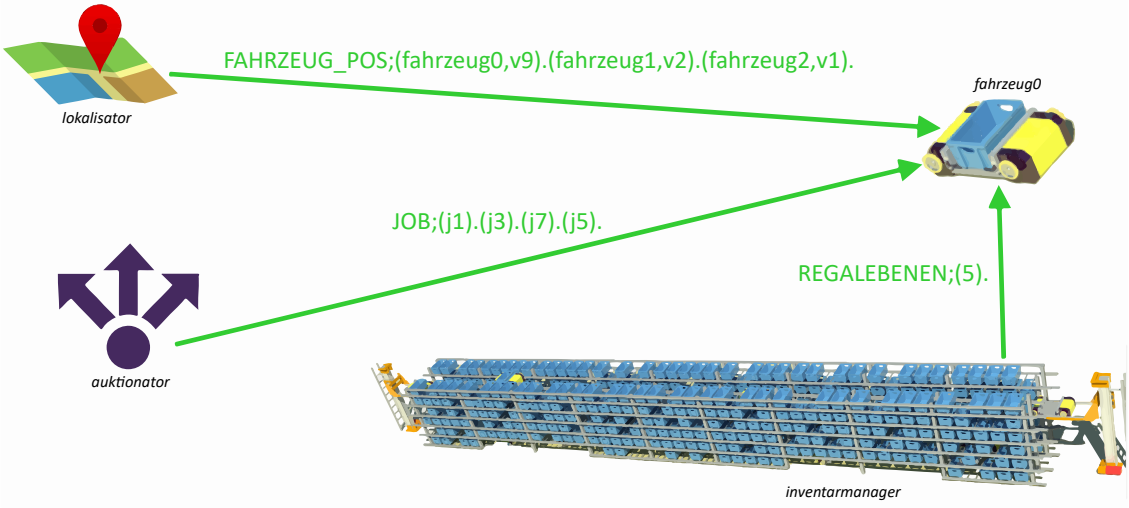


Abbildung 3.6: Die Methode *Antworten verarbeiten* für das Beispiel *AWMBieten*.

Eingabe generieren:

In dieser Methode werden die angefragten und verarbeiteten Eingabedaten für das Programm zusammengetragen und in eine fest definierte Syntax gebracht, sodass die nachfolgenden Module eine effiziente Arbeitsgrundlage haben. Dabei besteht die Struktur der Eingabedaten aus einer Menge von Abbildungen von einer Informations ID auf eine Liste, welche alle gesammelten Informationen zu dieser ID enthält. Das Listing 3.2 visualisiert beispielhaft die gesammelten Daten für das Beispiel *AWMBieten*.

```
{
  FAHRZEUG_POS : [(fahrzeug0,v9),(fahrzeug1,v2),(fahrzeug2,v1)],
  JOB : [(j1),(j3),(j7),(j5)],
  REGALEBENEN : [(5)]
}
```

Listing 3.2: Die Methode *Eingabe generieren* für das Beispiel *AWMBieten*.

3.1.3 Lösungs-Modul

Dieses Modul berechnet für die gesammelten Eingabedaten und einem ausgewählten Programm eine Lösung und wandelt diese anschließend in eine solverunabhängige Form um. Da die Methoden der Wissensrepräsentation verschiedene Syntaxen für ihre Solver benötigen, muss das Modul zunächst die gesammelten Eingabedaten aus dem vorherigen Modul in die Syntax des ausgewählten Solvers umwandeln. Dabei ist das *Lösungs-Modul* sehr variabel, sodass es für verschiedene Arten der Wissensrepräsentation verwendet werden kann. Dieses Modul muss angepasst werden, wenn eine neue Art der Wissensrepräsentation in der Architektur verwendet werden soll.

Eingabe:
Dieses Modul arbeitet mit der Ausgabe vom *Eingabe-Modul*. Zudem erhält das Modul den Pfad zu der Programmdatei und die zu verwendende Art der Wissensrepräsentation aus dem *Konfigurations-Modul*.

Ausgabe:
Die berechnete Lösung wird von dem Modul in eine solverunabhängige Form umgewandelt, welche folgende Syntax hat:

$$\{RueckgabeId_1 : [wert_{1,1}, ..., wert_{1,m}], ..., RueckgabeId_l : [wert_{l,1}, ..., wert_{l,k}]\}$$

Dabei ist $RueckgabeId_i$ die ID von einem Prädikat aus der Lösung, das von dem Benutzer für seine Interpretation definiert wurde und $wert_{i,j}$ die Parameterliste von einem Prädikat mit der ID $RueckgabeId_i$.

Ablauf

Die Abbildung 3.7 visualisiert den Ablauf des *Lösungs-Moduls*. Die im Ablauf verwendeten Methoden werden im Folgenden genauer betrachtet.

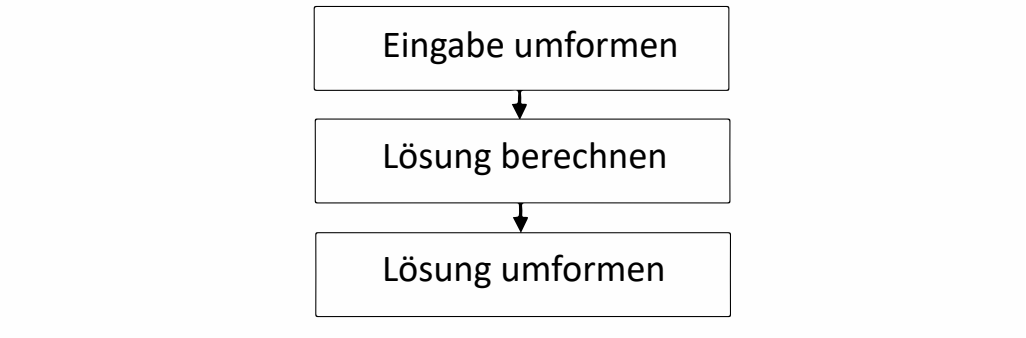


Abbildung 3.7: Der Ablauf von dem *Lösungs-Modul*.

Eingabe umformen:

Da die Architektur die Verwendung von verschiedenen Ansätzen der Wissensrepräsentation ermöglichen soll und die Solver der Ansätze verschiedene Syntaxen benutzen, müssen die gesammelten Eingabedaten in die Syntax des zu verwendenden Solvers umgeformt werden. In Listing 3.3 werden die Eingabedaten für das Beispiel *AWMBieten* (siehe Listing 3.2) in die Syntax des AWM Solvers *clingo* umgeformt. Die umgewandelten Eingabedaten aus dem Listing 3.3 werden im Folgenden als *fahrzeug0_ bieten_ base.lp* bezeichnet.

```
1 % FAHRZEUG_POS
2 fahrzeug_pos(fahrzeug0,v9). fahrzeug_pos(fahrzeug1,v2).
      fahrzeug_pos(fahrzeug1,v1).
3
4 % JOB
5 job(j1). job(j3). job(j7). job(j5).
6
7 % REGALEBENEN
8 regalebenen(5).
```

Listing 3.3: Die Methode *Eingabe umformen* für das Beispiel *AWMBieten*.

Lösung berechnen:

Die verschiedenen Ansätze der Wissensrepräsentation verwenden unterschiedliche Methoden zur Berechnung von Lösungen. Deshalb wird eine Methode benötigt, in welcher die Lösungsberechnung eines Solvers implementiert werden kann. Die Berechnungen des Solvers

sollen in dieser Methode umgesetzt werden. Die Abbildung 3.8 zeigt beispielhaft für den AWM Solver *clingo* den Ablauf der Berechnung der Lösung für das Beispiel *AWMBieten*. In dem Beispiel gehen die gesammelten Eingabedaten (siehe Listing 3.3) zusammen mit dem AWM Programm *AWMBieten.lp* in den Solver *clingo* und dieser liefert eine Ausgabe. Diese Ausgabe wird nach der Lösung des Problems gefiltert.

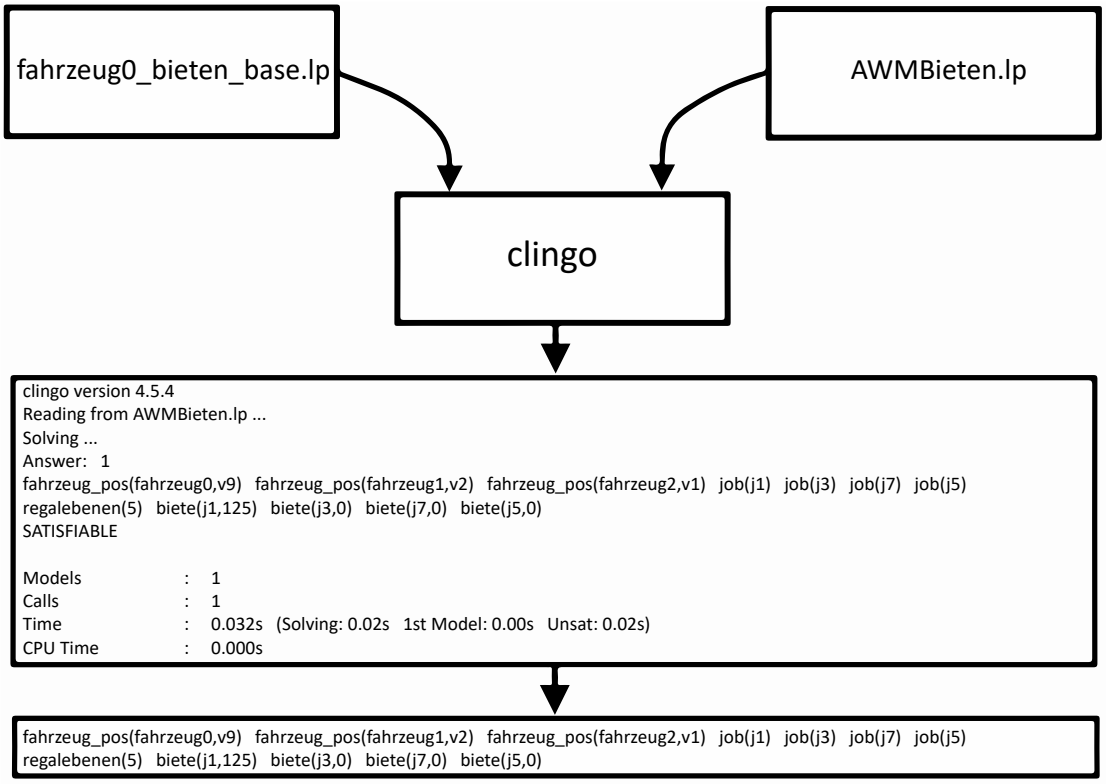


Abbildung 3.8: Die Methode *Lösung berechnen* für das Beispiel *AWMBieten*.

Lösung umformen:

In dieser Methode wird die berechnete Lösung in eine solverunabhängige Form umgeformt. Dafür wird zunächst die Lösung nach relevanten Prädikaten gefiltert. Ein Prädikat ist dabei relevant, wenn der Benutzer für dieses Prädikat eine Abbildung auf eine Rückgabe ID im *Konfigurations-Modul* definiert hat. Dabei wird die Lösung in eine fest definierte Syntax transformiert. Diese Methode ist für die weiteren Schritte der Architektur fundamental, da die Lösung nach dieser Methode unabhängig vom verwendeten Solver ist. Im Listing 3.4 kann die Umformung der Lösung für das Beispiel *AWMBieten* nachvollzogen werden.

```
1 {
2   BIETE_JOB : [(j1,125),(j3,0),(j7,0),(j5,0)]
3 }
```

Listing 3.4: Die Methode *Lösung umformen* für das Beispiel *AWMBieten*.

3.1.4 Interpretations-Modul

Diese Modul interpretiert die umgeformte Lösung des *Lösungs-Moduls*. Dafür ist es wichtig zu erwähnen, dass die Lösung des Solvers im letzten Schritt des *Lösungs-Moduls* solverunabhängig gemacht wird und damit die Art der Wissensrepräsentation, welche für die Erstellung der Lösung verwendet wurde, nicht mehr relevant ist.

Eingabe:
Das Modul erhält als Eingabe die solverunabhängige und gefilterte Lösung des *Lösungs-Moduls*, welche folgende Form hat:

$$\{RueckgabeId_1 : [wert_{1,1}, ..., wert_{1,m}], ..., RueckgabeId_n : [wert_{n,1}, ..., wert_{n,k}]\}$$

Dabei ist $RueckgabeId_i$ die ID von einem Prädikat aus der Lösung, das vom Benutzer für seine Interpretation definiert wurde und $wert_{i,j}$ die Parameterliste von einem Prädikat mit der ID $RueckgabeId_i$ aus der Lösung. Außerdem legt der Benutzer für jede $RueckgabeId$ Aktionen fest, die bei jedem Auftreten der ID in der Lösung, vom Agenten ausgeführt werden.

Ablauf

Die solverunabhängige und gefilterte Lösung des vorherigen Moduls wird elementweise durchgegangen und für jedes Element der Lösung werden die vom Benutzer vordefinierten Methoden ausgeführt. Da die Lösung bereits gefiltert wurde, enthält sie nur die Daten, die der Benutzer für seine Interpretation benötigt. Der Benutzer legt für ein Programm eine Interpretation fest, indem er für jede Rückgabe ID Aktionen des Agenten festlegt. Dabei wird auf die Aktionen im *Aktions-Modul* zurückgegriffen. Die Aktionen werden dabei aufgrund von $RueckgabeId_i$ und die Parameter der Aktion aufgrund des Wertes $wert_{i,j}$ gewählt. Nach der Ausführung aller Aktionen wurde die Lösung umgesetzt. Das Listing 3.5 visualisiert für das Beispiel *AWMBieten*, wie die Aktionen generiert werden. Der Benutzer hat dabei in seiner Interpretation festgelegt, dass bei Vorkommen der ID *BIETE_JOB* die Methode *versendeGebot(J,G)* ausgeführt wird, wobei J die ID des Jobs und G das Gebot für diesen Job ist.

```
1 BIETE_JOB(j1,125)   → fahrzeug0.versendeGebot(j1,125)
2 BIETE_JOB(j3,0)     → fahrzeug0.versendeGebot(j3,0)
3 BIETE_JOB(j7,0)     → fahrzeug0.versendeGebot(j7,0)
4 BIETE_JOB(j5,0)     → fahrzeug0.versendeGebot(j5,0)
```

Listing 3.5: Die Berechnungen von dem *Interpretations-Modul* für das Beispiel *AWMBieten*.

3.2 Arbeitsweise der entwickelten Systemarchitektur

Nach der Beschreibung der einzelnen Module, soll die Arbeitsweise der entwickelten Architektur genauer betrachtet werden. Dafür wird im Folgenden der Ablauf der Architektur vorgestellt.

Zunächst legt der Benutzer fest, welche Programme mit welcher Art der Wissensrepräsentation ausgeführt werden sollen. Dann werden diese Informationen an die beteiligten

Agenten im System weitergegeben, welche ihre Konfiguration dementsprechend setzen. Für die Ausführung eines Programms werden zwei Varianten unterschieden. Bei der ersten Variante wird das Programm ausgeführt, wenn ein bestimmtes Event, wie zum Beispiel eine Aufforderung zum Bieten oder das Verlassen einer Kommissionierstation, im System eintritt. In der zweiten Variante wird das Programm durch einen Timer ausgeführt, welcher das Programm alle x Sekunden ausführt. Dabei werden bei der Ausführung eines Programms alle benötigten Eingabedaten bei den Agenten im System mit dem *Eingabe-Modul* angefragt. Die Anfragen werden über das *Sende-Modul* gesendet. Die Agenten, welche eine Anfrage erhalten haben, antworten mit den benötigten Informationen. Diese Antwort empfängt der Agent mit dem *Empfangs-Modul* und speichert diese geeignet ab. Dann gehen die gesammelten Eingabedaten mit dem auszuführenden Programm in das *Lösungs-Modul*. In diesem Modul wird, mit der ausgewählten Art der Wissensrepräsentation des Benutzers, eine Lösung berechnet. Diese Lösung wird an das *Interpretations-Modul* weitergegeben. Dieses interpretiert die Ausgabeprädikate und führt die jeweiligen Aktionen mit dem *Aktions-Modul* aus. Nach der Ausführung aller Aktionen wurde die Lösung umgesetzt und der Durchlauf der Architektur beendet.

3.3 Implementierung der Systemarchitektur

Die entwickelte Systemarchitektur wurde in dieser Arbeit auch umgesetzt. Für die Umsetzung existiert ein Framework, welches ein ZFT System simuliert und mit Hilfe von JADE in der Bachelorarbeit [13] realisiert wurde. In diesem Kapitel wird die Implementierung der Architektur mit dem Framework beschrieben. Dafür wird im Folgenden beschrieben, wie fundamentale Aufgaben realisiert wurden. Im Anschluss daran wird die Umsetzung der einzelnen Module der Architektur genauer betrachtet. Zudem befindet sich im Abschnitt 3.5 ein UML Diagramm, welches die Implementierung der Klassen und Methoden im Framework visualisiert.

Ein zentraler Punkt für diese Architektur ist die Kommunikation der Agenten untereinander. Dabei werden für die Kommunikation und der Verarbeitung von Daten eindeutige IDs benötigt. Die benötigten IDs werden im Folgenden aufgelistet und deren Bedeutung im System kurz beschrieben.

- $KrAgentenArt$
  ID für jede Agentenart (z.B. Fahrzeuge, Auktionator)
- $KrInfoArt$
  ID für jede Information die angefragt werden kann (z.B. POSITION_FAHRZEUG, FREIER_PLATZ)

Neben den gerade eingeführten IDs ist die *JADE* Klasse *Behaviour* für die Umsetzung der Architektur im Framework fundamental. Dabei werden mit dieser Klasse die Aktionen festgelegt, die ein Agent ausführen kann. Bei der Implementierung zum Austausch von Informationen muss auf *Behaviour* Objekten zurückgegriffen werden, wie in dem UML Diagramm (siehe Abbildung 3.10) nachvollzogen werden kann. Nachdem die Grundlagen

zur Kommunikation gelegt wurden, wird im Folgenden beschrieben, wie die einzelnen Module der Architektur (siehe Abbildung 3.2) in das Framework implementiert wurden.

Implementierung der einzelnen Module

Konfigurations-Modul

Für jedes entwickelte Programm, das mit der Architektur ausgeführt werden soll, muss der Benutzer einige Informationen festlegen. Dafür muss der Benutzer für jedes Programm eine neue Klasse anlegen, welche die abstrakte Klasse *KrProgramm* erweitert. In dieser Klasse wird die Methode *config*() vorgegeben, die der Benutzer in seiner angelegten Klasse realisieren muss. In dieser Methode werden die Eingabedaten der Architektur eingegeben. Dafür wurden bereits die Datenstrukturen zur Speicherung der Eingabedaten definiert sowie initialisiert und müssen vom Benutzer nur noch geeignet gefüllt werden. Die Daten müssen vom Benutzer, wie in der Beschreibung (siehe Abschnitt 3.1.1) vorgegeben, eingegeben werden.

Eingabe-Modul

In diesem Modul sollen die benötigten Eingabedaten für das Programm gesammelt und in einer fest definierten Syntax zur Verfügung gestellt werden. Dafür wurde dieses Modul in der Beschreibung (siehe Abschnitt 3.1.2) in mehrere Methoden unterteilt. Im Folgenden wird die Umsetzung dieser Methoden beschrieben.

Agentenadressen besorgen:

Für die Besorgung der Agentenadressen muss das Framework um einige Funktionen erweitert werden. Zunächst muss eine Kommunikationsstruktur errichtet werden, mit dem die Agenten diese Informationen austauschen können. Dafür wurde mit dem Agent *Agenten* eine zentrale Instanz zur Verwaltung der Agentenadressen implementiert. Alle Agenten im System melden sich am Anfang mit der Methode *anmeldungAgenten*() bei diesem Agenten an und dieser speichert den Agenten in der passenden Liste ab. Falls ein Agent die Adressen von einer Agentenart benötigt, stellt er eine Anfrage mit der Methode $frageAgentenAn(a)$ an die zentrale Instanz und diese antwortet mit den gewünschten Adressen. Dabei ist a vom Typ *KrAgentenArt* und repräsentiert die ID der Agentenart von denen der Agent die Adressen benötigt. Für die Erstellung einer Antwort besitzt die zentrale Instanz das *Behaviour* Objekt *EmpfangeAnfragen*. Der Agent, welcher die Anfrage gestellt hat, empfängt die Antwort der zentralen Instanz mit dem *Behaviour* Objekt $EmpfangeKrInfos$. Dabei enthält die empfangene Antwort die benötigten Agentenadressen in einer fest definierten Syntax. Die Adressen werden aus der Antwort extrahiert und geeignet gespeichert.

Informationen anfragen:

Nachdem der Agent alle benötigten Agentenadressen hat, kann er die benötigten Informationen direkt bei den Agenten anfragen, welche die Information bereitstellen. Der Agent fragt die Information mit der Methode $frageInformationAn(i)$ an. Dabei ist i vom Typ $KrInfoArt$ und repräsentiert die ID der Information die angefragt werden soll. Diese Methode erstellt eine Anfrage, welche über die Sendeeinheit des Agenten direkt an die jeweiligen Agentenadressen gesendet wird. Die Agenten, an welchen die Nachricht gesendet

wurden, empfangen diese mit dem *Behaviour* Objekt $EmpfangeKrAnfragen$. In diesem *Behaviour* Objekt wird die angefragte $KrInfoArt$ aus der Nachricht extrahiert. Für diese ID werden dann die Daten generiert und in einer fest definierten Syntax an den anfragenden Agenten zurückgesendet.

Antworten verarbeiten:

Der Agent hat Anfragen für die benötigten Informationen an die jeweiligen Agenten gesendet. Dabei antwortet ein Agent mit einer Antwort m. Diese Antwort wird von dem *Behaviour* Objekt $EmpfangeKrInfos$ des anfragenden Agenten empfangen und weiterverarbeitet. Dabei werden die Informationen aus m extrahiert und geeignet gespeichert. Antworten haben die folgende Syntax:

$$KrInfoArt; item_1.item_2. \ldots item_n.$$

Wobei $item_k$ die folgende Form hat: $(Parameterliste)$. Durch die fest definierte Syntax ist eine Verarbeitung der Daten unabhängig von der Information effizient möglich. Eine weitere Aufgabe dieser Methode ist die Verwaltung der Antworten und Timer. Dabei setzt die Methode den Status einer Anfrage auf abgearbeitet, falls der Timer der Anfrage abgelaufen ist oder eine Antwort für die Anfrage erhalten wurde.

Eingabe generieren:

Wenn der Status aller Anfragen auf abgearbeitet geändert wurde, werden die gespeicherten Informationen in eine fest definierte Syntax umgewandelt und stehen dann für die weitere Verarbeitung zur Verfügung. Diese Informationen werden wie in Abschnitt 3.1.2 beschrieben abgespeichert und können vom *Lösungs-Modul* angefragt werden. Dafür steht eine Klasse zur Verarbeitung und Verwaltung dieser Informationen bereit.

Lösungs-Modul

Das *Lösungs-Modul* muss so implementiert werden, dass eine neue Art der Wissensrepräsentation mit möglichst wenig aufwand verwendet werden kann. Aus diesem Grund wurde eine abstrakte Klasse *Solver* erstellt, welche den Ablauf des Solvers mit Ein- und Ausgabedaten vorgibt. Dieser Ablauf wird durch die Methode *solve*() in der Klasse *Solver* definiert. Um eine neue Art der Wissensrepräsentation zu implementieren, muss lediglich eine neue Klasse erstellt werden, welche die abstrakte Klasse *Solver* erweitert. Dafür muss die Klasse die abstrakte Methode $eingabeUmwandeln(m)$ implementieren. Dabei repräsentiert m die Ausgabe von dem *Eingabe-Modul*, also die Eingabedaten für das Programm in einer fest definierten Syntax. Die Methode übernimmt die Umwandlung der Eingabedaten in die Syntax des Solvers. Dafür werden die Eingabedaten in einer Schleife durchlaufen und der Benutzer kann die Informationen mit Hilfe von geeigneten *Java* Operationen in die Syntax des Solvers umwandeln. Die Klasse muss zudem die Methode $berechneLoesung(e, f, p)$ implementieren. Dabei ist e ein String, welcher die umgewandelten Eingabedaten für das Programm repräsentiert. Der String f repräsentiert den Pfad zu der Programmdatei und p die Parameter für das Programm. In dieser Methode stehen dem Benutzer alle benötigten Daten zur Berechnung einer Lösung, unabhängig von der verwendeten Art der Wissensrepräsentation, zur Verfügung. Der Benutzer legt in dieser Methode fest, wie die Berechnungen für die verwendete Art der Wissensrepräsentation ablaufen soll. Dafür wurde zum Beispiel für AWM der Solver *clingo* verwendet. Diesem Solver werden die Parameter e, f

und p in einem separaten Prozess übergeben. Nach Beendigung der Berechnung oder eines Timeouts wird der Output des Solvers an den Agenten übergeben. Dafür wird der Output des Solvers in eine separate Datei geschrieben, die nach der Beendigung der Berechnung oder eines Timeouts eingelesen und bezüglich der Lösung gefiltert wird. Die Lösung wird anschließend für die Weiterverarbeitung geeignet gespeichert. Die gespeicherte Lösung l wird an die Methode $loesungUmwandeln(l)$ weitergegeben. Diese Methode muss auch vom Benutzer implementiert werden und formt die Lösung des Solvers in eine solverunabhängige Form um. Auch hier muss der Benutzer alle Prädikate der Lösung nacheinander durchgehen und mit geeigneten *Java* Operationen in die vorgegebene Syntax umwandeln. Generell gilt, dass die zu implementierenden Methoden so umgesetzt werden sollen wie in der Beschreibung des *Lösungs-Moduls* (siehe Abschnitt 3.1.3) vorgegeben.

Interpretations-Modul

Eine Interpretation wird für jedes Programm benötigt. Deshalb muss der Benutzer die abstrakte Methode $interpretiere(m)$ umsetzen, wobei m eine solverunabhängige Lösung in einer fest definierten Syntax aus dem *Lösungs-Modul* repräsentiert. Der Benutzer kann in dieser Methode die gefilterte und solverunabhängige Lösung durchgehen und geeignete Aktionen für die Lösung ausführen. Dafür werden die Prädikate aus der Lösung nacheinander durchgegangen und für jedes Prädikat Methodenaufrufe durchgeführt. Die Parameter für die Methoden werden aus der Parameterliste von dem interpretierten Prädikat extrahiert.

3.4 Verifikation der entwickelten Systemarchitektur

In diesem Abschnitt wird die umgesetzte Architektur verifiziert. Ein fundamentaler Ansatz der Verifizierung ist die dynamische Verifizierung [25, S.372]. Bei diesem Ansatz werden die einzelnen Ebenen des zu verifizierenden Systems getestet. Dabei werden zunächst Tests für die einzelnen Module der Architektur durchgeführt. Diese Art von Test wird in der Literatur als *Modultest* bezeichnet [21, S.21]. Anschließend wird das ganze System getestet, was in der Literatur als *Integrationstest* bezeichnet wird [21, S.51]. Durch einen solchen Test wird die Zusammenarbeit der einzelnen Module getestet.

3.4.1 Modultest

In diesem Abschnitt werden die einzelnen Module der Architektur separat getestet. Dafür werden lediglich die Module betrachtet, die neu in das Framework implementiert wurden. Bei den anderen Modulen der Architektur wird ein korrektes Verhalten vorausgesetzt.

Konfigurations-Modul

Dieses Modul dient zur Entgegennahme von Benutzereingaben für die Architektur und zur Speicherung dieser Daten. Da die Aufgaben des Moduls trivial sind, wird dieses Modul als verifiziert betrachtet.

Eingabe-Modul

Das *Eingabe-Modul* besorgt die Eingabedaten für das auszuführende Programm und speichert diese geeignet ab. Für die Verifizierung dieses Moduls wird ein Test durchgeführt, in dem für eine Situation in einem ZFT System überprüft wird, ob das Modul die korrekten Daten sammelt sowie korrekt speichert. Im Folgenden wird der Test ausführlich beschrieben. Zunächst muss der Benutzer die Informationen angeben, die das Modul besorgen soll. Dabei gibt er die Informationen in der folgenden Form an:

$$InformationID : AgentenID$$

Dabei ist $InformationID$ die ID der Information, die besorgt werden soll und $AgentenID$ die ID für die Art der Agenten, die diese Information bereitstellen. Im Folgenden werden die benötigten Informationen für den Testaufbau in der vorgegebenen Syntax aufgelistet:

- *POSITION_FAHRZEUG : LOKALISATOR*
- *NAECHSTER_KNOTEN_FAHRZEUG : LOKALISATOR*
- *AUFGELADENER_LADUNGSTRAEGER : FAHRZEUG*
- *KNOTEN_G : LOKALISATOR*

Dabei können mit *POSITION_FAHRZEUG* die Positionen aller Fahrzeuge im System beim *Lokalisator* Agenten angefragt werden. Die ID *NAECHSTER_KNOTEN_FAHRZEUG* gibt für jedes Fahrzeug den nächsten Knoten an, zu dem es fahren möchte. Mit *AUFGELADENER_LADUNGSTRAEGER* werden alle Fahrzeuge mit ihren aufgeladenen Ladungsträgern ausgegeben. Die ID *KNOTEN_G* gibt alle Knoten des Weggraphen aus. Die

Abbildung 3.9 visualisiert die Situation, in der das Modul die angegebenen Daten sammeln soll.

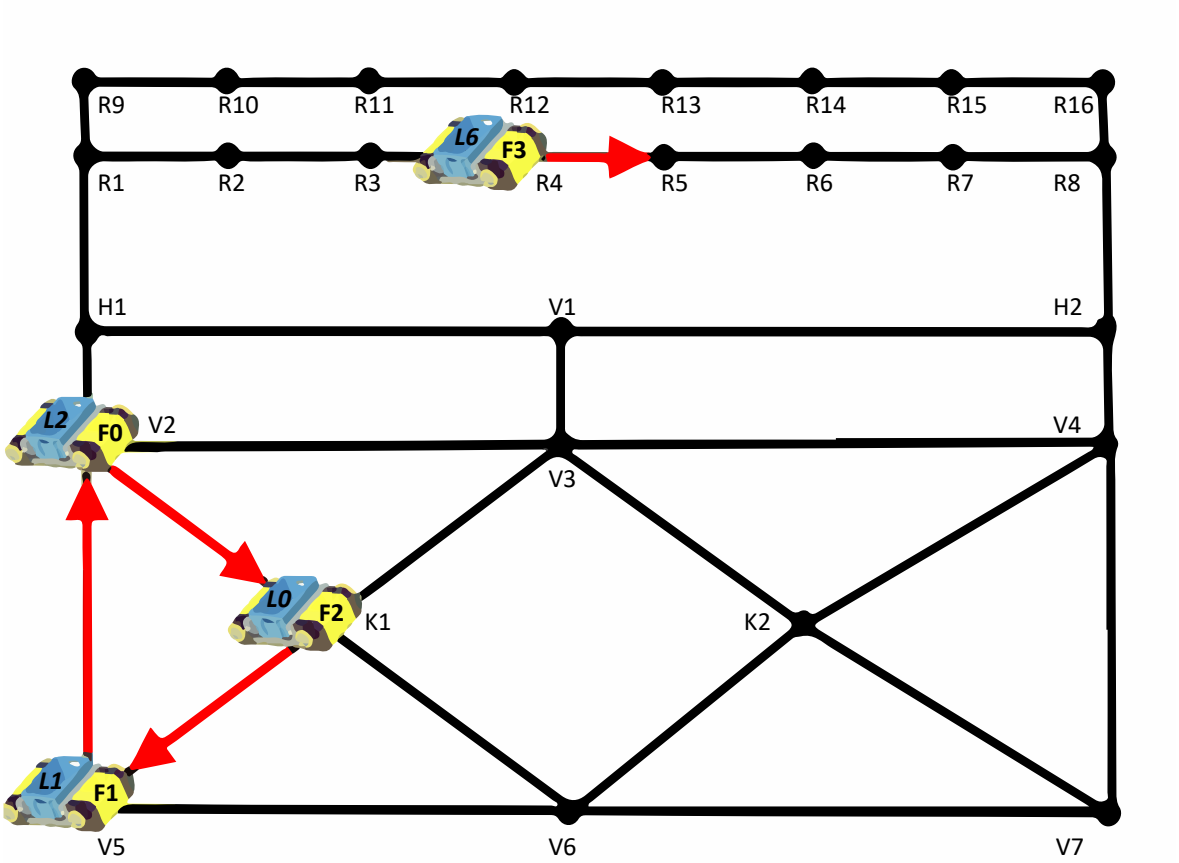


Abbildung 3.9: Die aktuelle Situation, in der das Modul die angegebenen Daten sammeln soll.

In dieser Abbildung repräsentieren die Knoten $R1 - R16$ die Lagerplätze des Regals, $H1$ und $H2$ die Lifte, $K1$ und $K2$ die Kommissionierstationen und $V1 - V7$ die Wegpunkte des Weggraphen. Ein roter Pfeil zeigt für ein Fahrzeug an, zu welchen Knoten es als nächstes fahren möchte. Das Fahrzeug F_J repräsentiert das Fahrzeug mit der ID J. Die Angabe LI auf dem Ladungsträger eines Fahrzeugs in der Abbildung repräsentiert den Ladungsträger mit der ID I. Das Listing 3.6 gibt die gesammelten Informationen so wieder, wie sie auch im Programm gespeichert werden.

```
{
  AUFGELADENER_LADUNGSTRAEGER : [(f3,l6), (f1,l1), (f2,l0), (f0,l2)],

  NAECHSTER_KNOTEN_FAHRZEUG : [ (f0,k1), (f1,v2), (f2,v5), (f3,r5) ],

  KANTEN_G : [(r1,r2), (r2,r3), (r3,r4), (r4,r5), (r5,r6), (r6,r7),
    (r7,r8), (r8,r9), (r1,r9), (r9,r10), (r10,r11), (r11,r12),
    (r12,r13), (r13,r14), (r14,r15), (r15,r16), (r16,r8), (r8,l2),
    (l1,r1), (v1,l1), (v2,l1), (l2,v1), (l2,v4), (v4,v7), (v7,v6),
    (v6,v5), (v5,v2), (k1,v5), (v3,k1), (v3,k2), (v4,k2), (v2,v3)
    , (v3,v2), (v4,v3), (v2,k1), (k1,v6), (k2,v6), (k2,v7)],

  POSITION_FAHRZEUG: [(f1,v5), (f0,v2), (f3,r4), (f2,k1)]
```

```
9 }
```

Listing 3.6: Die gesammelten Daten für die Situation aus Abbildung 3.9, wie sie auch im Programm gespeichert werden.

Die gesammelten Informationen sind vollständig und in der korrekten Syntax. Somit hat das Modul diesen Test erfolgreich bestanden. Für dieses Modul wurden weitere Tests dieser Art mit unterschiedlichen Situationen und Benutzereingaben durchgeführt, wobei alle Tests das gewünschte Ergebnis geliefert haben. Aus diesem Grund wird dieses Modul als verifiziert betrachtet.

Lösungs-Modul

Dieses Modul benötigt ein vorgegebenes Programm der Wissensrepräsentation, die gesammelten Eingabedaten für das Programm und einen Solver für die verwendete Art der Wissensrepräsentation. Mit diesen Eingaben berechnet das Modul eine Lösung für das vorgegebene Programm. Für die Verifizierung dieses Moduls wird ein Test durchgeführt, in dem das identifizierte Programm *Deadlock-Erkennung* (siehe Kapitel 4.2) mit dem AWM Solver *clingo* und den Eingabedaten aus dem Listing 3.6 eine Lösung berechnet. Der Test für das Modul ist erfolgreich, wenn die berechnete Lösung korrekt ist.

```
1 {
2   DEADLOCK : [(0)],
3   NEHME_FAHRZEUG : [(f2)]
4 }
```

Listing 3.7: Die berechnete Lösung des Moduls in einer solverunabhängigen Form für die angegebenen Eingabedaten.

Die berechnete Lösung des Solvers für die angegebene Situation wird im Listing 3.7 angegeben. Dabei wurde die Lösung in die korrekte solverunabhängige Syntax umgewandelt und liefert für die Eingabe das korrekte Ergebnis. Die Korrektheit von dem Ergebnis kann mit Hilfe der Eingabedaten und dem AWM Programm *Deadlock-Erkennung* nachvollzogen werden. Das Modul wurde mit weiteren Situationen und weiteren AWM Programmen erfolgreich getestet. Deshalb wird dieses Modul als verifiziert betrachtet.

Interpretations-Modul

Das *Interpretations-Modul* wandelt eine berechnete Lösung in Aktionen um. Dafür wird für die Verifizierung des Moduls überprüft, ob das Modul für das berechnete Ergebnis aus dem Listing 3.7 und den folgenden Angaben die korrekten Aktionen ausführt. Für das *Interpretations-Modul* muss der Benutzer eine Interpretation bereitstellen. Die Interpretation wird in folgender Form angegeben:

RückgabeID : Aktion

Die Angabe soll so verstanden werden, dass bei Vorkommen der ID *RückgabeID* in der Lösung die Aktion *Aktion* ausgeführt wird. Für den Test dieses Moduls wird die folgende Interpretation verwendet:

- *DEADLOCK : gebeDeadlockAus(b)*
- *NEHME_ FAHRZEUG : beendeFahrzeug(f)*

Dabei wird bei Vorkommen der ID *DEADLOCK* in der Lösung, die Aktion *gebeDeadlockAus(b)* ausgeführt, wobei b einen booleschen Wert repräsentiert, der aus der Lösung extrahiert werden muss. Bei jedem Vorkommen der ID *NEHME_FAHRZEUG* in der Lösung wird die Aktion *beendeFahrzeug(f)* ausgeführt, wobei f die ID von einem Fahrzeug im System ist.

```
1 gebeDeadlockAus(0)
2 beendeFahrzeug(f2)
```

Listing 3.8: Die ausgeführten Aktionen für die vorliegende Lösung aus dem Listing 3.7.

Das Modul führt die Aktionen aus dem Listing 3.8 aus, was korrekt ist. Es wurden neben diesem Test noch weitere Tests durchgeführt, in denen Lösungen in Aktionen umgewandelt wurden. Alle Tests haben die Lösungen in die korrekten Aktionen überführt, sodass das Modul als verifiziert betrachtet werden kann.

3.4.2 Integrationstest

Bei einem Integrationstest wird die Zusammenarbeit aller Module im System getestet. Dieser Test wird erst ausgeführt, wenn alle Module im System separat verifiziert wurden. Dabei wird für diesen Test auf die in der Arbeit umgesetzten Aufgabe *Deadlock-Erkennung* (siehe Kapitel 4.2) zurückgegriffen. Diese Aufgabe wird auf die Situation in Abbildung 3.9 angewendet und es wird überprüft, ob die korrekten Aktionen ausgeführt werden.

```
1 gebeDeadlockAus(0)
2 beendeFahrzeug(f2)
```

Listing 3.9: Die ausgeführten Aktionen des Systems für das Programm *Deadlock-Erkennung* und der Situation aus Abbildung 3.9.

Das System führt die Aktionen aus dem Listing 3.9 aus, was für die Aufgabe in dieser Situation korrekt ist. Es wurden zusätzlich erfolgreiche Tests mit weiteren Situationen durchgeführt. Das System wurde außerdem erfolgreich mit dem Programm *Rückkehr* (siehe Kapitel 4.1) getestet. Aus diesem Grund kann das System als verifiziert bezeichnet werden. Dabei ist zu beachten, dass mit der dynamischen Verifizierung die Korrektheit des Systems nicht bewiesen, sondern nur das Vertrauen in die Korrektheit des Systems erhöht werden kann.

3.5 UML Diagramm der implementierten Systemarchitektur

In diesem Abschnitt wird die Umsetzung der Architektur visualisiert. Die Visualisierung wird mit Hilfe eines UML Diagramms realisiert. Das UML Diagramm in Abbildung 3.10 zeigt die implementierten Klassen und Methoden sowie die Integration dieser in das bestehende Framework. Das UML Diagramm ist gerade für die weitere Entwicklung an diesem modifizierten Framework sehr hilfreich.

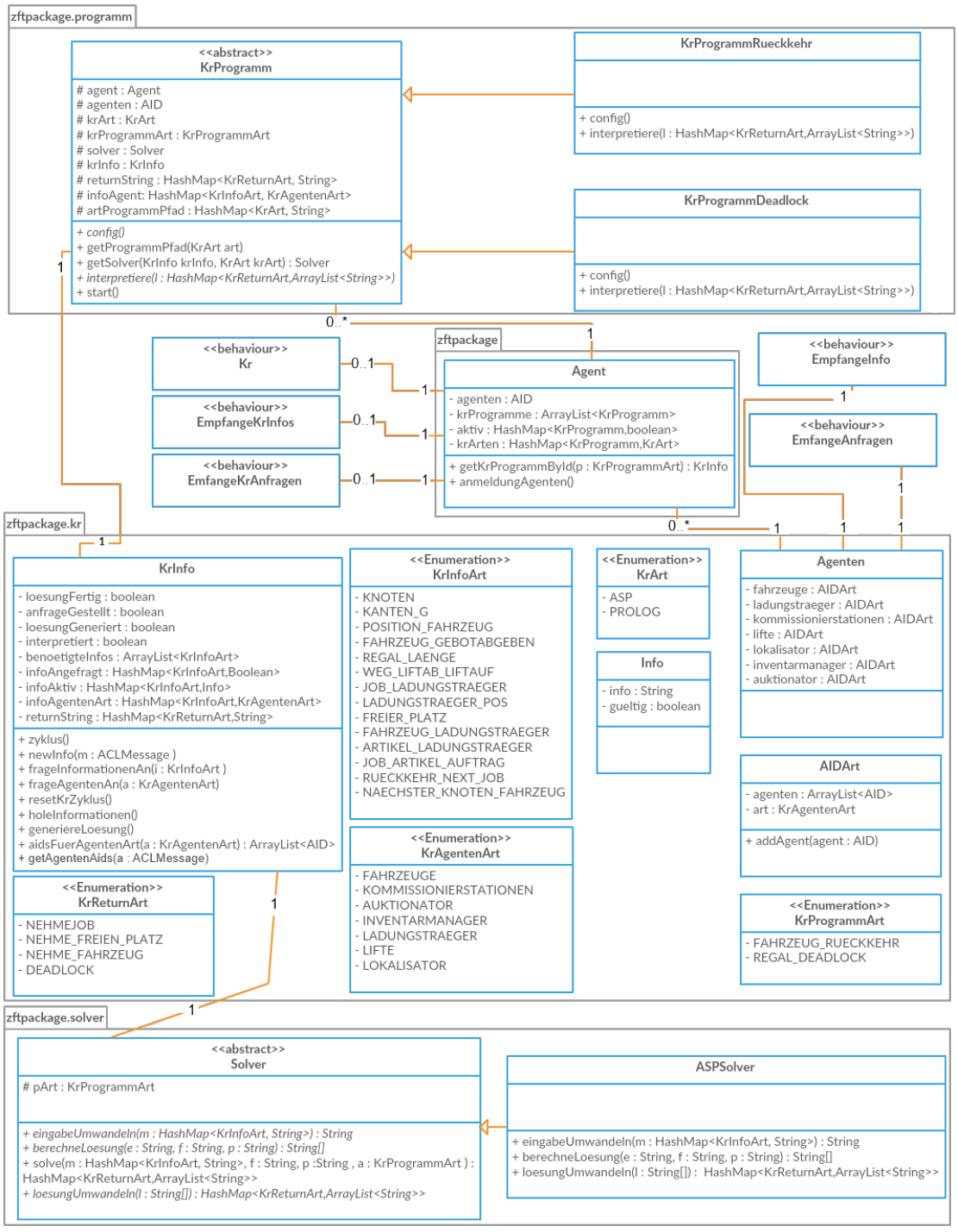


Abbildung 3.10: Das UML Diagramm für die Umsetzung.

Kapitel 4

Problemlösungen für zellulare Fördertechnik Systeme mit Hilfe der Antwortmengenprogrammierung

Wie in der Einleitung erwähnt, wurde in [24] bereits eine Planungsaufgabe mit Hilfe von AWM für das zellulare Transportsystem realisiert. Dabei wurden bei der Anwendungsaufgabe im Vergleich zur vorherigen Vorgehensweise deutlich bessere Ergebnisse erzielt, weshalb weitere Planungsaufgaben für AWM in diesem Kontext gesucht werden. Aus diesem Grund werden in diesem Kapitel weitere geeignete Aufgaben für die Antwortmengenprogrammierung im ZFT Kontext beschrieben, implementiert und evaluiert.

Bei der Vorstellung einer identifizierten Aufgabe wird zunächst die Problemstellung beschrieben und anschließend ein AWM Encoding für die Problemstellung angegeben. Alle erstellten Encodings werden in diesem Kapitel zusätzlich validiert. Ausgewählte Aufgaben werden zudem, mit Hilfe eines Frameworks, realisiert. Für eine begründete Auswahl der zu realisierenden Aufgaben wird im Abschnitt 4.6 eine Nutzwertanalyse durchgeführt. Zusätzlich werden für alle realisierten Aufgaben Evaluierungen vorgenommen.

Zunächst wird mit *Rückkehr* (siehe Abschnitt 4.1) ein Programm vorgestellt, mit dem ein Fahrzeug, welches sich auf dem Rückweg von einer Kommissionierstation zum Regal befindet, eine optimale Kombination aus freiem Stellplatz für den aufgeladenen Ladungsträger im Regal und einem neuen verfügbaren Job auswählt, sodass die Zeit für die Wege für das Auf- und Abladen minimiert wird. Anschließend wird ein Programm zur Deadlock Erkennung angeben, welches im Falle eines Deadlocks situationsabhängige Informationen für die Entfernung eines Deadlocks im System liefert. Diese Aufgabe wird als *Deadlock-Erkennung* bezeichnet und wird im Abschnitt 4.2 beschrieben. Im Abschnitt 4.3 wird das Programm *Minimiere Strafen* vorgestellt, welches eine optimale Reihenfolge für die Abarbeitung der verfügbaren Aufträge berechnet, sodass die Strafzahlungen, welche durch verspätete Fertigstellungen der Aufträge entstehen, minimiert werden. Danach wird mit dem Programm *Energieknappheit* in Abschnitt 4.4 ein Programm vorgestellt, welche alle verfügbaren Jobs aus einem Batch so auf den Fahrzeugen aufteilt, sodass die Fahrzeuge mit schwachem Energiestand nach der vollständigen Abarbeitung von dem Batch minimiert werden. Das letzte Programm aus Abschnitt 4.5 heißt *Kollisionsfreiheit* und berechnet für ein Fahrzeug mit einem Zielknoten einen minimalen Weg zu dem Zielknoten ohne Kollisionen mit anderen Fahrzeugen im System.

4.1 Rückkehr

Dieses Programm wählt für ein Fahrzeug, welches sich auf dem Rückweg von einer Kommissionierstation zum Regal befindet, eine optimale Kombination aus freiem Stellplatz für den aufgeladenen Ladungsträger im Regal und einem neuen verfügbaren Job aus, sodass die Zeit für die Wege für das Auf- und Abladen minimiert wird.

4.1.1 Beschreibung der Problemstellung

Ein Fahrzeug kehrt von einer Kommissionierstation zurück zum Regal und hat einen Ladungsträger, von seinem aktuellen Job, aufgeladen. Wenn noch Jobs verfügbar sind, dann soll das Fahrzeug einen weiteren Job erledigen. Zudem soll es seinen Ladungsträger im Regal abladen, sofern dieser nicht für den nächsten Job verwendet werden kann. Denn wenn der Ladungsträger für den nächsten Job verwendet werden kann, dann muss das Fahrzeug den Ladungsträger nicht im Regal abladen, sondern kann direkt zu der ihm zugewiesenen Kommissionierstation fahren. Dabei hat das Regal eine Menge von freien Stellplätzen, in welchen der Ladungsträger abgeladen werden kann. Dem Fahrzeug liegt zudem eine Menge von verfügbaren Jobs vor. Jedem dieser Jobs ist ein Ladungsträger, welcher die Ware für den Job enthält, zugeordnet. Das Ziel dieses Programms ist es, dass das Fahrzeug eine optimale Kombination aus Stellplatz für den Ladungsträger und einem neuen Job auswählt, sodass die Zeit für die Wege für das Auf- und Abladen minimiert wird. In Abbildung 4.1 kann die Problemstellung anhand eines Beispiels nachvollzogen werden.

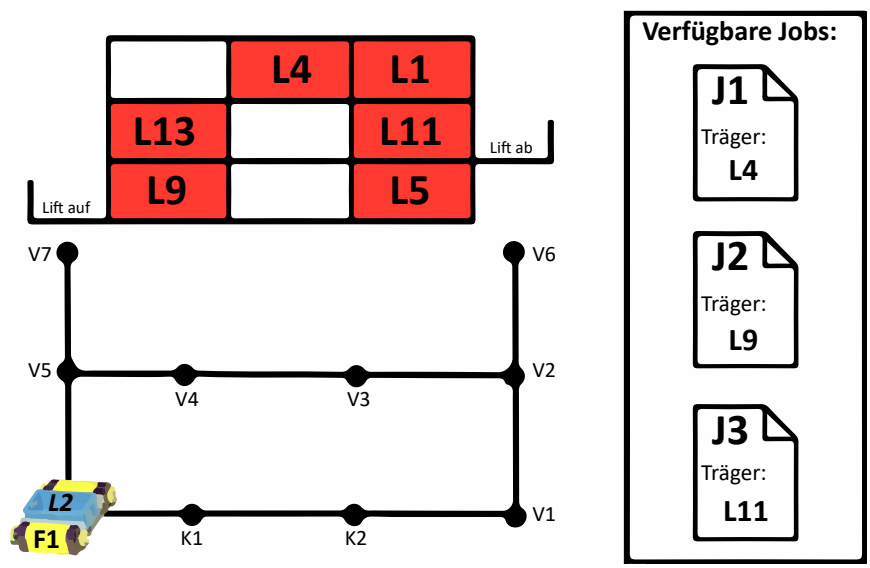


Abbildung 4.1: Ein Beispiel für die Aufgabe Rückkehr.

In dem Beispiel befindet sich das Fahrzeug $F1$, welches von seinem aktuellen Job den Ladungsträger $L2$ aufgeladen hat. Das Fahrzeug befindet sich auf den Rückweg von einer Kommissionierstation zu dem Regal. Dem Fahrzeug stehen die Jobs mit den IDs $J1$ bis $J3$ zur Verfügung. Jedem dieser Jobs ist ein Ladungsträger zugeordnet, welcher in der Abbildung als L_i bezeichnet wird. Dabei enthält ein Ladungsträger L_i den Artikel mit der ID i. Die freien Stellplätze des Regals sind in der Abbildung in weiß markiert. Das

Fahrzeug sollte optimalerweise den Stellplatz in der Etage 2 an der Position 2 in der Etage nehmen. Außerdem sollte es den Job mit der ID $J3$ nehmen. Diese Kombination aus Job und freien Stellplatz minimiert die Zeit für die Wege für das Auf- und Abladen im Regal, wie anhand der Tabelle 4.1, welche alle möglichen Kombination aus verfügbarem Job und freiem Stellplatz für das Fahrzeug mit den dazugehörigen Kosten beinhaltet, nachvollzogen werden kann. Dabei bedeutet $P_{i,j}$, dass das Fahrzeug den freien Stellplatz in der Etage i an der Position j nimmt. Die Berechnung der Zeit für die Wege für das Auf- und Abladen wird im Verlauf dieses Kapitels genauer definiert.

Job	Platz	Kosten
J1	$P_{1,2}$	19
J1	$P_{2,2}$	21
J1	$P_{3,1}$	9
J2	$P_{1,2}$	15
J2	$P_{2,2}$	17
J2	$P_{3,1}$	19
J3	$P_{1,2}$	17
J3	$P_{2,2}$	7
J3	$P_{3,1}$	21

Tabelle 4.1: Alle möglichen Kombination aus verfügbarem Job und freiem Stellplatz für das Fahrzeug aus Abbildung 4.1 mit den dazugehörigen Kosten.

4.1.2 Ablauf

Nachdem das Fahrzeug eine Kommissionierstation verlassen hat, bleibt es stehen und führt die Berechnungen für dieses Programm aus. Das Fahrzeug setzt erst dann seine fahrt fort, wenn es mit diesen Berechnungen fertig ist. Für die Berechnungen sammelt das Fahrzeug am Anfang alle Informationen, um das Programm ausführen zu können. Dabei fragt es bei dem Auktionator alle offenen Jobs an. Zudem fragt es alle freien Plätze im Regal, die Positionen der Ladungsträger für die verfügbaren Jobs und die Länge des Regals beim Lokalisator an. Für Informationen über den aufgeladenen Ladungsträger und dem Weg von dem einem Lift zu dem anderen, muss das Fahrzeug keinen anderen Agenten kontaktieren. Das Programm berechnet nun mit Hilfe der Eingaben das minimale Paar von freien Platz und neuen Job, um die Zeit für die Wege für das Auf- und Abladen zu minimieren. Wenn das Fahrzeug den aufgeladenen Ladungsträger abladen muss, dann lädt es diesen an dem berechneten freien Platz ab und lädt als nächstes den Ladungsträger des neuen Jobs auf, falls das Fahrzeug einen neuen Job bearbeiten soll. Anschließend bearbeitet das Fahrzeug den neuen Job oder fährt zum Ruheplatz, wenn kein verfügbarer Job mehr existiert.

4.1.3 Encoding

Im Folgenden wird ein Encoding für die beschriebene Problemstellung aus Abschnitt 4.1.1 vorgestellt. Dafür werden zunächst die Eingabe- und Ausgabeprädikate für das Encoding angegeben. Anschließend wird das Encoding vorgestellt und alle Regeln des Encodings ausführlich erläutert. Das Encoding ist im Listing 4.1 zu finden.

Eingabe:

- $regal\_laenge(L)$
  Die Anzahl der Stellplätze L pro Etage im Regal.
- $weg\_liftab\_liftauf(L)$
  Die Länge L des kürzesten Wegs zwischen dem Lift des Regals, welcher ein Fahrzeug nach unten und dem Lift des Regals, welcher ein Fahrzeug nach oben befördert.
- $job\_ladungstraeger(J, L)$
  Ein verfügbarer Job J und der zugehörige Ladungsträger L.
- $ladungstraeger\_pos(L, E, P)$
  Die Position des Ladungsträgers L im Regal. Mit der Regaletage E und der Position P in dieser Etage.
- $freier\_platz(E, P)$
  Ein freier Stellplatz in der Regaletage E an der Position P in dieser Etage.
- $aufgeladener\_ladungstraeger(L)$
  Der Ladungsträger L, den das Fahrzeug aktuell aufgeladen hat.

Ausgabe:

- $nehme\_freien\_platz(E, P)$
  Das Fahrzeug soll den aufgeladenen Ladungsträger in der Regaletage E an der Position P in dieser Etage abstellen.
- $nehme\_job(J)$
  Das Fahrzeug soll als nächstes den Job mit der ID J nehmen.

```
1  job(J) :- job_ladungstraeger(J,_).
2
3  job_pos(J,E,P) :- job_ladungstraeger(J,L), ladungstraeger_pos(L,E,P).
4
5  strafweg(2*L + W) :- regal_laenge(L), weg_liftab_liftauf(W).
6
7  1{nehme_job(J) : job(J)}1 :- {job(J)} > 0.
8
9  gleich :- aufgeladener_ladungstraeger(L), nehme_job(J),
       job_ladungstraeger(J,L).
10
11 1{nehme_freien_platz(E,P) : freier_platz(E,P)}1 :- not gleich.
12
13 nehme_job_pos(J,E,P) :- nehme_job(J), job_pos(J,E,P).
14
15 strafe_etage(2*E + L) :- nehme_freien_platz(E,P1), nehme_job_pos(_,E,P2),
       regal_laenge(L), P1 < P2, not gleich.
16
17 strafe_etage(2*E1 + 2*E2 + H) :- nehme_freien_platz(E1,_),
       nehme_job_pos(_,E2,_), E1 != E2, strafweg(H), not gleich.
18
19 strafe_etage(4*E + H) :- nehme_freien_platz(E,P1), nehme_job_pos(_,E,P2),
       P1 > P2, strafweg(H), not gleich.
```

```
20
21 strafe_etage(2*E + L) :- nehme_freien_platz(E,_), regal_laenge(L), not
       nehme_job(_), not gleich.
22
23 strafe_etage(-1) :- gleich.
24
25 :- nehme_freien_platz(E1,_), nehme_job_pos(_,E2,_), E1 != E2,
       freier_platz(E3,_), E3 < E1.
26
27 #minimize{E : strafe_etage(E)}.
28
29 #show nehme_freien_platz/2.
30 #show nehme_job/1.
```

Listing 4.1: Das Encoding für die Aufgabe Rückkehr.

In der ersten Zeile werden Informationen aus der Eingabe extrahiert und separat gespeichert. In den Zeilen drei und fünf werden Vorberechnungen durchgeführt. Zudem wird mit der Regel in Zeile sieben genau ein neuer Job ausgewählt, wenn es mindestens einen verfügbaren Job gibt. Die Bedingung in Zeile sieben ist wichtig, da im Normalfall irgendwann kein Job mehr zur Verfügung steht. Ist das der Fall, dann würde das Programm ohne diese Bedingung ausgeben, dass es keine Lösung gibt, jedoch wäre die korrekte Lösung, dass das Fahrzeug seinen Ladungsträger an einem freien Platz absetzt. In Zeile neun wird überprüft, ob es einen Job gibt, welcher den bereits aufgeladenen Ladungsträger benötigt. Zeile elf sorgt dafür, dass genau ein freier Stellplatz für den aufgeladenen Ladungsträger ausgewählt wird, wenn es keinen Job gibt, welcher den bereits aufgeladenen Ladungsträger benötigt. Die Regel in Zeile 13 ist eine weitere Vorberechnung, um die nachfolgenden Regeln im Programm übersichtlicher gestalten zu können. In den Zeilen 15 bis 23 werden die Kosten für die Auswahl des freien Platzes und des neuen Jobs berechnet. Die Berechnung der Kosten wird in dem Absatz *Kosten berechnen* beschrieben. Der Constraint in Zeile 25 eliminiert alle Modelle, in denen ein freier Platz ausgewählt wurde, welcher auf einer anderen Etage als der Ladungsträger des neuen Jobs liegt und es einen anderen freien Platz gibt, welcher eine niedrigere Etagenhöhe hat. Mit der Regel in Zeile 27 wird die Zeit für die Wege für das Auf- und Abladen minimiert. Die Zeilen 29 und 30 sorgen dafür, dass nur die angegebenen Prädikate ausgegeben werden.

Kosten berechnen

Die Kosten bestimmen, welches Paar von freiem Platz und verfügbarem Job ausgewählt wird. Dafür werden verschiedenste Fälle berücksichtigt, um in allen möglichen Regalmodellen ein optimales Paar von Job und Platz bestimmen zu können. Am Anfang muss der aufgeladene Ladungsträger abgeladen werden. Anschließend kann der Ladungsträger des neuen Jobs aufgeladen werden. Die Zeit für den Weg für das Auf- und Abladen beginnt, wenn das Fahrzeug mit dem Ladungsträger des alten Jobs den Lift in das Regal benutzt und endet, wenn das Fahrzeug den Lift des Regals mit dem Ladungsträger des neuen Jobs verlässt. Ein Spezialfall ist, dass es einen verfügbaren Job gibt, welcher den bereits aufgeladenen Ladungsträger benötigt. Sollte dieser Spezialfall eintreten, dann sollte das Fahrzeug auf jeden Fall einen solchen Job nehmen, da das Fahrzeug weder den Ladungsträger abnoch aufladen muss und somit die Zeit für die Wege für das Auf- und Abladen im Regal gleich null, also optimal ist. Wählt ein Fahrzeug eine Kombination aus verfügbarem Job und freiem Platz, welche in derselben Etage liegen und wo der freie Platz eine Position vor der Position des neuen Ladungsträgers in der Etage hat, dann kann das Fahrzeug den aufgeladenen Ladungsträger abladen und den neuen Ladungsträger aufladen, ohne das Regal

verlassen zu müssen. In allen anderen Fällen muss das Fahrzeug in das Regal fahren, den Ladungsträger abladen, das Regal verlassen und danach wieder in das Regal fahren, um den Ladungsträger des neuen Jobs aufzuladen. Somit ist es in den meisten Fällen optimal, wenn das Fahrzeug einen freien Platz vor dem Ladungsträger des Jobs in derselben Etage bekommen kann. Jedoch ist dies nicht in allen Regalmodellen der Fall. Es kann nämlich ein Regal geben, das nur eine sehr geringe Breite, dafür jedoch eine große Etagenhöhe hat. Dann wäre es in manchen Fällen besser, einen freien Platz und einen Job in unterschiedlichen aber niedrigen Etagen zu nehmen, als ein Paar in derselben aber hohen Etage.

Die Regeln in den Zeilen 15 bis 21 werden lediglich angewendet, wenn es keinen Job gibt, der den bereits aufgeladenen Ladungsträger benötigt. Dabei wird die Regel in Zeile 15 angewendet, wenn sich der ausgewählte freie Platz in derselben Etage wie der Ladungsträger des ausgewählten Jobs befindet und eine Position vor dem Ladungsträger hat. Dann betragen die Kosten zweimal die Etagenhöhe plus die Länge des Regals, da das Fahrzeug das Regal nicht verlassen muss und in derselben Etage ab- und aufladen kann. In den Fällen, in denen sich der freie Platz und der Ladungsträger des Jobs in einer unterschiedlichen Etage befinden oder in derselben Etage, wobei jedoch der freie Platz eine Position hinter der Position des neuen Ladungsträgers hat, werden die folgenden Kosten generiert. Das Fahrzeug muss in diesen Fällen den Ladungsträger absetzen und dann das Regal verlassen, wodurch Kosten von zweimal der Etagenhöhe des freien Platzes verursacht werden. Zudem kommen Kosten in Höhe von zweimal der Etagenhöhe des Jobs, für das Aufladen des neuen Ladungsträgers. Außerdem muss das Fahrzeug das Regal zwischenzeitlich verlassen und von dem einem Lift zum anderen zurückfahren. Zudem durchläuft das Fahrzeug das Regal im Vergleich zum ersten Fall einmal mehr. Diese Kosten werden aufsummiert und stellen die Gesamtkosten für die beiden Fälle dar. Ist kein Job mehr verfügbar, dann muss das Fahrzeug lediglich den Ladungsträger im Regal ablegen. Dabei entstehen Kosten von zweimal der Etagenhöhe des freien Platzes plus die Länge des Regals. Tritt der oben erwähnte Spezialfall auf, dass es einen verfügbaren Job gibt, welcher den bereits aufgeladenen Ladungsträger benötigt, dann soll ein solcher Job genommen werden. Dafür erhält ein solcher Spezialfall, Kosten in Höhe von -1. Damit wird ein Spezialfall bei der Minimierung auf jeden Fall genommen. Die Kostenberechnung scheint auf den ersten Blick ein wenig zu kompliziert, jedoch sollten alle Spezialfälle berücksichtigt werden, um eine optimale Auswahl für unterschiedliche Regalsysteme garantieren zu können.

Für diese Problemstellung wäre ein weiterer geeigneter Berechnungsansatz, dass nicht die Zeit für den Weg für das Auf- und Abladen, sondern der Weg selbst berechnet wird. Bei dem Weg Ansatz würden die Wege, die ein Fahrzeug für das Auf- und Abladen fahren würde, simuliert und nicht wie bei dem Zeit Ansatz approximiert. Dabei kann es bei der Berechnung des Weges in seltenen Fällen zu besseren Entscheidungen bei der Auswahl des freien Platzes und neuen Jobs kommen. Jedoch ist dieser Ansatz im Vergleich zu dem Zeit Ansatz zeitaufwendiger und die besseren Entscheidungen bringen in den meisten Fällen nur geringe Zeitvorteile. Aus diesen Gründen wurde sich für die Berechnung der Zeit und damit gegen die Berechnung der Wege entschieden.

4.1.4 Validierung

Für die Validierung des AWM Programms wird ein Äquivalenzklassentest durchgeführt. Alle möglichen Situationen werden in Äquivalenzklassen unterteilt. Dabei verhalten sich

alle Mitglieder einer Klasse auf vergleichbare Weise. Für die Validierung des Programms wird jeweils ein Repräsentant von jeder Äquivalenzklasse getestet. Mit dieser Technik werden die Testfälle systematisch ausgewählt und stark reduziert. Die Situationen werden in fünf Äquivalenzklassen unterteilt. Dabei werden alle Spezialfälle, die in dem Programm auftreten können, getestet. Diese Spezialfälle sollen im Folgenden genauer betrachtet werden.

Es gibt insgesamt drei verschiedene Spezialfälle, die eintreten können. Der erste Spezialfall ist, dass sich ein verfügbarer Job und freier Platz in derselben Etage befinden und der freie Platz eine Position vor der Position des neuen Ladungsträgers in der Etage hat. Dann muss das Fahrzeug die Etage für das Auf- und Abladen nicht verlassen. Dieser Spezialfall wird im Folgenden als S_1 bezeichnet. Ein weiterer Spezialfall ist, dass es einen verfügbaren Job gibt, welcher den bereits aufgeladenen Ladungsträger benötigt. In diesem Fall müsste das Fahrzeug gar nicht in das Regal fahren und die Ladungsträger weder auf- noch abladen. Das Fahrzeug kann dann einfach zu der ihm zugewiesenen Kommissionierstation fahren. Dieser Spezialfall wird im Folgenden als S_2 bezeichnet. Aufgrund der Berücksichtigung von verschiedenen Regalabmessungen muss ein weiterer Spezialfall berücksichtigt werden. Dieser Spezialfall wird im Folgenden als S_3 bezeichnet und tritt ein, wenn die folgende Formel erfüllt ist.

$$h_{jp} > h_j + h_p + (b + w)/2$$

Dabei sei h_{jp} die niedrigste Etagenhöhe, in der es ein Paar von verfügbarem Job und freiem Platz gibt, welches S_1 erfüllt. Des Weiteren sei h_j die niedrigste Etagenhöhe, in der sich ein verfügbarer Job befindet. Sei h_p die niedrigste Etagenhöhe, in der sich ein freier Platz befindet. Zudem sei b die Regallänge und w die Länge des kürzesten Wegs von dem einen Lift zu dem anderen. Die Spezialfälle des Programms sind in der Tabelle 4.2 übersichtlich dargestellt.

Spezialfall	Beschreibung
S_1	Ein verfügbarer Job und freier Platz befinden sich in derselben Etage und der freie Platz hat eine Position vor der Position des neuen Ladungsträgers in der Etage.
S_2	Es gibt einen verfügbaren Job, welcher den bereits aufgeladenen Ladungsträger des Fahrzeugs benötigt.
S_3	Dieser Spezialfall tritt ein, wenn die Formel: $h_{jp} > h_j + h_p + (b + w)/2$ erfüllt ist.

Tabelle 4.2: Die Spezialfälle von dem Programm übersichtlich dargestellt.

Nachdem die Spezialfälle definiert wurden, werden im Folgenden die fünf Äquivalenzklassen betrachtet. In der ersten Klasse sind alle Situationen, die mindestens einen freien Platz und keine verfügbaren Jobs enthalten. Die zweite Klasse beinhaltet alle Situationen, in denen mindestens ein verfügbarer Job und ein freier Platz enthalten sind, wobei S_1 und S_2 nicht erfüllt sein dürfen. In der dritten Klasse befinden sich alle Situationen, in denen S_2 erfüllt ist. In der vierten Klasse befinden sich alle Situationen, in denen S_2 und S_3 nicht erfüllt sind und es mindestens ein Paar von verfügbarem Job und freiem Platz gibt, das S_1 erfüllt. In der fünften Klasse befinden sich alle Situationen, in denen S_2 nicht erfüllt ist, es jedoch mindestens ein Paar von verfügbarem Job und freiem Platz gibt, das S_1 erfüllt und in denen S_3 erfüllt ist. Im Folgenden wird für jede Klasse ein Repräsentant getestet. Für

alle Repräsentanten gilt, dass die Länge des kürzesten Wegs von dem einen Lift zu dem anderen gleich der Regalbreite ist.

Test: Erste Äquivalenzklasse

In der ersten Äquivalenzklasse befinden sich alle Situationen, in denen es mindestens einen freien Platz und keine verfügbaren Jobs gibt. Das Fahrzeug sollte einen Platz mit der niedrigsten Etagenhöhe aus der Menge der freien Plätze nehmen. Mit dieser Strategie wird die Zeit für die Wege für das Auf- und Abladen minimiert. Im Folgenden wird ein Repräsentant dieser Äquivalenzklasse getestet. Die Abbildung 4.2 visualisiert den Repräsentanten.

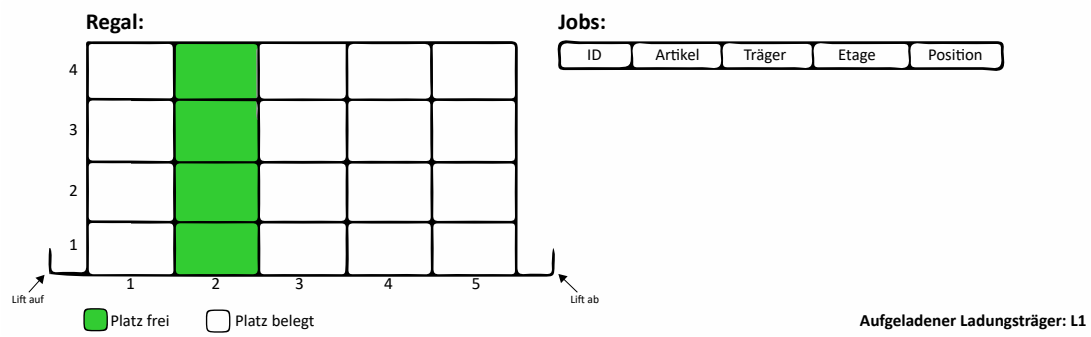


Abbildung 4.2: Ein Test für den Fall *Erste Äquivalenzklasse.*

Das AWM Programm berechnet folgende Ausgabe:

$$nehme\_freien\_platz(1, 2).$$

Die Ausgabe ist korrekt, da dies der niedrigste Platz ist, welcher dem Fahrzeug zur Verfügung steht.

Test: Zweite Äquivalenzklasse

In dieser Äquivalenzklasse befinden sich alle Situationen, die mindestens einen verfügbaren Job und einen freien Platz enthalten, wobei jedoch S_1 und S_2 nicht erfüllt sind. Somit gilt für jeden Job, dass sich die Position des Ladungsträgers von dem Job vor einem freien Platz oder in einer anderen Etage als ein freier Platz befindet. In diesen Situationen soll das Programm von den freien Plätzen, den mit der niedrigsten Etagenhöhe auswählen. Zudem soll es aus den verfügbaren Jobs, den Job mit der niedrigsten Etagenhöhe nehmen. Diese Strategie minimiert die Zeit für die Wege für das Auf- und Abladen. Die Abbildung 4.3 visualisiert den ausgewählten Repräsentanten der Äquivalenzklasse, welcher getestet werden soll.

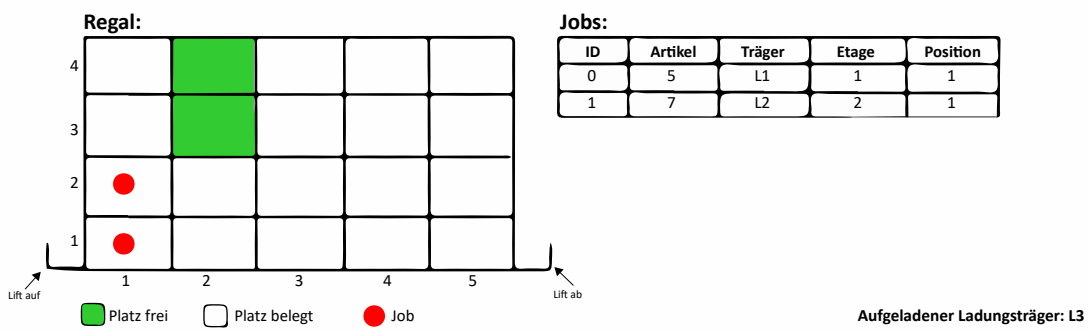


Abbildung 4.3: Ein Test für den Fall *Zweite Äquivalenzklasse*.

Das AWM Programm berechnet folgende Ausgabe:

$$nehme\_freien\_platz(3,2).\ nehme\_job(0).$$

Das Programm nimmt den Platz mit der geringsten Etagenhöhe und den Job mit der geringsten Etagenhöhe, somit ist die Ausgabe nach der Überlegung aus der Beschreibung korrekt.

Test: Dritte Äquivalenzklasse

In dieser Äquivalenzklasse befinden sich alle Situationen, in denen S_2 erfüllt ist. Somit enthält die Klasse alle Situationen, in denen es mindestens einen verfügbaren Job gibt, welcher den bereits aufgeladenen Ladungsträger benötigt. In diesen Situationen sollte das Fahrzeug einen solchen Job nehmen, denn dann muss es weder den aufgeladenen Ladungsträger im Regal abladen, noch den Ladungsträger des neuen Jobs aufladen. Das Fahrzeug kann direkt von seiner aktuellen Position aus, zu der ihm zugewiesenen Kommissionierstation fahren. Durch diese Situationen kann ein Fahrzeug im Normalfall Zeit einsparen. Die Abbildung 4.4 visualisiert den ausgewählten Repräsentanten der Äquivalenzklasse, welcher getestet werden soll. Dieser Repräsentant enthält neben einem Paar von freiem Platz und verfügbarem Job, das S_2 erfüllt einen Job in der Etage 1 an der Position 2 und einen Platz in der Etage 1 an der Position 1, da diese Paar, für alle Paare die nicht S_2 erfüllen, minimale Kosten verursacht.

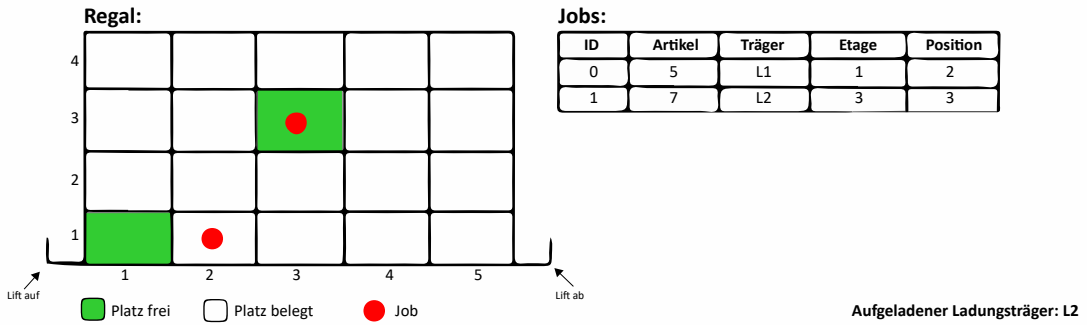


Abbildung 4.4: Ein Test für den Fall *Dritte Äquivalenzklasse*.

Das AWM Programm berechnet folgende Ausgabe:

$$nehme\_job(1).$$

Das Fahrzeug nimmt den Job mit dem Artikel 7, welcher sich in dem Ladungsträger $L2$ befindet. Genau diesen Ladungsträger hat das Fahrzeug bereits aufgeladen, somit ist die Antwort des Programms korrekt.

Test: Vierte Äquivalenzklasse

Alle Situationen, in denen S_1 erfüllt ist und die Fälle S_2 sowie S_3 nicht erfüllt sind, befinden sich in dieser Äquivalenzklasse. Somit befinden sich in diesen Situationen mindestens ein verfügbarer Job und ein freier Platz, die in derselben Etage liegen und wo der freie Platz eine Position vor der Position des neuen Ladungsträgers in der Etage hat. In den Situationen gibt es keinen Job, der den bereits aufgeladenen Ladungsträger benötigt. Zusätzlich erfüllen die Situationen den Spezialfall S_3 nicht. Sei P die Menge aller Paare von Jobs und Plätzen, die S_1 erfüllen und p_1 ein Paar aus P mit der niedrigsten Etagenhöhe. In diesen Situationen ergibt sich aus der Wahl von p_1 als Paar von Job und Platz der Vorteil, dass das Fahrzeug die Etage nicht verlassen muss, um den neuen Job aufzuladen. Die Abbildung 4.5 visualisiert den ausgewählten Repräsentanten der Äquivalenzklasse, welcher getestet werden soll. Dieser Repräsentant enthält neben einem Paar von freiem Platz und verfügbarem Job, das S_1 erfüllt einen Job in der Etage 1 an der Position 1 und einen Platz in der Etage 1 an der Position 2, da diese Paar, für alle Paare die keinen Spezialfall erfüllen, minimale Kosten verursacht.

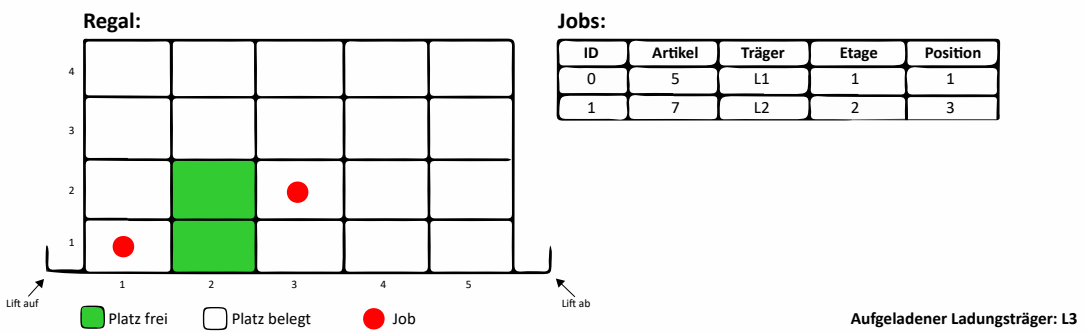


Abbildung 4.5: Ein Test für den Fall *Vierte Äquivalenzklasse*.

Das AWM Programm berechnet folgende Ausgabe:

$$nehme\_freien\_platz(2,2).\ nehme\_job(1).$$

Das Programm hat den korrekten Job und Platz ausgewählt. Der Job liegt in derselben Etage wie der Platz und hat eine Position hinter dem Platz in der Etage, wodurch der oben beschriebene Vorteil für das Fahrzeug entsteht.

Test: Fünfte Äquivalenzklasse

In dieser Äquivalenzklasse befinden sich alle Situationen, in denen die Fälle S_1 sowie S_3 erfüllt sind und S_2 nicht erfüllt ist. Damit enthalten die Situationen mindestens einen verfügbaren Job und einen freien Platz, die in derselben Etage liegen und wo der freie Platz eine Position vor der Position des neuen Ladungsträgers in der Etage hat. Sei P die Menge aller Paare von Jobs und Plätzen, die S_1 erfüllen und p_1 ein Paar aus P mit der niedrigsten Etagenhöhe. Wie bereits bei den Situationen aus der vierten Äquivalenzklasse, gibt es auch in diesen Situationen keinen Job, der den bereits aufgeladenen Ladungsträger benötigt. Im

Gegensatz zu der vierten Äquivalenzklasse erfüllen die Situationen S_3. Das bedeutet, dass es für das Fahrzeug günstiger wäre, einen Job und einen Platz aus verschiedenen Etagen zu nehmen. Obwohl das Fahrzeug das Regal dafür verlassen muss, ist die Auswahl günstiger als p_1 zu bearbeiten, was an den Regalmaßen liegt. Die Abbildung 4.6 visualisiert den ausgewählten Repräsentanten der Äquivalenzklasse, welcher getestet werden soll.

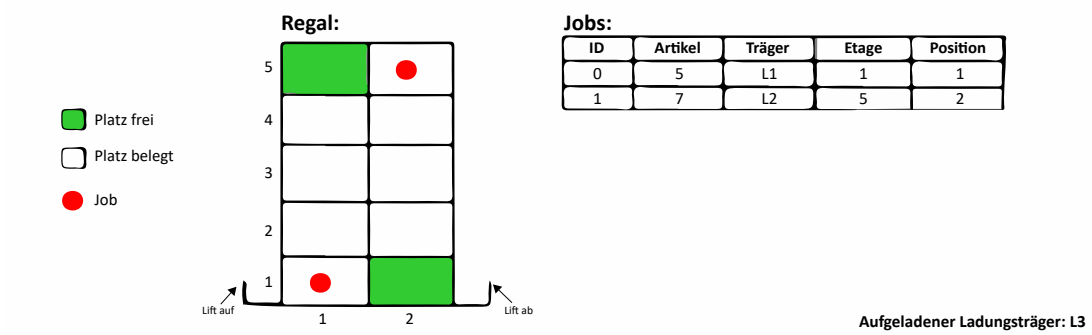


Abbildung 4.6: Ein Test für den Fall *Fünfte Äquivalenzklasse.*

Das AWM Programm berechnet folgende Ausgabe:

$$nehme\_freien\_platz(1,2).\ nehme\_job(0).$$

Bei diesem Regal ist es in dieser Situation besser einen Job und einen Platz aus einer unterschiedlichen Etage zu nehmen, als einen Job und einen Platz aus derselben Etage. Somit ist das Ergebnis für diese Situation korrekt. In der Praxis ist der beschriebene Vorteil aus dieser Äquivalenzklasse für viele Systeme kritisch zu betrachten, da die Lifte in vielen Systemen eine hohe Laufzeit haben und hier von einem optimalen Lift ausgegangen wird. Aus diesem Grund führt der theoretische Vorteil für die Auswahl der Jobs aus der unteren Etage in der Praxis zu keiner Zeitverbesserung. Jedoch könnte der beschriebene Vorteil bei einem nahezu optimalen Lift beobachtet werden. Dabei wird ein Lift als optimal bezeichnet, wenn für die zu befördernden Fahrzeuge keine bis geringe Wartezeiten an dem Lift entstehen.

Fazit

Da aus jeder Äquivalenzklasse ein Repräsentant erfolgreich getestet wurde, wurde der Äquivalenzklassentest für dieses AWM Programm erfolgreich durchgeführt und das Programm damit validiert.

4.2 Deadlock - Erkennung

Das Programm erkennt, ob es einen Deadlock im System gibt. Wenn ein Deadlock im System existiert, liefert das Programm situationsabhängige Informationen für die Entfernung eines Deadlocks im System.

4.2.1 Beschreibung der Problemstellung

„Ein Deadlock bezeichnet in der Informatik einen Zustand, bei dem eine zyklische Wartesituation zwischen mehreren Prozessen auftritt, wobei jeder beteiligte Prozess auf die Freigabe von Betriebsmitteln wartet, die ein anderer beteiligter Prozess bereits exklusiv belegt hat“ [1]. Die Fahrzeuge im ZFT System bewegen sich auf einem Weggraphen. Möchte ein Fahrzeug von einem Knoten zu einem anderen fahren, dann fragt das Fahrzeug diesen Knoten an und fährt erst zu dem Knoten, wenn dieser frei ist. Solange blockiert das Fahrzeug den Knoten, auf dem es sich befindet. So kann es passieren, dass Fahrzeuge wechselseitig aufeinander warten und dadurch ein Teil des Systems still steht. Wenn dieser Fall eintritt, dann existiert ein Deadlock in dem System. Im ZFT System kommt es, gerade wenn eine große Anzahl an Fahrzeugen eingesetzt wird, häufig zu Deadlocks. Ein Deadlock muss nach Coffman [14] notwendiger Weise folgende Eigenschaften erfüllen.

1. Die Betriebsmittel werden ausschließlich durch die Prozesse freigegeben.
2. Die Prozesse fordern Betriebsmittel an, behalten aber zugleich den Zugriff auf andere.
3. Der Zugriff auf die Betriebsmittel ist exklusiv.
4. Mindestens zwei Prozesse besitzen bezüglich der Betriebsmittel eine zirkuläre Abhängigkeit.

Dabei sind im ZFT Kontext die Knoten des Weggraphen die Betriebsmittel und die Fahrzeuge die Prozesse. Das Programm soll einen Deadlock im System erkennen, sodass dieser aufgelöst werden kann. Dafür gibt das Programm nicht nur aus, ob es einen Deadlock gibt, sondern es erkennt auch, welche Fahrzeuge an dem Deadlock beteiligt sind. Das Programm wählt aus diesen Fahrzeugen ein Fahrzeug aus, welches von seinem aktuellen Standort entfernt werden soll. Dieses Fahrzeug gibt danach seinen Knoten frei und der Deadlock ist damit aufgelöst. Die Idee der Beschreibung kann anhand des Beispiels in der Abbildung 4.7 nachvollzogen werden.

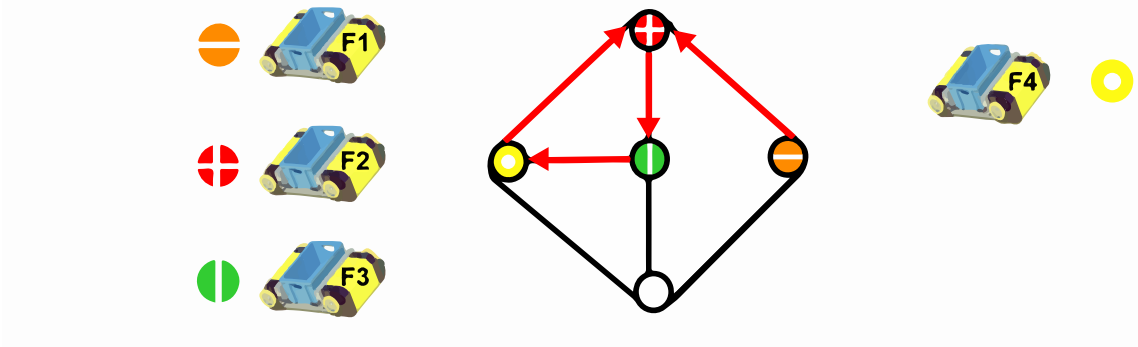


Abbildung 4.7: Ein Beispiel für die Aufgabe Deadlock-Erkennung.

In dem Beispiel befinden sich die Fahrzeuge $F1$ bis $F4$ an den farblich markierten Wegknoten im Weggraphen. Dabei bedeutet ein roter Pfeil, dass das Fahrzeug, welches sich am Pfeilanfang befindet, den Knoten an der Pfeilspitze als nächstes belegen möchte. In dem Weggraphen existiert zwischen den Fahrzeugen $F2$, $F3$ und $F4$ ein Deadlock, da diese Fahrzeuge zyklisch auf die Freigabe der von ihnen reservierten Wegknoten warten.

4.2.2 Ablauf

Der Lokalisator erhält von jedem Fahrzeug die Position, an dem es sich befindet. Zusätzlich hat der Lokalisator die Aufgabe den Fahrzeugen mitzuteilen, ob ein Knoten verfügbar ist oder nicht. Diese Aufgabe wird benötigt, da ein Fahrzeug bei jeder Fahrt von einem Knoten zu einem anderen die Verfügbarkeit des Knotens anfragen muss. Damit ist der Lokalisator der ideale Agent im ZFT System, um eine Deadlock-Erkennung durchzuführen. Der Agent führt in regelmäßigen Abständen das AWM Programm zur Deadlock-Erkennung aus und erhält damit die Information, ob es aktuell einen Deadlock im Weggraph gibt oder nicht. Falls es einen Deadlock gibt, erhält der Lokalisator von dem AWM Programm eine ID von einem Fahrzeug. Das Fahrzeug mit der ID ist an einem Deadlock im System beteiligt und soll von seinem aktuellen Platz entfernt werden, damit der Deadlock aufgelöst wird. In der Praxis ist die Aufgabe der Entfernung eines Fahrzeugs schwer zu bewältigen. Eine Möglichkeit wäre, dass eine Person die Aufgabe der Entfernung des Fahrzeugs mit geeigneten Werkzeugen übernimmt. Das Fahrzeug wird dann zu einem Ruheplatz befördert und behält seinen aktuellen Job. Das Fahrzeug fängt nun von dem Ruheplatz aus an, den Job weiter zu bearbeiten.

4.2.3 Encoding

Für die beschriebene Problemstellung aus Abschnitt 4.2.1 wird im Folgenden ein Encoding vorgestellt. Das Encoding ist im Listing 4.2 zu finden. Für die Vorstellung des Encodings werden zunächst dessen Eingabe- und Ausgabeprädikate angegeben und anschließend alle Regeln des Encodings ausführlich erläutert.

Eingabe:

- $belegt(F, K)$:
  Das Fahrzeug F belegt aktuell den Knoten K im Weggraphen.
- $naechster(F, K)$:
  Das Fahrzeug F fragt den Knoten K an, zu dem es als nächstes fahren möchte.

Ausgabe:

- $deadlock\_frei(B)$:
  Falls ein Deadlock existiert, ist $B = 0$. Ansonsten ist $B = 1$.
- $beende\_fahrzeug(F)$:
  Das Fahrzeug F soll von seiner aktuellen Position entfernt werden.

```
1  kante_d(K1,K2) :- belegt(F,K1), naechster(F,K2), K1 != K2.
2
3  erreichbar(K1,K3) :- kante_d(K1,K2), kante_d(K2,K3).
4
5  erreichbar(K1,K3) :- erreichbar(K1,K2), kante_d(K2,K3).
6
7  deadlock_frei(0) :- erreichbar(K,K).
8
9  beteiligte_knoten(K) :- erreichbar(K,K).
10
11 deadlock_frei(1) :- not deadlock_frei(0).
12
13 1{ausgewaehlter_knoten(K) : beteiligte_knoten(K)}1 :- deadlock_frei(0).
14
15 beende_fahrzeug(F) :- belegt(F,K), ausgewaehlter_knoten(K).
16
17 #show deadlock_frei/1.
18 #show beende_fahrzeug/1.
```

Listing 4.2: Das Encoding für die Aufgabe Deadlock-Erkennung.

Mit der Regel in der Zeile eins wird für jedes Fahrzeug überprüft, ob es einen Knoten anfragt, der von seinem aktuellen Knoten verschieden ist. Ist das der Fall, dann wird diese Anfrage durch eine Kante in einem Graphen zur Deadlock-Erkennung modelliert. Durch die Regel in der Zeile drei wird modelliert, dass das Fahrzeug, welches sich am Knoten $K1$ befindet, warten muss bis das Fahrzeug, welches sich am Knoten $K2$ befindet, zum Knoten $K3$ gefahren ist. Die Erreichbarkeit in dem Graphen zur Deadlock-Erkennung modelliert die Abhängigkeiten der einzelnen Anfragen der Fahrzeuge. Zeile fünf modelliert, dass ein Fahrzeug, welches sich am Knoten $K1$ befindet und davon abhängig ist, dass der Knoten $K2$ frei wird, auch noch warten muss, bis der Knoten $K3$ frei wird. Das Fahrzeug muss darauf warten, da ein Fahrzeug, welches sich am Knoten $K2$ befindet, den Knoten $K3$ anfragt. Ein Deadlock liegt vor, wenn sich ein Zyklus, welcher größer ist als eins, im Graph zur Deadlock-Erkennung befindet. Diese Überprüfung wurde in der Zeile sieben modelliert. Mit der Regel in der Zeile neun werden alle Knoten ermittelt, die an einem Deadlock im System beteiligt sind. Dabei sind alle Knoten, die auf einem Zyklus im Graphen zur Deadlock-Erkennung liegen, ein Teil des Deadlocks. Ein Knoten liegt auf einem Zyklus, wenn sich der Knoten selbst erreichen kann. Mit der Regel in der Zeile elf wird festgelegt, dass der Graph solange deadlockfrei ist, bis ein Deadlock identifiziert wurde. Falls es einen Deadlock gibt, wird aus der Menge der beteiligten Knoten genau ein Knoten ausgewählt, von dem ein Fahrzeug entfernt werden soll, sodass der Deadlock aufgelöst werden kann. Diese Auswahl wird mit der Regel in der Zeile 13 modelliert. Mit der Regel in der Zeile 15 wird für den ausgewählten Knoten das Fahrzeug bestimmt, welches sich aktuell auf diesem befindet. Dieses Fahrzeug soll von seinem Platz entfernt werden. Die Zeilen 17 und 18 sorgen dafür, dass nur die angegebenen Prädikate ausgegeben werden.

4.2.4 Validierung

Die Validierung des Programms wird in zwei Bereiche unterteilt. Zunächst soll sichergestellt werden, dass die Idee hinter dem Encoding korrekt ist. Danach soll strukturiert getestet werden, ob das Encoding korrekt arbeitet.

Idee korrekt

Im Folgenden soll bewiesen werden, dass die Deadlock-Entfernung korrekt funktioniert.

Aussage: Ein Deadlock im System wird entfernt, wenn ein Fahrzeug, welches sich auf einem Zyklus im Graphen zur Deadlock-Erkennung befindet, beendet wird.

Beweis: Angenommen der Deadlock, auf dem sich das zu beendende Fahrzeug befindet, existiert weiter nach der Beendigung von diesem Fahrzeug. Sei f das zu beendende Fahrzeug. Sei v der Knoten auf dem sich f befindet und k die ausgehende Kante von v aus dem Graphen zur Deadlock-Erkennung. Zudem sei h die Größe des Zyklus z. Dabei ist z der Zyklus im Graphen zur Deadlock-Erkennung, der das zu beendende Fahrzeug enthält. Zudem sei t der Teilgraph, welcher alle Knoten und Kanten enthält, die an dem Zyklus z beteiligt sind. Da jedes Fahrzeug maximal einen Knoten reservieren kann, hat jeder Knoten im Graphen zur Deadlock-Erkennung maximal eine ausgehende gerichtete Kante. Wird f beendet, dann wird der Knoten v freigegeben und die Kante k entfernt. Sei t' der neue Teilgraph mit $t' = t \setminus \{k\}$. Dann gibt es in t' genau h Knoten und $h-1$ gerichtete Kanten. Damit kann kein Zyklus der Größe h entstehen, da für einen Zyklus der Größe h mindestens h gerichtete Kanten existieren müssen. Aus diesem Grund ist für t' die vierte Bedingung, die für einen Deadlock nach Coffman notwendig ist, nicht erfüllt und dadurch enthält t' auch keinen Deadlock. Das ist jedoch ein Widerspruch zur Annahme.

Encoding korrekt

Zur Überprüfung des Encodings werden vier Situationen getestet. Die ausgewählten Situationen decken sowohl die Standardfälle als auch die Spezialfälle ab.

Test: Kein Deadlock

Dieser Test steht stellvertretend für alle Situationen, in denen es kein Deadlock gibt. Das Programm soll erkennen, dass es keinen Deadlock gibt und auch kein Fahrzeug ausgeben, welches beendet werden soll.

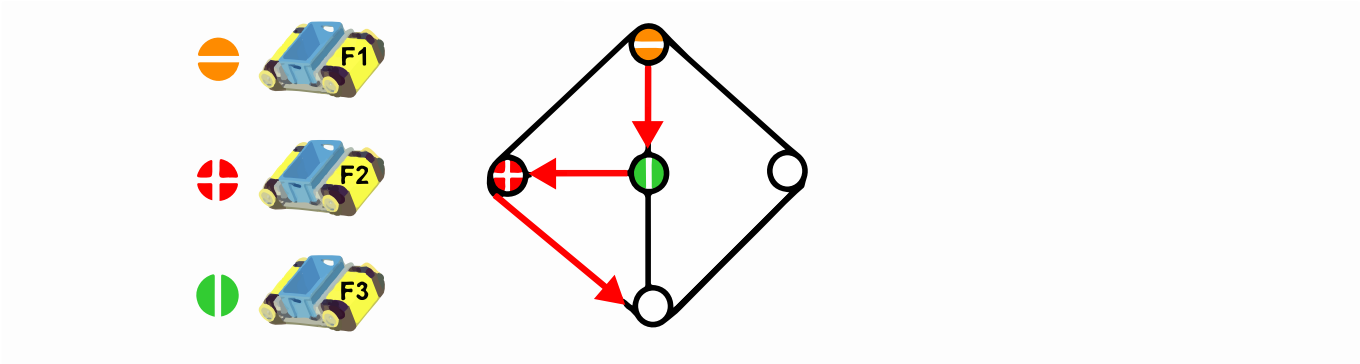


Abbildung 4.8: Ein Test für den Fall *Kein Deadlock*.

Das AWM Programm berechnet folgende Ausgabe:

$$deadlock\_frei(1).$$

Die Ausgabe des Programms ist korrekt. Die 1 in $deadlock\_frei(1)$ bedeutete, dass *deadlock_frei true* ist, also kein Deadlock in dieser Situation existiert.

Test: Ein Deadlock, welcher genau zwei Fahrzeuge enthält

In diesem Test soll überprüft werden, ob sich das Programm auch bei dem kleinstmöglichen Deadlock für dieses System korrekt verhält. Das Programm sollte ausgeben, dass es einen Deadlock in dem System gibt und sollte ein beteiligtes Fahrzeug angeben, welches von seinem aktuellen Platz entfernt werden soll.

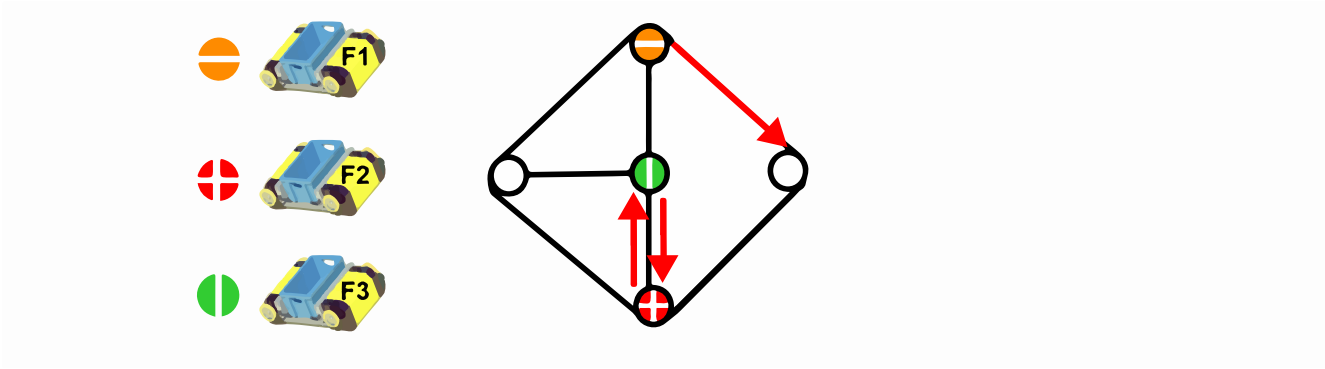


Abbildung 4.9: Ein Test für den Fall *Ein Deadlock, welcher genau zwei Fahrzeuge enthält.*

Das AWM Programm berechnet folgende Ausgabe:

$$deadlock\_frei(0).\ beende\_fahrzeug(f3).$$

Die Ausgabe des Programms ist korrekt. Die 0 in $deadlock\_frei(0)$ bedeutete, dass *deadlock_frei false* ist, also ein Deadlock in dieser Situation existiert. Laut dem Programm kann ein Deadlock aufgelöst werden, indem das Fahrzeug $f3$ beendet wird. Dies ist korrekt, da dieses Fahrzeug an einem Deadlock beteiligt ist.

Test: Ein Deadlock, welcher mindestens drei Fahrzeuge enthält

Dieser Test steht stellvertretend für alle Situationen, in denen es einen Deadlock gibt, der größer ist als zwei. Damit soll überprüft werden, ob das Programm auch größere Deadlocks erkennt. Für diesen Test wird ein Deadlock der Größe drei getestet. Falls dieser Test erfolgreich durchgeführt wurde, kann davon ausgegangen werden, dass auch größere Deadlocks erkannt werden. Das Programm sollte ausgeben, dass es einen Deadlock gibt und ein beteiligtes Fahrzeug ausgeben, welches von seinem aktuellen Platz entfernt werden soll.

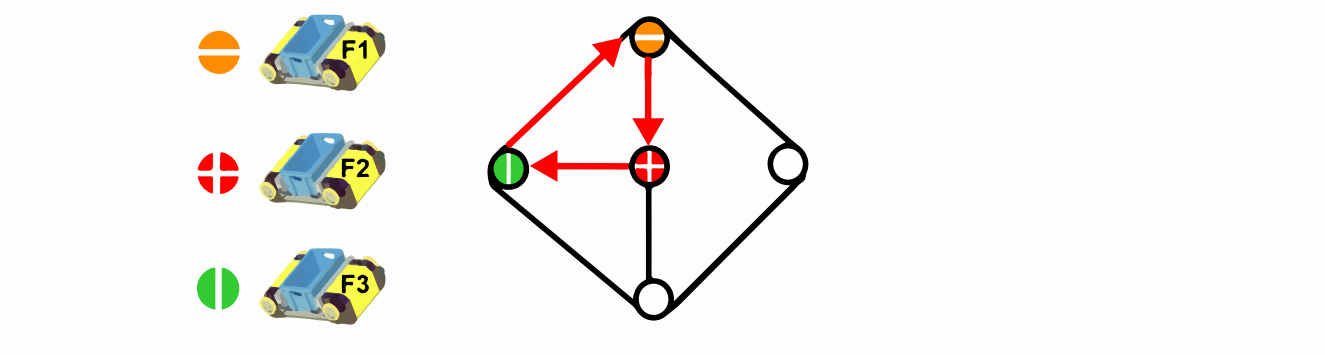


Abbildung 4.10: Ein Test für den Fall *Ein Deadlock, welcher mindestens drei Fahrzeuge enthält.*

Das AWM Programm berechnet folgende Ausgabe:

$$deadlock\_frei(0).\ beende\_fahrzeug(f3).$$

Die Ausgabe des Programms ist korrekt. Durch $deadlock\_frei(0)$ wird signalisiert, dass ein Deadlock in dieser Situation existiert. Zudem ist das Fahrzeug $f3$ an einem Deadlock beteiligt.

Test: Mehrere Deadlocks

Ein Spezialfall ist, dass es in einer Situation zu mehreren Deadlocks gleichzeitig gekommen ist. In einer solchen Situation sollte das Programm erkennen, dass es einen Deadlock gibt und versuchen einen beliebigen Deadlock aufzulösen. Dabei existiert mindestens ein Deadlock nach dem Prozess, aber das System kann weiter arbeiten und die übrigen Deadlocks bei der nächsten Deadlock Überprüfung eliminieren.

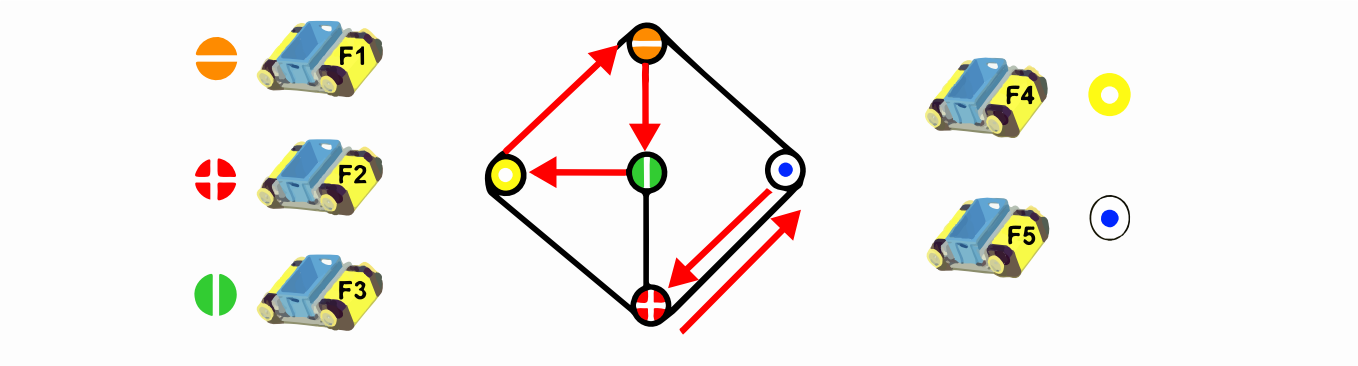


Abbildung 4.11: Ein Test für den Fall *Mehrere Deadlocks*.

Das AWM Programm berechnet folgende Ausgabe:

$$deadlock\_frei(0).\ beende\_fahrzeug(f2).$$

Die Ausgabe des Programms ist korrekt. Durch $deadlock\_frei(0)$ wird signalisiert, dass ein Deadlock in dieser Situation existiert. Außerdem wurde mit dem Fahrzeug $f2$ ein Fahrzeug ausgewählt, welches an einen der zwei Deadlocks beteiligt ist.

Fazit

Die Idee hinter der Deadlock-Entfernung wurde in dem ersten Teil der Validierung erfolgreich bewiesen. Das Encoding wurde mit Hilfe von mehreren Tests strukturiert getestet. Mit den Tests kann die Korrektheit des Encodings nicht bewiesen werden, jedoch wird dadurch das Vertrauen in das Encoding stark erhöht.

4.3 Minimiere Strafen

Dieses Programm berechnet eine optimale Reihenfolge für die Abarbeitung der verfügbaren Aufträge, sodass die Strafzahlungen, welche durch verspätete Fertigstellungen der Aufträge entstehen, minimiert werden.

4.3.1 Beschreibung der Problemstellung

Aufträge haben eine bestimmt Deadline, bis zu der sie fertiggestellt werden sollen. Wird diese Deadline überschritten, dann muss das Unternehmen pro überschrittener Zeiteinheit eine Strafzahlung leisten. Diese Strafen sind meistens je nach Unternehmen und Auftrag individuell. In manchen Situationen ist es nicht zu vermeiden, dass es zu Verspätungen und damit zu Strafzahlungen kommt. Um diese Zahlungen zu minimieren, soll dieses Programm eine Reihenfolge für die Abarbeitung der Aufträge bestimmen, bei der die Strafzahlungen minimal sind. Die Idee der Beschreibung kann anhand des Beispiels in Abbildung 4.12 nachvollzogen werden.

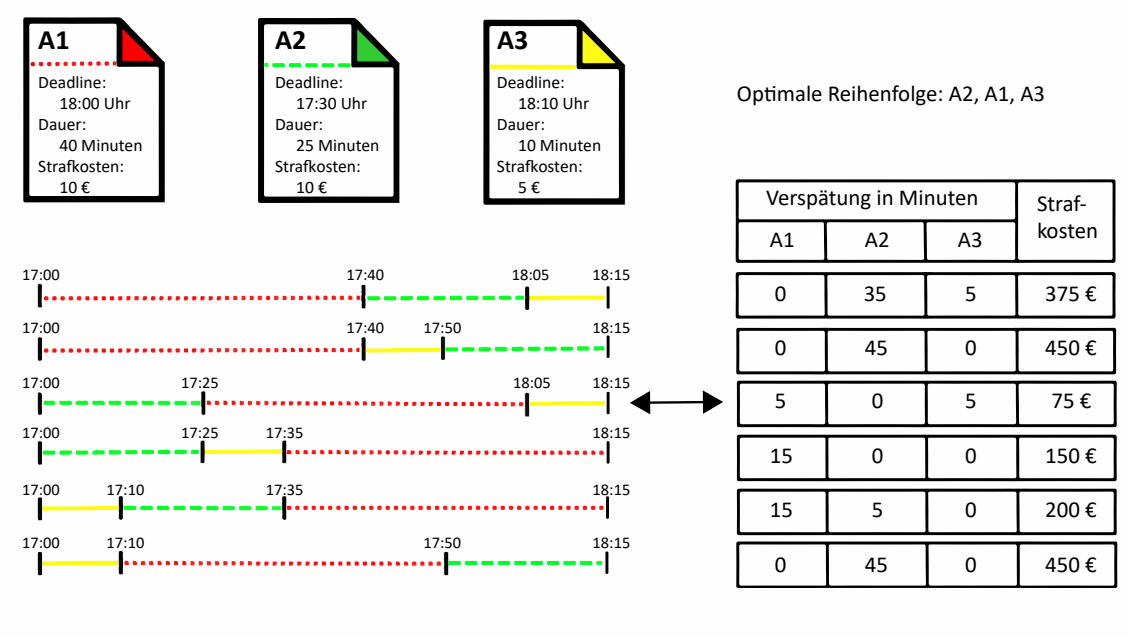


Abbildung 4.12: Ein Beispiel für die Aufgabe Minimiere Strafen.

In dem Beispiel befinden sich die Aufträge $A1$, $A2$ und $A3$. Ein Auftrag hat dabei eine Deadline, die den Zeitpunkt angibt, bis zu dem der Auftrag ohne Strafkosten fertiggestellt werden kann. Zudem besitzt jeder Auftrag eine Dauer, welche die geschätzte Zeit für die Abarbeitung des Auftrags in Minuten angibt. Außerdem sind jedem Auftrag Strafkosten zugeordnet, welche die individuellen Kosten pro verspäteter Minute angeben. In der abgebildeten Situation ist es aktuell 17 Uhr. In dieser Situation ist die Abarbeitung der Aufträge in der Reihenfolge $A2$, $A1$ und $A3$ optimal, da damit die Strafkosten minimiert werden. Die Optimalität der Reihenfolge kann anhand der Tabelle aus der Abbildung, in der alle möglichen Kombinationen mit deren Strafkosten aufgelistet sind, nachvollzogen werden.

4.3.2 Ablauf

Ein Agent, vorzugsweise der Auktionator, da er schon alle Aufträge kennt, berechnet den Zeitaufwand für die Erfüllung der Aufträge. Um den Zeitaufwand für einem Auftrag zu bestimmen, wird für jeden Job in dem Auftrag der Zeitaufwand berechnet und die Summe über alle Zeiten ergibt den Gesamtzeitaufwand für den Auftrag. Dafür muss der Zeitaufwand eines einzelnen Jobs bestimmt werden. Aus diesem Grund werden Aktionen Zeiten zugewiesen. Dabei wird auf die Daten des Multishuttle Moves zurückgegriffen: „Es bewegt sich mit einer Geschwindigkeit von 1,0 m/s auf dem Boden und mit Geschwindigkeiten von bis zu 2,0m/s im Regallager“ [19, S.3]. Dafür werden für jeden Job die Meter, die ein Fahrzeug auf dem Boden zur Erledigung des Jobs benötigt, geschätzt. Die Zeit im Regal kann in diesem System als konstant angenommen werden, denn egal welcher Ladungsträger benötigt wird, ein Fahrzeug muss immer die gesamte Regallänge durchlaufen. Ein Fahrzeug muss für einen Job insgesamt zweimal die gesamte Regallänge fahren, da angenommen wird, dass sich das zugewiesene Fahrzeug am Lift, welcher aus dem Regal führt, befindet. Ohne diese Annahme wären die Berechnungen zu zeitaufwendig und komplex, da für die konkreten Werte alle möglichen Zuweisungen von Aufträgen auf Fahrzeugen simuliert werden müssten. Die beiden Durchläufe kommen zustande, da ein Fahrzeug zunächst den Ladungsträger aufladen und schließlich noch abladen muss. Im Folgenden wird angenommen, dass sich der Lift mit einer Etage pro Sekunde bewegen kann. Somit ist der Zeitaufwand an den Liften gleich zweimal der Etagenhöhe des benötigten Ladungsträgers. Außerdem wird ein konstanter Wert von vier Sekunden bei der Benutzung der Kommissionierstation angenommen. Auch das Abladen und Aufladen verursacht jeweils eine Sekunde. Bei der Berechnung des Zeitaufwands handelt es sich lediglich um eine Schätzung des realen Zeitaufwandes, denn der Zeitverbrauch für das Warten von einem Fahrzeug wird vernachlässigt.

Zusätzlich müssen die Daten für die Deadline und die Kosten pro Verspätung in das AWM Programm eingetragen werden. Es wird angenommen, dass diese Daten dem Auktionator beim Anlegen eines neuen Auftrags mit übergeben werden. Mit Hilfe der gesammelten Daten wird dann eine Reihenfolge für die Abarbeitung der Aufträge berechnet, welche die Strafzahlungen minimiert. Die Aufträge werden dann, in der berechneten Reihenfolge, allen Fahrzeugen im System per Auktion angeboten.

4.3.3 Ideen für effizientes Encoding

In einem intuitiven Encoding wird für die Berechnung der Strafkosten ein rekursiver Ansatz verwendet, da die Berechnung am intuitivsten mit Hilfe der Rekursion modelliert werden kann. Jedoch wird aufgrund des rekursiven Ansatzes, die Berechnungszeit der Lösung schon bei einer kleinen Anzahl von Aufträgen erheblich erhöht. Dabei ist es die Groundingzeit, die bei steigender Anzahl von Aufträgen stark zunimmt. Deshalb wird ein AWM Encoding benötigt, welches die Groundingzeit gering hält.

Die Strafkosten werden in diesem Programm durch das Prädikat $kosten(Z, K, T)$ repräsentiert. Dabei ist Z die aktuelle Zeit, K die bisherigen Strafkosten und T die Anzahl der bereits abgearbeiteten Aufträge. Bei einem intuitiven Encoding würden für alle möglichen Reihenfolgen der Aufträge die Strafkosten berechnet und die Reihenfolge mit den minimalen Strafkosten ausgegeben. Dabei würde für jedes Prädikat $kosten(\_, \_, T)$ mit jedem Auftrag ein neues Prädikat $kosten(\_, \_, T + 1)$ berechnet, solange T kleiner ist als

die Anzahl der Aufträge. Denn dadurch würden für alle Reihenfolgen intuitiv die Strafkosten berechnet. Die ermittelten Strafkosten würden in dem intuitiven Ansatz am Ende minimiert. Das führt beim Grounding jedoch schon bei einer kleinen Anzahl von Aufträgen zu einer hohen Anzahl an Regel. Die Idee für ein effizienteres Encoding ist, dass nicht jeder Auftrag auf ein *kosten* Prädikat angewendet wird, sondern nur die Aufträge, die noch nicht auf dieses angewendet wurden. Dafür wird in jedem *kosten* Prädikat zusätzlich ein String gespeichert, der die Aufträge, die schon angewendet wurden, enthält. AWM liefert leider keine Funktion für Listen oder Strings. Aus diesem Grund wird Lua verwendet, um diese Funktionalität bereitzustellen. In den Grundlagen (siehe Abschnitt 2.1.4) ist eine kurze Beschreibung für Lua und der Zusammenhang mit dem AWM Solver *clingo* zu finden.

Die Idee soll im Folgenden anhand eines Beispiels verdeutlicht werden. In dem Beispiel gibt es insgesamt acht Aufträge, wobei jedem Auftrag eine ID a_i mit $i \in \{0, ..., 7\}$ zugewiesen ist. Die Abarbeitung der Aufträge in der folgenden Reihenfolge: a_0, a_3, a_4, a_6, a_7, würde in dem intuitiven Encoding das Prädikat $kosten(A, B, 6)$ und in dem effizienteren Encoding $kosten(A, B, 6, "a_0, a_3, a_4, a_6, a_7")$ erzeugen. Dabei ist A die Zeit nach der Abarbeitung der Aufträge und B die aktuellen Strafkosten bis zu diesem Zeitpunkt. Im intuitiven Encoding würden beim Grounding folgende Regeln erzeugt:

$$
\begin{aligned}
&kosten(A_0, B_0, 7) :\text{-}\ kosten(A, B, 6), auftrag(a_0).\\
&kosten(A_1, B_1, 7) :\text{-}\ kosten(A, B, 6), auftrag(a_1).\\
&kosten(A_2, B_2, 7) :\text{-}\ kosten(A, B, 6), auftrag(a_2).\\
&kosten(A_3, B_3, 7) :\text{-}\ kosten(A, B, 6), auftrag(a_3).\\
&kosten(A_4, B_4, 7) :\text{-}\ kosten(A, B, 6), auftrag(a_4).\\
&kosten(A_5, B_5, 7) :\text{-}\ kosten(A, B, 6), auftrag(a_5).\\
&kosten(A_6, B_6, 7) :\text{-}\ kosten(A, B, 6), auftrag(a_6).\\
&kosten(A_7, B_7, 7) :\text{-}\ kosten(A, B, 6), auftrag(a_7).
\end{aligned}
$$

Dabei bedeutet die Regel $kosten(A', B', T + 1) :\text{-}\ kosten(A, B, T), auftrag(a_i).$, dass der Auftrag mit der ID a_i auf das Prädikat $kosten(A, B, T)$ angewendet wird, wodurch das Prädikat $kosten(A', B', T+1)$ entsteht. Beim effizienteren Encoding werden jedoch lediglich die folgenden Regeln erzeugt:

$$
\begin{aligned}
&kosten(A_1, B_1, 7, "a_0, a_3, a_4, a_6, a_7, a_1") :\text{-}\ kosten(A, B, 6, "a_0, a_3, a_4, a_6, a_7"), auftrag(a_1).\\
&kosten(A_2, B_2, 7, "a_0, a_3, a_4, a_6, a_7, a_2") :\text{-}\ kosten(A, B, 6, "a_0, a_3, a_4, a_6, a_7"), auftrag(a_2).\\
&kosten(A_5, B_5, 7, "a_0, a_3, a_4, a_6, a_7, a_5") :\text{-}\ kosten(A, B, 6, "a_0, a_3, a_4, a_6, a_7"), auftrag(a_5).
\end{aligned}
$$

Die Regel $kosten(A', B', T + 1, S') :\text{-}\ kosten(A, B, T, S), auftrag(a_i).$ bedeutet, dass der Auftrag mit der ID a_i auf das Prädikat $kosten(A, B, T, S)$ angewendet wird, wodurch das Prädikat $kosten(A', B', T+1, S')$ entsteht. Dabei werden nur die Aufträge auf das *kosten* Prädikat angewendet, die vorher noch nicht auf dieses angewendet wurden. Mit dieser Strategie wird verhindert, dass Aufträge doppelt auf *kosten* Prädikate angewendet werden.

In einem Experiment wurden die Groundingzeiten der beiden Ansätze verglichen. Der Grounding Prozess vom intuitiven Encoding wurde für acht Aufträge nach 300 Sekunden abgebrochen. Das effizientere Encoding war bei der gleichen Eingabe bereits nach 7 Sekunden mit dem Grounding fertig. Dabei ist zu erwähnen, dass die berechneten Lösungen des Grounders für beide Ansätze in dem Beispiel nach dem Solving Prozess zu demselben Ergebnis führen.

4.3.4 Encoding

Im Folgenden wird ein Encoding für die beschriebene Problemstellung aus Abschnitt 4.3.1 vorgestellt. Dafür werden zunächst die Eingabe- und Ausgabeprädikate für das Encoding angegeben. Anschließend wird das Encoding vorgestellt und alle Regeln des Encodings ausführlich erläutert. Das Encoding ist im Listing 4.3 zu finden.

Eingabe:

- $auftrag\_fertig(A, F)$
  Der Auftrag A muss spätestens zum Zeitpunkt F fertig sein, ansonsten wird eine Strafzahlung fällig.
- $auftrag\_kosten(A, K)$
  Die Strafkosten K für den Auftrag A, die pro verspäteter Zeiteinheit fällig werden.
- $auftrag\_zeit(A, Z)$
  Die Zeit Z, die der Auftrag A benötigt, um fertiggestellt zu werden.
- $kosten(K)$
  Die aktuellen Strafkosten K bis zu diesem Zeitpunkt.

Ausgaben:

- $reihenfolge\_auftrag(S, A)$
  Der Auftrag A soll im Schritt S bearbeitet werden.
- $kosten\_ges(K)$
  Die minimalen Strafkosten K nach Abarbeitung aller Aufträge.

```
#script (lua)
function add(a, b)
    return a..","..tostring(b)
end

function notContains(p,a)
    if not string.match(p, tostring(a)) or p== '' then
        return 1
    else
        return 0
    end
end
#end.

anzahl_auftraege(N) :- N = #count{A  : auftrag_fertig(A,_)}.

step(1..N) :- anzahl_auftraege(N).

kosten(0,K,1,"") :- kosten(K).

1{reihenfolge_auftrag(T,A) : step(T)}1 :- auftrag_fertig(A,_).

:- step(T), not 1{reihenfolge_auftrag(T,A)}1.

```

```
25 kosten(X+H,Y,T+1,@add(P,A)) :- kosten(X,Y,T,P), reihenfolge_auftrag(T,A),
       @notContains(P,A) = 1, auftrag_zeit(A,H), auftrag_fertig(A,F),
       F-(X+H) >= 0.
26
27 kosten(X+H,K,T+1,@add(P,A)) :- kosten(X,Y,T,P), reihenfolge_auftrag(T,A),
       @notContains(P,A) = 1, auftrag_kosten(A,M), auftrag_zeit(A,H),
       auftrag_fertig(A,F), F-(X+H) < 0, K = Y+((X+H)-F)*M.
28
29 kosten_ges(K) :- kosten(_,K,N+1,_), anzahl_auftraege(N).
30
31 #minimize {K : kosten_ges(K)}.
32
33 #show reihenfolge_auftrag/2.
34 #show kosten_ges/1.
```

Listing 4.3: Das Encoding für die Aufgabe Minimiere Strafen.

In den Zeilen eins bis 13 befindet sich Lua Programmcode. Die Lua Funktion $add(a, b)$ nimmt einen String a und die ID von einem Auftrag b und hängt b an den String a an. Die Funktion $notContains(p, a)$ überprüft, ob die ID von einem Auftrag a in dem String p vorkommt. Ist das der Fall, dann wird 0 zurückgegeben. Ist das nicht der Fall oder ist der String leer, dann wird 1 zurückgegeben.

In Zeile 15 wird die Anzahl der Aufträge ermittelt, welche in mehreren Regeln benötigt wird. In Zeile 17 wird das Prädikat *step* definiert, um eine Reihenfolge für die Bearbeitung der Aufträge zu ermöglichen. Das erste *kosten* Prädikat wird in Zeile 19 generiert. In Zeile 21 wird jedem Auftrag genau ein Schritt zugeordnet, welcher den Zeitpunkt der Abarbeitung repräsentiert. Die Regel in Zeile 23 stellt sicher, dass bei jedem Schritt genau ein Auftrag ausgeführt wird. In der 25. Zeile werden die Strafen bis zum Zeitpunkt $T + 1$ mit der aktuellen Reihenfolge berechnet. Die Regel wird angewendet, wenn der Auftrag, der an Schritt T ausgeführt werden soll, keine Verspätung verursacht. Die Regel in der 27. Zeile wird angewendet, wenn der Auftrag, der an Schritt T ausgeführt werden soll, eine Verspätung verursacht. Dann werden die neuen Strafkosten berechnet, indem die aktuellen Strafkosten bis zum Schritt T um die Strafkosten für den aktuellen Auftrag erhöht werden. Die Verspätung des aktuellen Auftrags wird dadurch bestimmt, dass die aktuelle Zeit plus die Zeit den der Auftrag für die Bearbeitung benötigt minus die Deadline des Auftrags gerechnet wird. Dieser Wert repräsentiert nun die verspätete Zeit des Auftrags und wird mit den Strafkosten pro verspäteter Zeiteinheit multipliziert. Dies ergibt insgesamt die Strafkosten für diesen Auftrag. Wie bereits in der Idee beschrieben, wird mit Hilfe der Lua Funktion $add(a, b)$ eine Zeichenkette für jedes *kosten* Prädikat erstellt, welche die bereits angewendeten Aufträge beinhaltet. Außerdem werden die Regeln um die Bedingung $@notContains(P, A) = 1$ erweitert. Diese Bedingung sorgt dafür, dass kein Auftrag zweimal auf ein *kosten* Prädikat angewendet wird, indem die Lua Funktion $notContains(p, a)$ überprüft, ob die ID von einem Auftrag a in dem String p vorkommt. In Zeile 29 werden die gesamten Strafkosten für die Reihenfolge der Aufträge separiert gespeichert. Die Gesamtkosten werden in Zeile 31 minimiert. Die Zeilen 33 und 34 beschränken die Ausgabe auf die angegebenen Prädikate.

4.3.5 Validierung

Im Folgenden soll das Encoding validiert werden. Dafür wird am Anfang bewiesen, dass die Idee hinter dem Encoding korrekt ist und anschließend wird das Encoding strukturiert getestet. In den Tests werden sowohl Standard- als auch Spezialfälle getestet.

Idee korrekt

Im Folgenden soll bewiesen werden, dass die Berechnungen der Strafkosten und der Bearbeitungszeiten korrekt sind. Dafür wird ein Beweis per Induktion über die Abarbeitungsschritte T für die Reihenfolge der Aufträge geführt.

Aussage: Nach jedem Auftrag, der im Schritt T abgearbeitet wird, sind die Gesamtstrafkosten und die Bearbeitungszeiten im Schritt $T+1$ korrekt.

Beweis: Sei $kosten(Z_I, K_I, I)$ ein Prädikat, welches die Zeit Z_I und die Strafkosten K_I nach I abgearbeiteten Aufträgen repräsentiert.

Basisfall $T = 0$: Nach der Initialisierung existiert das Prädikat $kosten(0,0,0)$. Dieses Prädikat ist korrekt, da im Schritt 0 noch kein Auftrag betrachtet wurde und es damit noch keine Strafkosten geben kann. Außerdem wurde auch noch keine Zeit für die Bearbeitung der Aufträge aufgewendet.

Induktionsschritt $T \rightarrow T+1$: Nach der Induktionsvoraussetzung ist $kosten(Z_T, K_T, T)$ korrekt. Angenommen es gibt einen Auftrag A, welcher im Schritt T bearbeitet werden soll, dann müssen zwei Fälle für diesen Auftrag unterschieden werden. Der erste Fall ist, dass der Auftrag eine Verspätung hat und der zweite Fall ist, dass der Auftrag keine Verspätung hat. Ob der Auftrag eine Verspätung hat, kann überprüft werden, indem die folgende Ungleichung ausgewertet wird. Dabei sei Z_A die Zeit, die der Auftrag A benötigt, K_A die Kosten pro verspäteter Zeiteinheit für den Auftrag A und F_A die Deadline des Auftrags A.

$$F_A - (Z_T + Z_A) < 0$$

Es gibt genau dann eine Verspätung, wenn die Ungleichung erfüllt ist.

1.Fall (Auftrag hat keine Verspätung):
In diesem Fall werden die neuen Kosten mit der folgender Regel berechnet:

$$kosten(Z_T + Z_A, K_T, T+1) :\text{-}\ kosten(Z_T, K_T, T), Z_A, F_A.$$

Das neue *kosten* Prädikat ist korrekt, da die Gesamtzeit um die Bearbeitungszeit des Auftrags in diesem Schritt erhöht wird und nach der Induktionsvoraussetzung die Gesamtzeit bis zum Schritt T korrekt war. Zudem sind die Kosten gleich geblieben, was korrekt ist, da der Auftrag keine Verspätung hat und somit auch keine Strafzahlung für den Auftrag fällig wird.

2.Fall (Auftrag hat Verspätung):
In diesem Fall wird das neue *kosten* Prädikat mit der folgenden Regel berechnet:

$$kosten(Z_T + Z_A, K_T + ((Z_T + Z_A) - F_T) * K_A, T+1) :\text{-}\ kosten(Z_T, K_T, T), Z_A, F_A, K_A.$$

Das neue *kosten* Prädikat ist korrekt, da die Gesamtzeit um die Bearbeitungszeit des Auftrags erhöht wird und die Gesamtstrafkosten nach T bearbeiteten Aufträgen um die Verspätung des Auftrags mal die Strafkosten pro verspäteter Zeiteinheit des Auftrags erhöht werden. Die Berechung der Verspätung ist dabei korrekt, da die aktuelle Zeit mit der Bearbeitungszeit addiert wird, was zu der Zeit nach der Bearbeitung des Auftrags führt und anschließend noch die Deadline des Auftrags von dieser Zeit abgezogen wird.

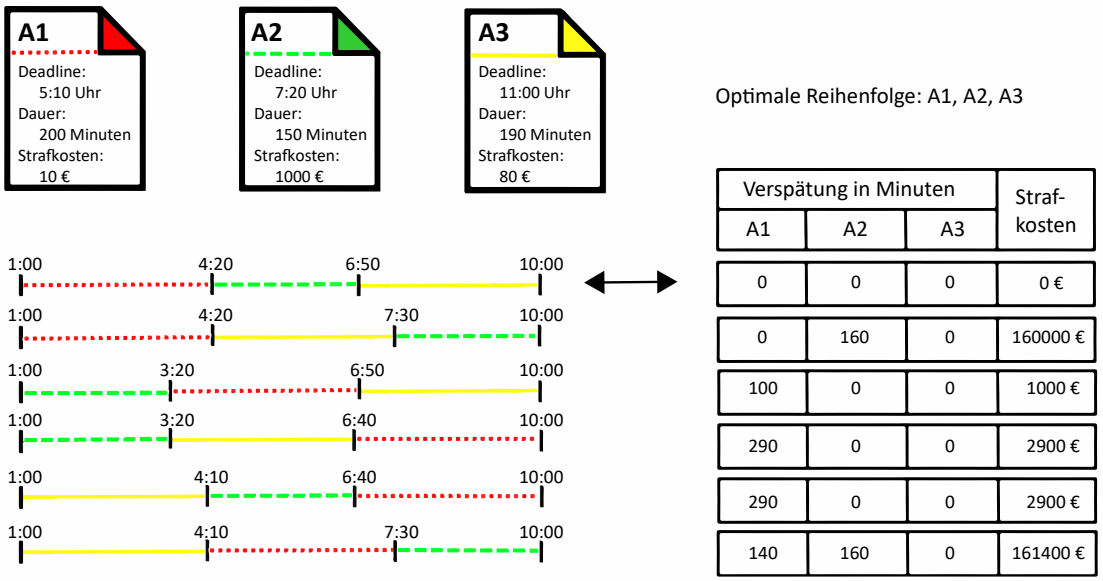
Da in allen Fälle das *kosten* Prädikat nach $T + 1$ Aufträgen korrekt ist, wurde die Korrektheit der Berechnung der Strafkosten und der Bearbeitungszeiten per Induktion bewiesen.

Encoding korrekt

Im Folgenden soll mit Hilfe von ausgewählten Testfällen das Encoding getestet werden. Dabei decken die Tests sowohl Standardfälle als aus Ausnahmefälle ab. In den Beispielen befindet sich jeweils eine Tabelle, in welcher alle möglichen Reihenfolgen der Aufträge mit ihren fälligen Strafkosten aufgelistet sind. Mit dieser Tabelle kann die Korrektheit der Ausgabe des AWM Programms nachvollzogen werden.

1. Fall: Alle Aufträge ohne Verspätung

Der Test des Repräsentanten aus der Abbildung 4.13 steht stellvertretend für alle Situationen, in denen alle Aufträge ohne Verspätung fertig gestellt werden können.



Verspätung in Minuten			Straf-kosten
A1	A2	A3	
0	0	0	0 €
0	160	0	160000 €
100	0	0	1000 €
290	0	0	2900 €
290	0	0	2900 €
140	160	0	161400 €

Abbildung 4.13: Ein Test für den Fall *Alle Aufträge ohne Verspätung.*

Das AWM Programm berechnet folgende Ausgabe:

$$reihenfolge\_auftrag(1, a1).\ reihenfolge\_auftrag(2, a2).\ reihenfolge\_auftrag(3, a3).$$
$$kosten\_ges(0).$$

Die Ausgabe ist korrekt, wie an der Tabelle, in der alle möglichen Kombinationen mit deren Strafkosten aufgelistet sind, nachvollzogen werden kann.

2. Fall: Kein Auftrag ohne Verspätung

Mit dem zweiten Fall werden alle Situationen abgedeckt, in denen bei der Reihenfolge für die minimalen Strafkosten kein Auftrag ohne Verspätung fertiggestellt werden kann. Der Repräsentant in Abbildung 4.14 wird stellvertretend getestet.

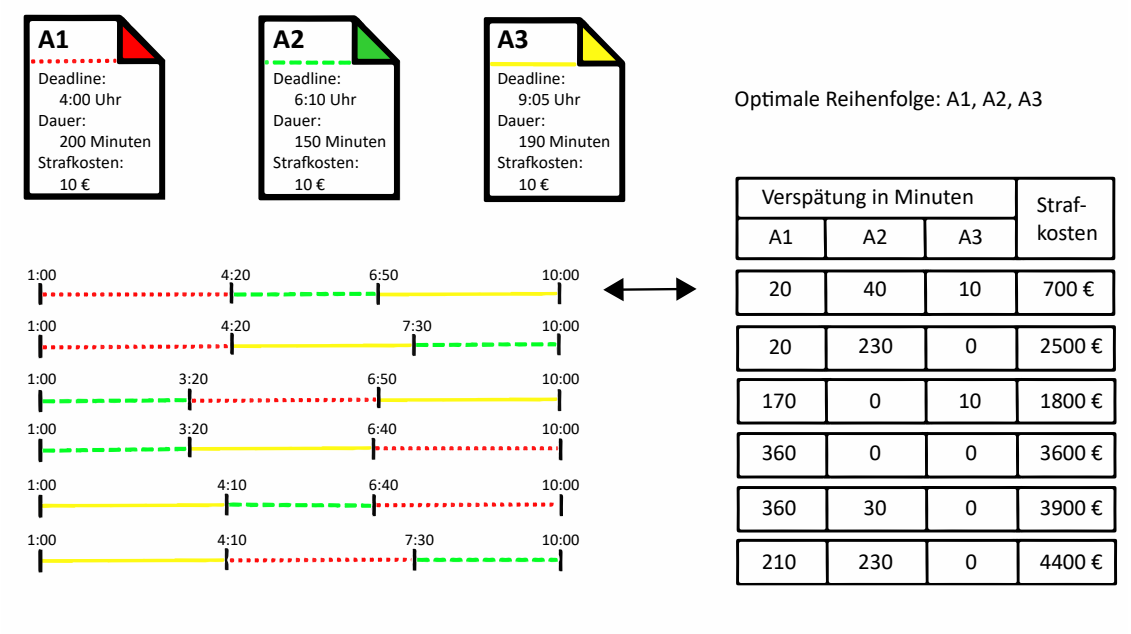


Abbildung 4.14: Ein Test für den Fall *Kein Auftrag ohne Verspätung.*

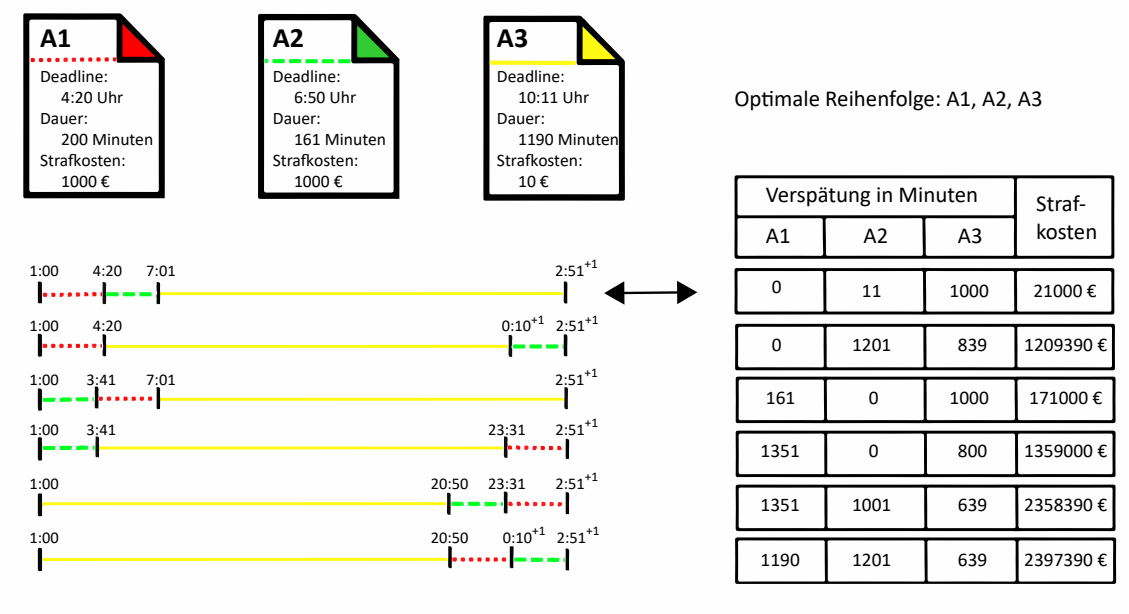
Das AWM Programm berechnet folgende Ausgabe:

$$reihenfolge\_auftrag(1, a1).\ reihenfolge\_auftrag(2, a2).\ reihenfolge\_auftrag(3, a3).$$
$$kosten\_ges(700).$$

Die Ausgabe ist korrekt, wie an der Tabelle, in der alle möglichen Kombinationen mit deren Strafkosten aufgelistet sind, nachvollzogen werden kann.

3. Fall: Große Verspätung mit kleinen Strafkosten vs. kleine Verspätung mit großen Strafkosten

Dieser Test soll einen Spezialfall überprüfen. Dabei gibt es einen Job mit einer großen Verspätung und mit kleinen Strafkosten sowie einen Job mit einer kleinen Verspätung aber mit großen Strafkosten. Hierbei verursacht der Job mit der großen Verspätung insgesamt geringere Strafkosten als der Job mit der kleinen Verspätung, sodass der Job mit der kleinen Verspätung vorher abgearbeitet werden sollte. Der Repräsentant in Abbildung 4.15 wird stellvertretend getestet.



Verspätung in Minuten			Straf-kosten
A1	A2	A3	
0	11	1000	21000 €
0	1201	839	1209390 €
161	0	1000	171000 €
1351	0	800	1359000 €
1351	1001	639	2358390 €
1190	1201	639	2397390 €

Abbildung 4.15: Ein Test für den Fall *Große Verspätung mit kleinen Strafkosten vs. kleine Verspätung mit großen Strafkosten.*

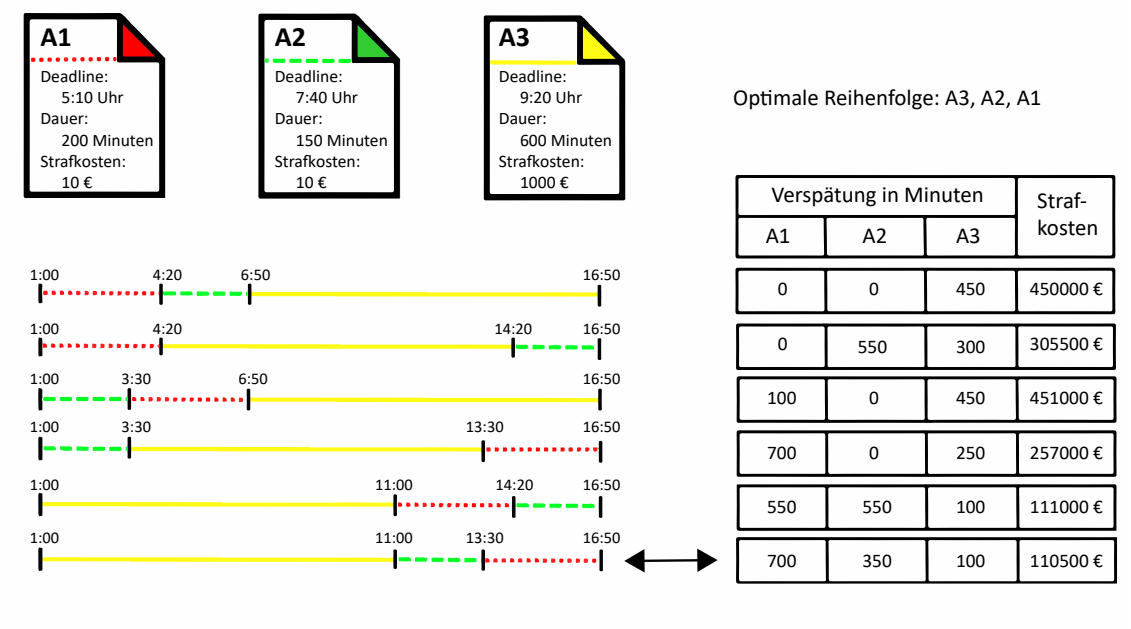
Das AWM Programm berechnet folgende Ausgabe:

$$reihenfolge\_auftrag(1, a1).\ reihenfolge\_auftrag(2, a2).\ reihenfolge\_auftrag(3, a3).$$
$$kosten\_ges(21000).$$

Die Ausgabe ist korrekt, wie an der Tabelle, in der alle möglichen Kombinationen mit deren Strafkosten aufgelistet sind, nachvollzogen werden kann.

4. Fall: Mehrere Aufträge mit geringen Strafkosten

In diesem Fall wird erneut ein Spezialfall getestet. Dabei gibt es mehrere Aufträge mit geringen Strafkosten, die ohne Verspätung sein könnten, jedoch gibt es einen Auftrag mit hohen Strafkosten, sodass die Aufträge mit geringen Strafkosten bei der optimalen Reihenfolge eine Verspätung haben. Der Repräsentant in Abbildung 4.16 wird stellvertretend getestet.



Verspätung in Minuten			Straf-kosten
A1	A2	A3	
0	0	450	450000 €
0	550	300	305500 €
100	0	450	451000 €
700	0	250	257000 €
550	550	100	111000 €
700	350	100	110500 €

Abbildung 4.16: Ein Test für den Fall *Mehrere Aufträge mit geringen Strafkosten.*

Das AWM Programm berechnet folgende Ausgabe:

$$reihenfolge\_auftrag(1, a3).\ reihenfolge\_auftrag(2, a2).\ reihenfolge\_auftrag(3, a1).$$
$$kosten\_ges(110500).$$

Die Ausgabe ist korrekt, wie an der Tabelle, in der alle möglichen Kombinationen mit deren Strafkosten aufgelistet sind, nachvollzogen werden kann.

Fazit

Das Programm hat alle Tests erfolgreich bestanden. Die Tests wurden so ausgewählt, dass neben Standardfällen auch Spezialfälle getestet wurden. Damit wurde das Vertrauen in das Encoding mit einer geringen Anzahl an Testfällen deutlich erhöht.

4.4 Energieknappheit

Dieses Programm verteilt alle verfügbaren Jobs aus einem Batch so auf den Fahrzeugen auf, dass die Fahrzeuge mit schwachem Energiestand nach der vollständigen Abarbeitung von dem Batch minimiert werden.

4.4.1 Beschreibung der Problemstellung

In einem System, in dem die Kommissionierung batchweise abläuft, ist es wichtig, dass immer genügend einsatzfähige Fahrzeuge verfügbar sind. Das bedeutet, dass wenn neue Aufträge eintreffen, sich so wenig Fahrzeuge wie möglich an den Ladestationen befinden sollen. Deshalb weist dieses AWM Programm alle verfügbaren Jobs aus einem Batch so den Fahrzeuge zu, dass die Fahrzeuge mit schwachem Energiestand nach der vollständigen Abarbeitung des Batches minimiert werden. Dabei ist ein Batch eine Zusammenstellung von mehreren Aufträgen zu einer geordneten Liste von Aufträgen und die Batch-Kommissionierung eine Kommissionierstrategie, bei der der Batch kontinuierlich durch den nächsten Auftrag gefüllt wird, sobald ein Auftrag vollständig abgearbeitet ist [3].

Den Fahrzeugen steht nur ein bestimmter Energiestand zur Verfügung. Dabei haben die einzelnen Jobs jeweils einen bestimmten Energieverbrauch, der für die Erledigung des Jobs aufgewendet werden muss. Die berechnete Verteilung der Jobs auf die Fahrzeuge soll garantieren, dass alle offenen Jobs erledigt werden und der Energiestand, nach der Bearbeitung der Jobs, von keinem Fahrzeug kleiner ist als null. Der Energiestand eines Fahrzeugs ist schwach, falls er einen festgelegten Schwellwert unterschreitet. Dabei sind Fahrzeuge, welche einen Energiestand unter dem Schwellwert haben, nicht mehr in der Lage weitere Jobs zu kommissionieren. Die Idee der Beschreibung kann anhand der Abbildung 4.17 nachvollzogen werden.

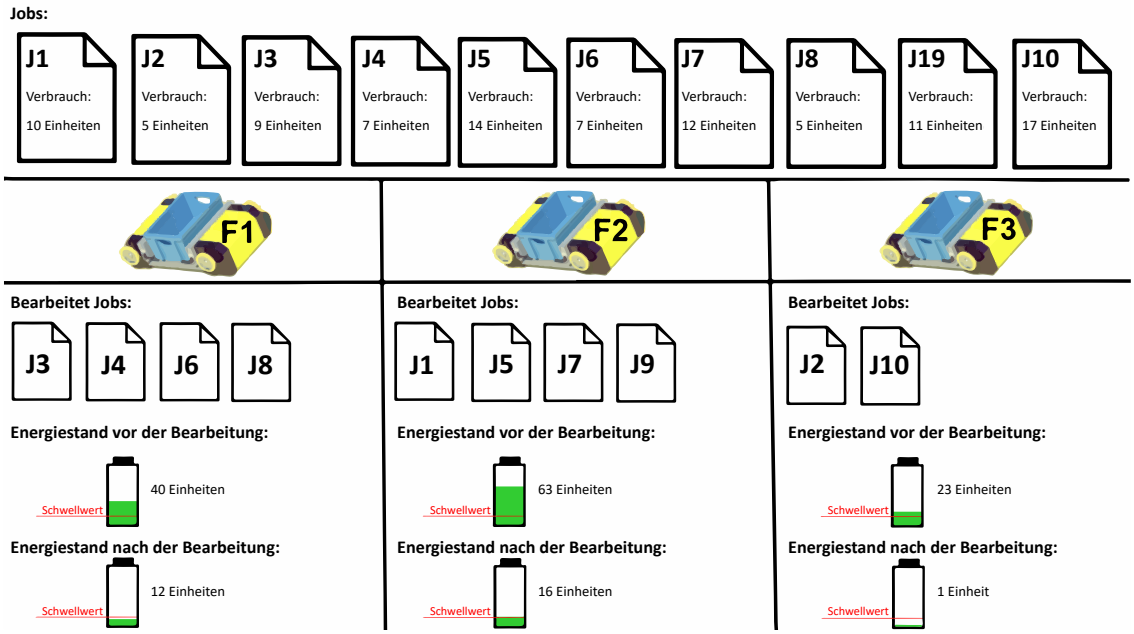


Abbildung 4.17: Ein Beispiel für die Aufgabe Energieknappheit.

In dem Beispiel gibt es drei Fahrzeuge $F1$, $F2$ und $F3$. Zudem gibt es die Jobs $J1$ bis $J10$, welchen jeweils ein Energieverbrauch zugeordnet ist. Bei dem Energieverbrauch handelt es sich um eine Schätzung, welche angibt, wie viel Energie für die Bearbeitung des Jobs von einem Fahrzeug aufgewendet werden muss. Außerdem wird der Energiestand der Fahrzeuge vor der Bearbeitung der zugewiesenen Aufgaben angegeben. Dabei werden die Jobs so wie in der Abbildung angegeben auf die Fahrzeuge aufgeteilt. Auch der Energiestand der Fahrzeuge nach der Bearbeitung der zugewiesenen Jobs kann anhand der Abbildung nachvollzogen werden. Bei der Aufteilung der Jobs auf die Fahrzeuge handelt es sich um eine optimale Lösung für die visualisierte Situation, da es in dieser Situation höchstens ein Fahrzeug ohne schwachen Energiestand geben kann. Bei allen Beispielen in diesem Kapitel liegt der Schwellwert für den Energiestand von einem Fahrzeug bei 15 Einheiten.

4.4.2 Ablauf

Ein Agent, vorzugsweise der Auktionator, da er schon alle Jobs kennt, berechnet den Energieverbrauch der Jobs. Dafür muss der Energieverbrauch eines einzelnen Jobs bestimmt werden. Aus diesem Grund werden Aktionen Kosten zugewiesen. Im Folgenden werden für die Aktionen des Agenten Energieverbräuche geschätzt. Dabei wird angenommen, dass bei jedem Meter den sich das Fahrzeug bewegt, 0.1 Energieeinheiten verbraucht werden. Auch wenn das Fahrzeug bei der Benutzung des Lifts passiv ist, verbraucht es zum Beispiel Energie um festzustellen, ob es an der Zielebene angekommen ist. Deshalb werden bei der Benutzung des Lifts für jede Etage die das Fahrzeug fährt 0.05 Energieeinheit verbraucht. Zudem werden 0.25 Energieeinheiten bei der Benutzung der Kommissionierstation benötigt. Auch beim Abladen und Aufladen ist das Fahrzeug passiv, jedoch werden hier für die Überprüfung des Ladevorgangs jeweils 0.05 Energieeinheiten verbraucht. Der Energieverbrauch beim Warten von einem Fahrzeug wird vernachlässigt.

Mit dieser Vorberechnung erhalten wir einen guten Wert für den Energieverbrauch von jedem Job. Diese berechneten Werte werden als Fakten in das AWM Programm mit dem Prädikat *job_ energie(J,E)*, wobei J der Job und E der berechnete Energieverbrauch ist, geschrieben. Als nächstes werden die aktuellen Energiestände der Fahrzeuge benötigt. Der Auktionator hat eine Liste von allen Fahrzeugen, die sich bei ihm angemeldet haben. Er fragt bei jedem Fahrzeug in der Liste die Energiestände an und trägt diese in das AWM Programm mit dem Prädikat *fahrzeug_ energie(F,E)* ein, wobei F das Fahrzeug und E der aktuelle Energiestand von dem Fahrzeug ist. Das Programm versucht nun, wie in der Problembeschreibung beschrieben, eine gültige Verteilung der Jobs zu berechnen. Der Auktionator weißt die Jobs dann aufgrund der berechneten Verteilung den Fahrzeugen zu.

4.4.3 Encoding

Für die beschriebene Problemstellung aus Abschnitt 4.4.1 wird im Folgenden ein Encoding vorgestellt. Das Encoding ist im Listing 4.4 zu finden. Für die Vorstellung des Encodings werden zunächst dessen Eingabe- und Ausgabeprädikate angegeben und anschließend alle Regeln des Encodings ausführlich erläutert.

Eingabe:

- $fahrzeug\_energie(F, E)$
  Der aktuelle Energiestand E des Fahrzeugs F.
- $job\_energie(J, E)$
  Die Energie E, die der Job J verbraucht.

Ausgabe:

- $faehrt(F, J)$
  Das Fahrzeug F soll den Job J übernehmen.

```
1 1{faehrt(F,J) : fahrzeug_energie(F,E), E >= Y }1 :- job_energie(J,Y).
2
3 fahrzeug_restenergie(F,A-V) :- fahrzeug_energie(F,A),
      V = #sum{K,J : faehrt(F,J), job_energie(J,K)}.
4
5 :- fahrzeug_restenergie(F,R), R < 0.
6
7 :- S = #sum{K,J : job_energie(J,K)},
      H = #sum{E,F : fahrzeug_energie(F,E)}, S > H.
8
9 anzahl_schwacher_fahrzeuge(S) :-
      S = #count{F : fahrzeug_restenergie(F,R), R < 15}.
10
11 #minimize{S : anzahl_schwacher_fahrzeuge(S)}.
12
13 #show faehrt/2.
```

Listing 4.4: Das Encoding für die Aufgabe Energieknappheit.

In der ersten Zeile wird jedem Job genau ein Fahrzeug zugewiesen, dabei werden für einen Job nur die Fahrzeuge berücksichtigt, die noch genügend Energie für diesen haben. Für jedes Fahrzeug wird mit der Regel aus der dritten Zeile die Energie berechnet, die ihm nach der Erledigung der zugewiesenen Jobs noch zur Verfügung steht. In der fünften Zeile werden die Lösungskandidaten entfernt, in denen mindestens ein Fahrzeug mehr Energie durch die Jobs verbrauchen würde, als ihm zur Verfügung steht. Mit dem Constraint in Zeile sieben wird überprüft, ob die gesamte Energie von den Jobs schon die Energie der Fahrzeuge übersteigt, dann wäre nämlich keine Lösung möglich. In der neunten Zeile wird die Anzahl der Fahrzeuge mit einem schwachen Energiestand bestimmt. Diese Anzahl wird in Zeile elf minimiert. Zeile 13 sorgt dafür, dass nur das angegebene Prädikat ausgegeben wird.

4.4.4 Validierung

Im Folgenden soll das Encoding validiert werden. Für die Validierung wird das Encoding geeignet getestet. Die Tests decken dabei sowohl Standard- als auch Ausnahmesituationen ab.

Test: Ein Fahrzeug mit schwachem Energiestand

Diese Situation steht stellvertretend für alle Situationen, in denen das Minimum der Fahrzeuge mit schwachem Energiestand genau eins ist. Anhand der Situation in Abbildung 4.18 soll beispielhaft überprüft werden, ob sich das Encoding in solchen Situationen korrekt verhält.

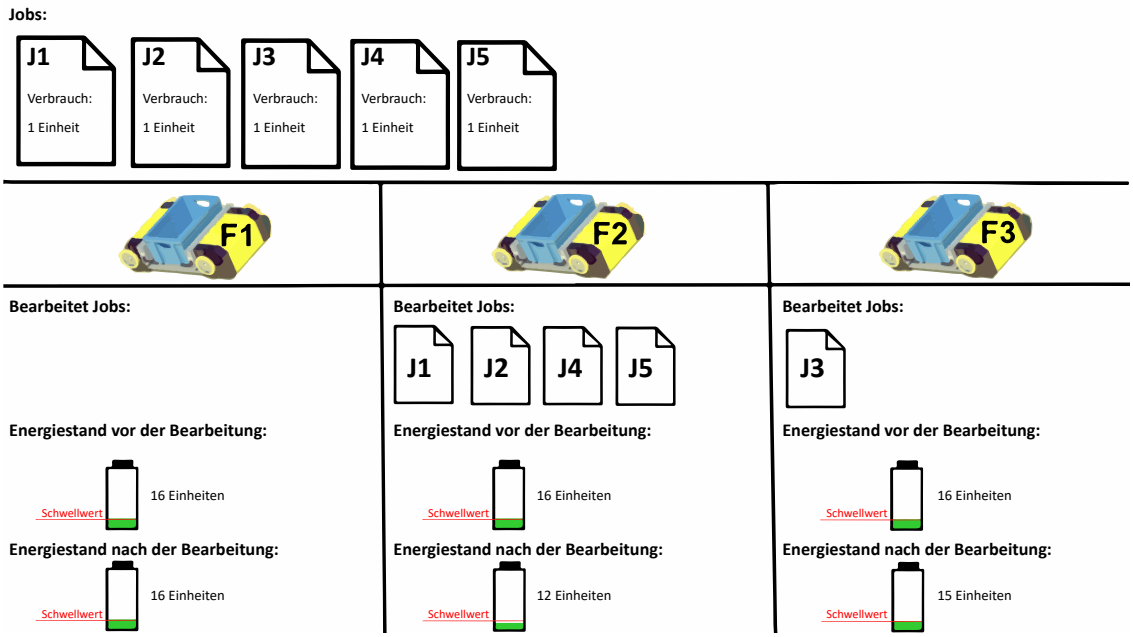


Abbildung 4.18: Ein Test für den Fall *Genau ein Fahrzeug mit schwachem Energiestand.*

Das AWM Programm berechnet folgende Ausgabe:

$$faehrt(f2, j1).\ faehrt(f2, j2).\ faehrt(f2, j4).\ faehrt(f2, j5).\ faehrt(f3, j3).$$

Die Zuweisung der Jobs auf die Fahrzeuge durch das AWM Programm ist korrekt, da es nur ein Fahrzeug mit schwachem Energiestand gibt und das Minimum für diese Situation bei einem Fahrzeug liegt.

Test: Kein Fahrzeug mit schwachem Energiestand

Mit diesem Test sollen, stellvertretend durch Abbildung 4.19, alle Situationen getestet werden, in denen es möglich ist, dass nach der Bearbeitung der Jobs der Energiestand von keinem Fahrzeug schwach ist.

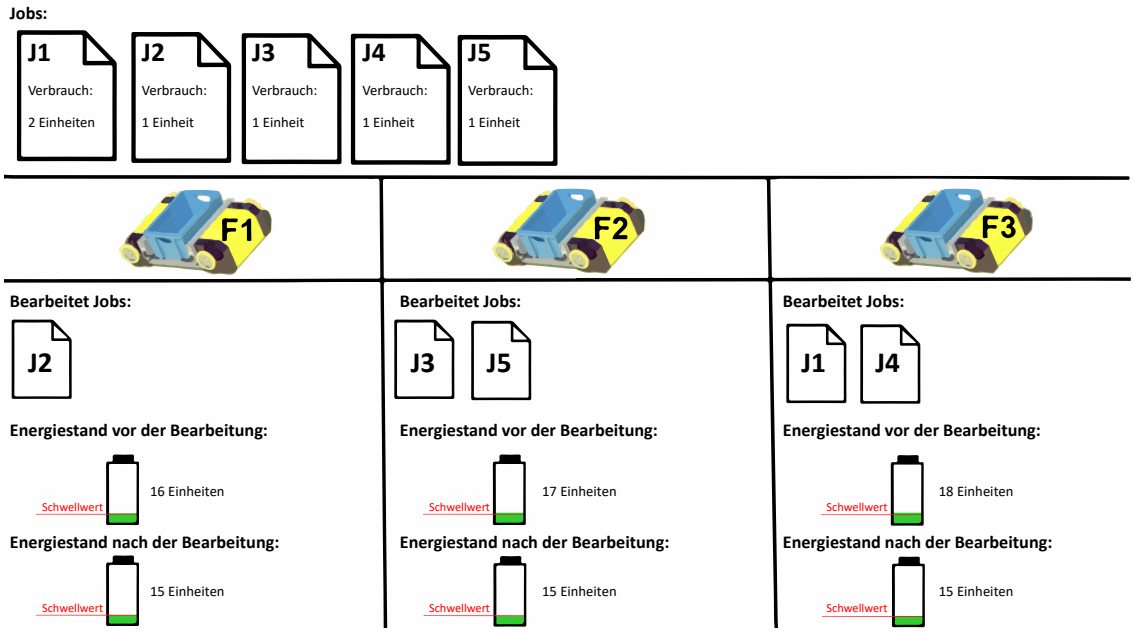


Abbildung 4.19: Ein Test für den Fall *Kein Fahrzeug mit schwachem Energiestand.*

Das AWM Programm berechnet folgende Ausgabe:

$$faehrt(f1, j2).\ faehrt(f2, j3).\ faehrt(f2, j5).\ faehrt(f3, j1).\ faehrt(f3, j4).$$

Die Ausgabe des AWM Programms ist korrekt, da es kein Fahrzeug mit schwachem Energiestand gibt.

Test: Gesamtenergie der Jobs > Gesamtenergie der Fahrzeuge

Der Test des Repräsentanten aus der Abbildung 4.20 steht stellvertretend für alle Situationen, in denen die Gesamtenergie der Jobs größer ist als die Gesamtenergie der Fahrzeuge.

Abbildung 4.20: Ein Test für den Fall *Gesamtenergie der Jobs > Gesamtenergie der Fahrzeuge.*

Das AWM Programm berechnet folgende Ausgabe:

$$UNSATISFIABLE$$

Das Programm stellt richtig fest, dass es keine gültige Verteilung der Jobs auf die Fahrzeuge gibt.

Test: Mehr als ein Fahrzeug mit schwachem Energiestand

Diese Situation steht stellvertretend für alle Situationen, in denen das Minimum der Fahrzeuge mit schwachem Energiestand größer ist als eins. Anhand der Abbildung 4.21 soll beispielhaft überprüft werden, ob sich das Encoding in solchen Situationen korrekt verhält.

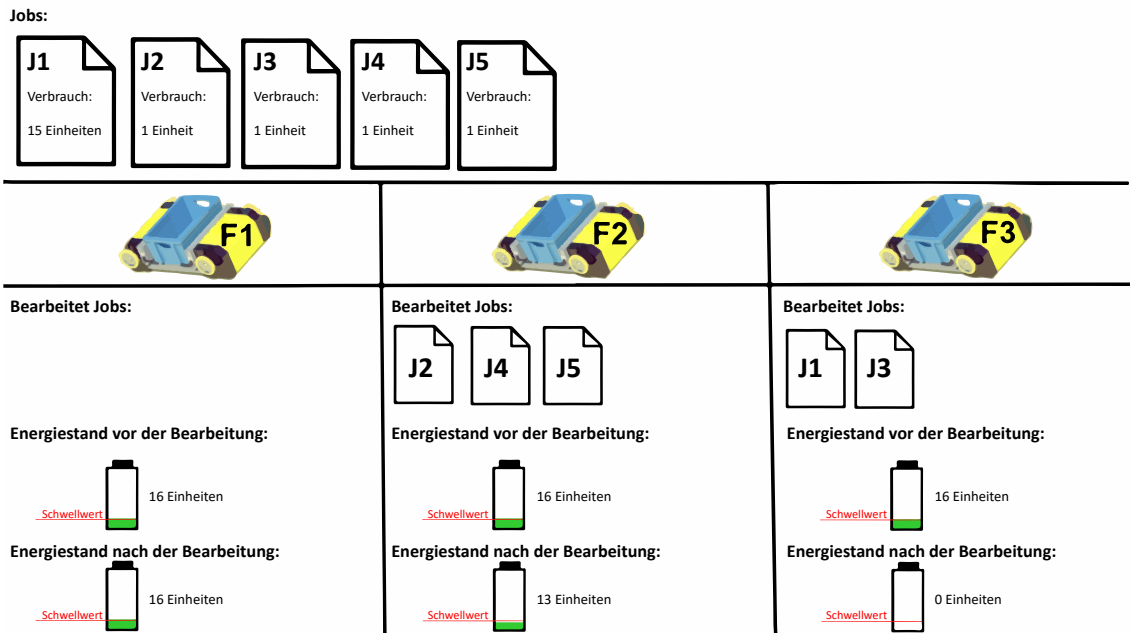


Abbildung 4.21: Ein Test für den Fall *Mehr als ein Fahrzeug mit schwachem Energiestand.*

Das AWM Programm berechnet folgende Ausgabe:

$$faehrt(f2, j5).\ faehrt(f2, j2).\ faehrt(f2, j4).\ faehrt(f3, j1).\ faehrt(f3, j3).$$

Die Zuweisung der Jobs auf die Fahrzeuge ist korrekt, da dadurch das Minimum der Fahrzeuge mit schwachem Energiestand für diese Situation erreicht wird.

Fazit

Das Programm hat alle Tests erfolgreich bestanden. Die Tests wurden so ausgewählt, dass alle Situationen die nicht getestet wurden sich äquivalent zu einen der getesteten Situationen verhalten würde. Damit wurde das Vertrauen in das Encoding mit einer sehr geringen Anzahl an Testfällen stark erhöht.

4.5 Kollisionsfreiheit

Dieses Programm berechnet für ein Fahrzeug mit einem Zielknoten einen minimalen Weg zu dem Zielknoten ohne Kollisionen mit anderen Fahrzeugen im System.

4.5.1 Beschreibung der Problemstellung

Im bestehenden Ansatz, welcher durch das Framework aus der Bachelorarbeit [13] repräsentiert wird, müssen die Fahrzeuge bei jedem Knoten den sie anfahren wollen, beim Lokalisator nachfragen, ob dieser Knoten frei ist. Um diese Fragen zu vermeiden, berechnet das AWM Programm für ein Fahrzeug mit einem Zielknoten einen minimalen Weg zu dem Zielknoten ohne Kollisionen mit anderen Fahrzeugen. Dabei beobachtet der Lokalisator die Positionen und die Strecken der Fahrzeuge im System. Mit Hilfe vom AWM Programm müsste ein Fahrzeug nur pro Weg eine Anfrage an den Lokalisator senden und nicht für jeden Wegpunkt auf dem Weg. Zudem kann es mit dem bestehenden Ansatz zu Deadlocks kommen, welche im neuen Ansatz vermieden werden. Das neue System unterscheidet sich in der Arbeitsweise sehr stark vom bestehenden Ansatz. Im neuen Ansatz fahren die Fahrzeuge synchronisiert und verlassen ihren Knoten erst in Richtung des neuen Knotens, wenn alle Fahrzeuge bereit sind. Die Fahrzeuge sind bereit, wenn alle an ihrer Position angekommen sind, sich also nicht mehr auf einer Wegstrecke befinden und ihre Aufgabe an der Position verrichtet haben. Da Fahrzeuge an einer Position beliebig lange warten können, muss das Programm eine Beschränkung für die Weglänge haben. Damit wird eine Antwort in praktikabler Zeit garantiert. Die Idee kann anhand der Abbildung 4.22 nachvollzogen werden.

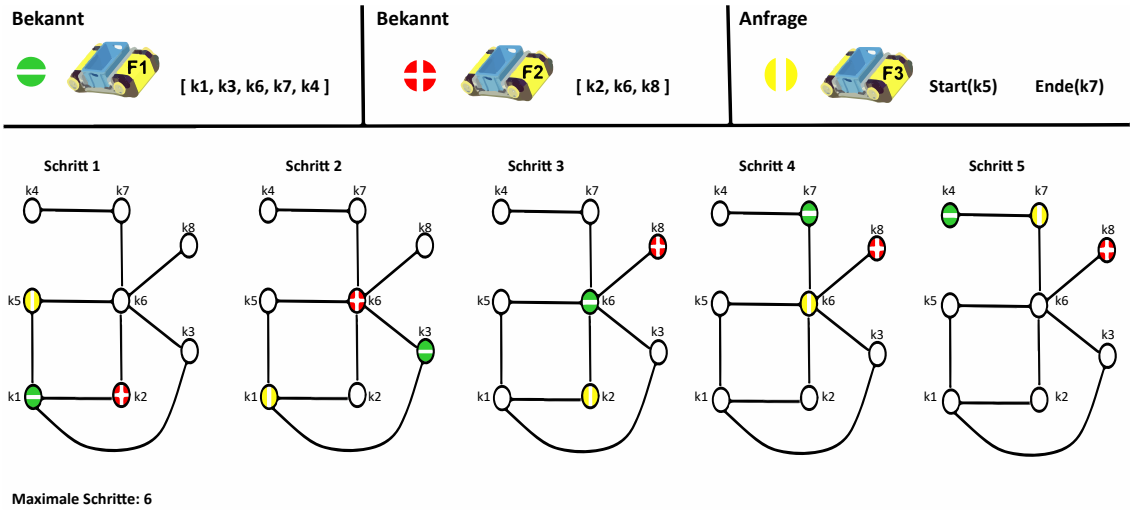


Abbildung 4.22: Ein Beispiel für die Aufgabe Kollisionsfreiheit.

In dem Beispiel befinden sich die Fahrzeuge $F1$, $F2$ und $F3$. Die Pfade von den Fahrzeugen $F1$ und $F2$ sind bereits bekannt und wurden neben den Abbildungen der Fahrzeuge visualisiert. Dabei bedeutet zum Beispiel der Pfad $[k2, k6, k8]$, dass ein Fahrzeug im Schritt 1 den Knoten $k2$, im Schritt 2 den Knoten $k6$ und im Schritt 3 den Knoten $k8$ im Weggraphen belegt. Das bedeutet, dass ein Fahrzeug in jeder Zeiteinheit genau von einem Knotenpunkt zu einem anderen fährt, wobei sich der Start- und Zielknoten in einem Schritt nicht un-

terscheiden müssen. Zudem ist in dieser Situation die maximale Anzahl an Schritten für das anfragende Fahrzeug auf sechs begrenzt. Die Angabe $Start(k5)$ bedeutet, dass sich das anfragende Fahrzeug aktuell am Knoten $k5$ befindet. Außerdem bedeutet die Angabe $Ende(k7)$, dass das anfragende Fahrzeuge den Knoten $k7$ anfahren möchte. Die einzelnen Weggraphen in der Abbildung visualisieren die Situationen im Weggraphen nach jedem Zeitschritt.

4.5.2 Ablauf

Der Lokalisator besitzt den Weggraphen und überführt die Kanten des Graphen in die vorgesehene Eingabeform für das AWM Programm. Die modellierten Kanten fügt er dann als Fakten in das Programm ein. Zudem kennt der Lokalisator die Position von jedem Fahrzeug im System. Wenn ein Fahrzeug den Weggraphen benutzen möchte, um an seinen Zielknoten zu gelangen, dann stellte das Fahrzeug eine Anfrage an den Lokalisator. Der Lokalisator berechnet dann einen kollisionsfreien und minimalen Weg, indem er die bereits vorliegenden Fahrplänen der anderen Fahrzeuge für die Berechnung berücksichtigt. Das Ergebnis der Berechnung speichert der Lokalisator in seinen Fahrplänen ab und sendet den Fahrplan an das anfragende Fahrzeug, welches nun diesen Weg fährt.

4.5.3 Encoding

Im Folgenden wird ein Encoding für die beschriebene Problemstellung aus Abschnitt 4.5.1 vorgestellt. Dafür werden zunächst die Eingabe- und Ausgabeprädikate für das Encoding angegeben. Anschließend wird das Encoding vorgestellt und alle Regeln des Encodings ausführlich erläutert. Das Encoding ist im Listing 4.5 zu finden.

Eingabe:

- $kante(X, Y)$
  Eine gerichtete Kante aus dem Weggraphen, die den Wegpunkt X mit dem Wegpunkt Y verbindet.
- $max\_schritte(T)$
  Die maximale Anzahl von Schritten T, die das anfragende Fahrzeug fahren darf.
- $belegt(F, K, T)$
  Das Fahrzeug F belegt den Knotenpunkt K im Schritt T.
- $start(K)$
  Der Startknoten K vom anfragenden Fahrzeug.
- $ende(K)$
  Der Zielknoten K vom anfragenden Fahrzeug.

Ausgabe:

- $fahre\_zu(K, T)$
  Das anfragende Fahrzeug soll im Schritt T zu dem Knotenpunkt K fahren.

- *ziel_erreicht*(T)
  Das anfragende Fahrzeug hat seinen Zielknoten nach T Schritten erreicht.

```
1  knoten(K) :- kante(K,_).
2
3  knoten(K) :- kante(_,K).
4
5  kante(X,X) :- knoten(X).
6
7  schritt(1..T) :- max_schritte(T).
8
9  belegt(K,T) :- belegt(_,K,T).
10
11 probiere(K,1) :- start(K).
12
13 1{probiere(K,T) : knoten(K)}1 :- schritt(T), T > 1.
14
15 ziel_erreicht(T) :- probiere(K,T), ende(K), not ziel_erreicht(T-1).
16
17 :- probiere(K1,T), probiere(K2,T+1), not kante(K1,K2).
18
19 :- not {ziel_erreicht(T)} = 1.
20
21 :- belegt(K,T), probiere(K,T).
22
23 :- probiere(K1,T-1), probiere(K2,T), belegt(F,K2,T-1), belegt(F,K1,T),
       K1 != K2, T > 1.
24
25 #minimize{T : ziel_erreicht(T)}.
26
27 fahre_zu(K,E) :- ziel_erreicht(T), E <= T, probiere(K,E).
28
29 #show fahre_zu/2.
30 #show ziel_erreicht/1.
```

Listing 4.5: Ein Encoding für die Aufgabe Kollisionsfreiheit.

In den Zeilen eins und drei werden die Knoten aus den Kanten extrahiert. Mit der Regel in Zeile fünf wird garantiert, dass es für jeden Knoten eine Kante zu sich selbst gibt. Dies ist wichtig, da damit das Warten eines Fahrzeugs an einem Wegpunkt realisiert wird. Für die Strecke sind maximal T Schritte verfügbar, welches in Zeile sieben definiert wird. In der neunten Zeile wird das Eingabeprädikat *belegt* um die Fahrzeug Information reduziert, da sie in vielen Fällen nicht relevant ist. Zeile elf legt den Startpunkt des anfragenden Fahrzeugs fest. In Zeile 13 wird festgelegt, dass das anfragende Fahrzeug in jedem Schritt zu einem Knotenpunkt fährt. Dabei ist es auch möglich, dass das Fahrzeug an einem Knoten X wartet, mit *probiere*(X,T) und *probiere*$(X,T+1)$. Das Ziel ist erreicht, wenn das Fahrzeug zum ersten Mal an seinem Zielknoten angekommen ist. Das wird durch die Regel in Zeile 15 modelliert. Mit dem Constraint in Zeile 17 wird sichergestellt, dass es nicht möglich ist, in einem Schritt von einem Knoten zu einem anderen zu fahren, wenn diese nicht durch eine Kante miteinander verbunden sind. Zudem sind alle Modelle die das Ziel nicht in der angegebenen Schrittzahl erreichen keine valide Option für das Problem. Diese Bedingung wird mit dem Constraint in Zeile 19 modelliert. Die Regel in Zeile 21 garantiert, dass es zu keiner Kollision kommt, indem sichergestellt wird, dass das Fahrzeug nicht zu einem Knoten fährt, der in dem Schritt von einem anderen Fahrzeug bereits belegt ist. Mit dem Constraint in Zeile 23 wird sichergestellt, dass keine Kante im Weggraphen

zu einem Zeitpunkt gleichzeitig von zwei Fahrzeugen benutzt wird. Mit der Zeile 25 wird die Anzahl der Schritte, die zum Erreichen des Zielknotens benötigt wird, minimiert. In Zeile 27 wird die Ausgabe optimiert, indem nur die relevanten Fahranweisungen für das Fahrzeug ausgegeben werden. Damit erhält das Fahrzeug einen Fahrplan mit minimalen Schritten zum Zielknoten ohne Kollisionen mit anderen Fahrzeugen. Die Zeilen 29 und 30 beschränken die Ausgabe auf die angegebenen Prädikate.

4.5.4 Validierung

Die Validierung des Programms wird in zwei Bereiche unterteilt. Zunächst soll sicherstellen, dass die Idee hinter dem Encoding korrekt ist. Danach soll strukturiert getestet werden, ob das Encoding korrekt arbeitet.

Idee korrekt

Das Programm soll garantieren, dass die entworfenen Lösungen kollisionsfrei sind. Aus diesem Grund wird eine Idee benötigt, um Kollisionen zu erkennen. Für die Kollisionserkennung wird für jede potenzielle Lösung in jedem Schritt überprüft, ob sich in einem Schritt zwei unterschiedliche Fahrzeuge befinden, die denselben Knoten belegen würden. Im Folgenden soll bewiesen werden, dass diese Strategie korrekt ist.

Aussage: Wenn zwei unterschiedliche Fahrzeuge an derselben Position in ihrem Fahrplan denselben Knoten haben, dann kommt es im System zu einer Kollision zwischen diesen Fahrzeugen.

Beweis: Angenommen zwei unterschiedliche Fahrzeuge dürften in ihren Wegen in demselben Schritt denselben Knoten haben, ohne dass es zu einer Kollision kommt. Alle Knoten im Weggraphen G seien exklusiv belegbar. Sei $W_1 = [K_1, ..., K_P, ..., K_M]$ der Weg von Fahrzeug F_1 und $W_2 = [Q_1, ..., K_P, ..., Q_N]$ der Weg von Fahrzeug F_2. Dabei repräsentieren $K_1...K_M$ und $Q_1...Q_N$ Knoten des Weggraphen G, wobei K_i beziehungsweise Q_i der Knoten ist, der von einem Fahrzeug im Schritt i belegt wird. Damit belegen die Fahrzeuge F_1 und F_2 den Knoten K_P zur selben Zeit. Das ist jedoch ein Widerspruch zur exklusiven Belegung von einem Knoten. Damit konnte gezeigt werden, dass wenn zwei Fahrzeuge in ihren Wegen in demselben Schritt denselben Knoten haben, es zu einer Kollision kommt und damit die Idee der Kollisions-Erkennung korrekt ist.

Encoding korrekt

Um das AWM Encoding zu testen, werden Testfälle generiert, welche neben den Standardfällen auch Spezialfälle betrachten.

Test: Kollision

Diese Situation steht stellvertretend für alle Situationen, in denen es keinen kollisionsfreien Weg gibt. Das Programm sollte dann ausgeben, dass es keine Lösung für die Anfrage gibt. Anhand der Situation aus Abbildung 4.23 soll beispielhaft überprüft werden, ob sich das Programm in solchen Situationen korrekt verhält.

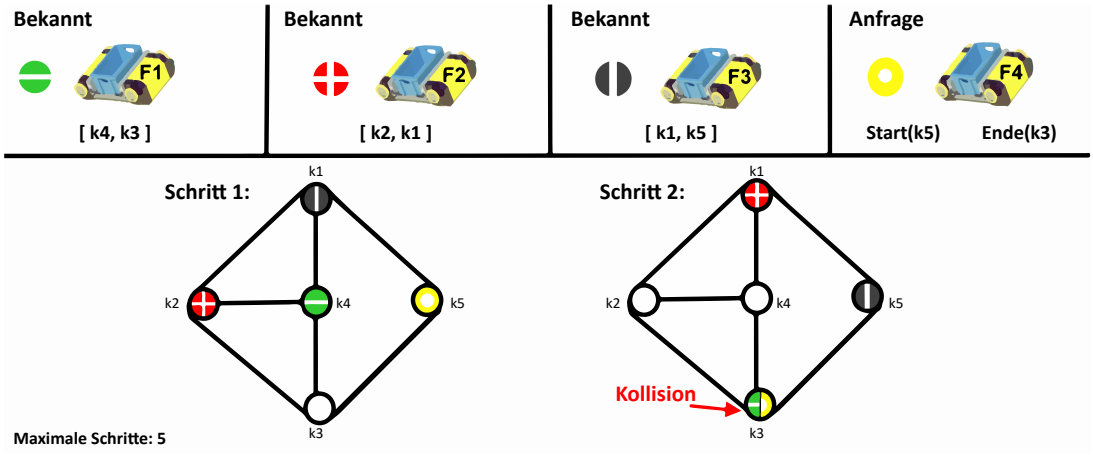


Abbildung 4.23: Ein Test für den Fall *Kollision*.

Das AWM Programm berechnet folgende Ausgabe:

$$UNSATISFIABLE$$

Das Programm gibt korrekt aus, dass es keine Lösung für diese Anfrage gibt. Somit hat das Programm den Test erfolgreich bestanden.

Test: Keine Kollision

Mit diesem Test werden Situationen getestet, in denen es mindestens einen kollisionsfreien Weg gibt, welcher die maximale Anzahl der Schritte einhält. Anhand der Situation aus Abbildung 4.24 soll beispielhaft überprüft werden, ob sich das Programm in solchen Situationen korrekt verhält.

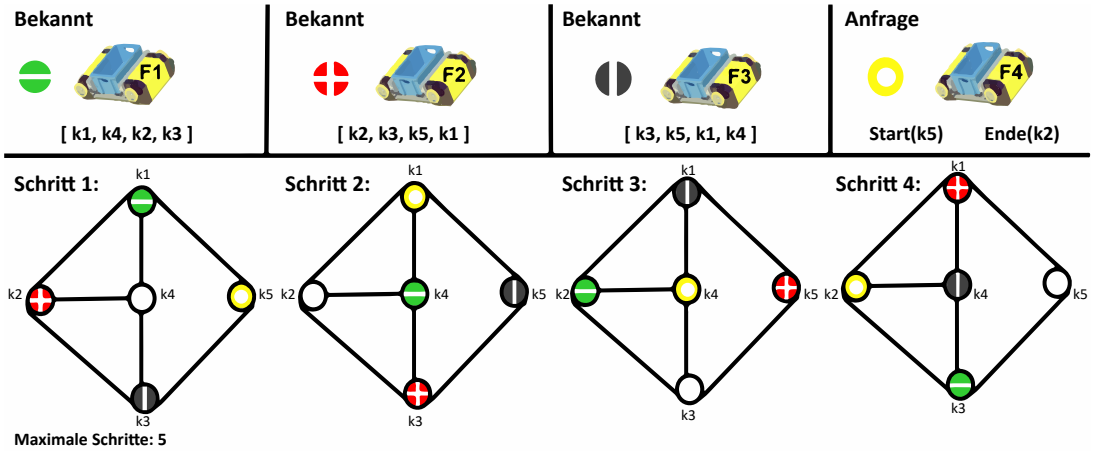


Abbildung 4.24: Ein Test für den Fall *Keine Kollision*.

Das AWM Programm berechnet folgende Ausgabe:

$$fahre\_zu(k5,1).\ ziel\_erreicht(4).\ fahre\_zu(k2,4).\ fahre\_zu(k1,2).\ fahre\_zu(k4,3).$$

Die Ausgabe des Programms ist korrekt, da der ausgegebene Weg kollisionsfrei ist. Zudem ist es ein minimaler und kollisionsfreier Weg für diese Situation.

Test: Maximale Schritte

In diesem Programm kann es den Fall geben, dass eine Situation einen kollisionsfreien Weg hat, dieser Weg jedoch die maximale Schrittzahl überschreitet. In solchen Fällen soll das Programm ausgeben, dass es keine Lösung gibt. Mit der Situation, welche in Abbildung 4.25 visualisiert wurde, wird beispielhaft überprüft, ob sich das Programm auch bei diesen Spezialfällen korrekt verhält.

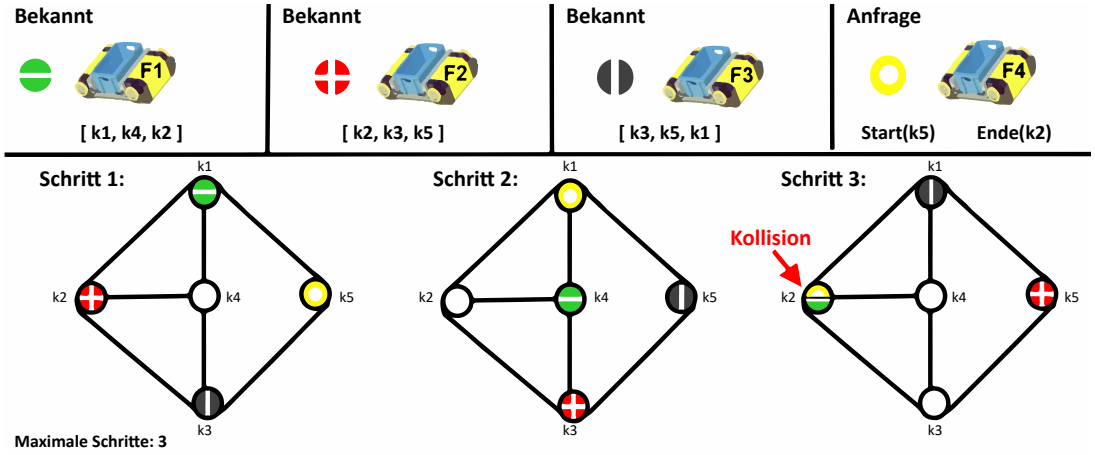


Abbildung 4.25: Ein Test für den Fall *Maximale Schritte*.

Das AWM Programm berechnet folgende Ausgabe:

UNSATISFIABLE

Die Ausgabe des Programms ist korrekt, da es keine Lösung gibt, die kollisionsfrei ist und die maximale Schrittzahl einhält. Damit hat das Programm auch diesen Test erfolgreich bestanden.

Fazit

Das Programm hat alle Tests erfolgreich bestanden. Die Tests wurden so ausgewählt, dass alle Situationen die nicht getestet wurden sich äquivalent zu einen der getesteten Situationen verhalten würde. Damit wurde das Encoding mit einer sehr geringen Anzahl an Testfällen strukturiert getestet. Die Idee hinter dem Encoding wurde als korrekt bewiesen. Zudem wurde das Vertrauen in das Encoding stark erhöht.

4.6 Nutzwertanalyse für die Implementierung der identifizierten AWM Aufgaben

In diesem Kapitel wurden mehrere AWM Aufgaben identifiziert, welche jedoch aus zeitlichen Gründen nicht alle implementiert werden können. In dieser Arbeit sollen dennoch zwei Aufgaben umgesetzt werden. Aus diesem Grund muss eine begründete Entscheidung zur Auswahl der Aufgaben getroffen werden. Diese Entscheidung soll mit Hilfe einer Nutzwertanalyse durchgeführt werden.

4.6.1 Beschreibung einer Nutzwertanalyse

Die Nutzwertanalyse ist eine Methodik der Entscheidungstheorie, mit der die Entscheidung bei komplexen Problemen rational unterstützt werden soll. Die Nutzwertanalyse findet in den Bereichen statt, in denen Beurteilungen auf Basis mehrerer quantitativer und qualitativer Kriterien, Zielen oder Bedingungen getroffen werden müssen [26]. Dabei wird eine Nutzwertanalyse häufig bei „weichen“ Kriterien verwendet, durch welche eine Entscheidung zwischen mehreren Alternativen gefällt werden soll. Eine Nutzwertanalyse hat nach [2] den folgenden Ablauf:

1. **Festlegung der Entscheidungsvarianten**
   Dabei werden die einzelnen Entscheidungsmöglichkeiten festgelegt.
2. **Definition von Bewertungskriterien**
   In diesem Schritt werden die Kriterien festgelegt, anhand derer eine Entscheidung getroffen werden soll.
3. **Gewichtung der Bewertungskriterien**
   Jedem Kriterium wird ein Prozentsatz hinterlegt, der die Wichtigkeit des Kriteriums belegt. Die Summe der Einzelgewichtungen muss 100% ergeben.
4. **Festlegung des Bewertungsmaßstabes**
   Die einzelnen Kriterien werden mit Punkten bewertet. Um hier eine Eindeutigkeit sicherzustellen, muss der Bewertungsmaßstab genau definiert werden, z.B. 5 Punkte = sehr gut, 1 Punkt = mangelhaft.
5. **Bewertung der Alternativen**
   Hier erfolgt die eigentliche Bewertung: Pro Kriterium und Alternative werden nun Punkte vergeben und die gewichteten Punkte berechnet.
6. **Summierung und Auswahl**
   Durch Summierung der Einzelgewichtungen ergibt sich die gewichtete Punktzahl pro Alternative. Die Alternative mit der höchsten Punktzahl entspricht den definierten Kriterien am besten.

4.6.2 Durchführung einer Nutzwertanalyse

1. Festlegung der Entscheidungsvarianten

Die Entscheidungsvarianten entsprechen den identifizierten AWM Aufgaben. Damit sind die Entscheidungsvarianten:

- Rückkehr
- Deadlock-Erkennung
- Minimiere Strafen
- Energieknappheit
- Kollisionsfreiheit

2. Definition von Bewertungskriterien

Die folgenden Bewertungskriterien sind subjektive Bewertungskriterien.

- **Umsetzbarkeit**
  Wie viel Zeit wird ungefähr benötigt, um das Programm im ZFT Framework zu implementieren und wie komplex ist diese Umsetzung?
- **Relevanz**
  Wie wichtig ist das Programm für das ZFT System? Gibt es ohne dem Programm Probleme im ZFT System oder kommt es zu starken zeitlichen Verzögerungen?
- **Bereits vorhanden**
  Ist die Funktionalität des Programms bereits im System vorhanden?
- **Zeitlicher Nutzen**
  Wie viel Zeit kann durch das Programm im Vergleich zu dem bestehenden Programm circa eingespart werden?

3. Gewichtung der Bewertungskriterien

In der Tabelle 4.3 wird jedem Kriterium ein Prozentsatz hinterlegt, der die Wichtigkeit des Kriteriums belegt.

Kriterium	Gewichtung
Umsetzbarkeit	30%
Relevanz	40%
Bereits vorhanden	10%
Zeitlicher Nutzen	20%

Tabelle 4.3: Gewichtung der Bewertungskriterien.

4. Festlegung des Bewertungsmaßstabes

In den Tabellen 4.4, 4.5, 4.6 und 4.7 werden die Bewertungsmaßstäbe aller Kriterien genau festgelegt.

Umsetzbarkeit:

Bewertungsmaßstab	Bedeutung
5	10 - 20 Arbeitsstunden
4	20 - 30 Arbeitsstunden
3	30 - 40 Arbeitsstunden
2	40 - 50 Arbeitsstunden
1	mehr als 50 Arbeitsstunden

Tabelle 4.4: Festlegung des Bewertungsmaßstabes für das Kriterium *Umsetzbarkeit*.

Relevanz:

Bewertungsmaßstab	Bedeutung
5	sehr wichtig
4	wichtig
3	gut, wenn vorhanden
2	optional
1	nicht wichtig

Tabelle 4.5: Festlegung des Bewertungsmaßstabes für das Kriterium *Relevanz*.

Bereits vorhanden:

Bewertungsmaßstab	Bedeutung
5	noch gar nicht vorhanden
4	vorhanden aber mangelhaft
3	vorhanden und befriedigend
2	vorhanden und gut
1	vorhanden und sehr gut

Tabelle 4.6: Festlegung des Bewertungsmaßstabes für das Kriterium *Bereits vorhanden*.

Zeitlicher Nutzen:

Bewertungsmaßstab	Bedeutung
5	viel mehr Ersparnis
4	mehr Ersparnis
3	wenig Ersparnis
2	Ersparnis kaum bemerkbar
1	kein Ersparnis

Tabelle 4.7: Festlegung des Bewertungsmaßstabes für das Kriterium *Zeitlicher Nutzen*.

5. Bewertung der Alternativen

In der Tabelle 4.8 werden für jedes Kriterium und jede Alternative Punkte vergeben und die gewichteten Punkte berechnet.

Kriterium	Gewichtung	Rückkehr	Deadlock-Erkennung	Minimiere Strafen	Energieknappheit	Kollisionsfreiheit
Umsetzbarkeit	30%	4 (1,2)	5 (1,5)	3 (0,9)	2 (0,6)	2 (0,6)
Relevanz	40%	2 (0,8)	5 (2,0)	3 (1,2)	3 (1,2)	2 (0,8)
Bereits vorhanden	10%	1 (0,1)	5 (0,5)	5 (0,5)	5 (0,5)	1 (0,1)
Zeitlicher Nutzen	20%	5 (1,0)	5 (1,0)	1 (0,2)	1 (0,2)	4 (0,8)

Tabelle 4.8: Bewertung der Alternativen.

6. Summierung und Auswahl

In der Tabelle 4.9 werden mit der Summierung der Einzelgewichtungen die gewichtete Punktzahl pro Alternative ermittelt.

Kriterium	Rückkehr	Deadlock-Erkennung	Minimiere Strafen	Energieknappheit	Kollisionsfreiheit
Umsetzbarkeit	1,2	1,5	0,9	0,6	0,6
Relevanz	0,8	2,0	1,2	1,2	0,8
Bereits vorhanden	0,1	0,5	0,5	0,5	0,1
Zeitlicher Nutzen	1,0	1,0	0,2	0,2	0,8
Summe	3,1	5	2,8	2,5	2,3

Tabelle 4.9: Summierung und Auswahl.

Nach Ermittlung und Aufsummierung der gewichteten Punktzahlen ergibt sich das in der Tabelle 4.10 repräsentierte Ergebnis:

Programm	Wert
Deadlock-Erkennung	5
Rückkehr	3,1
Minimiere Strafen	2,8
Energieknappheit	2,5
Kollisionsfreiheit	2,3

Tabelle 4.10: Auswertung der Nutzwertanalyse.

Die Tabelle 4.10 enthält alle Entscheidungsvarianten absteigend sortiert nach den gewichteten Punktzahlen. Damit wird die Reihenfolge der Implementierung festgelegt. Aus diesem Grund werden die Programme *Deadlock-Erkennung* und *Rückkehr* in dieser Arbeit umgesetzt.

4.7 Implementierung und Evaluierung von ausgewählten Aufgaben

In diesem Abschnitt werden die Implementierungen und Evaluierungsergebnisse der ausgewählten Aufgaben *Rückkehr* und *Deadlock-Erkennung* vorgestellt. Dafür wird bei den Implementierungen abstrakt beschrieben, wie die jeweilige Aufgabe realisiert wurde. Bei den Evaluierungen wird zunächst der experimentelle Aufbau für die jeweilige Evaluierung angegeben und anschließend werden die Ergebnisse vorgestellt.

4.7.1 Rückkehr

Im Folgenden wird für die Aufgabe *Rückkehr* (siehe Abschnitt 4.1) die Implementierung und Evaluierung angegeben.

4.7.1.1 Implementierung

Die Aufgabe *Rückkehr* wurde mit Hilfe des Frameworks aus der Bachelorarbeit [13] realisiert. Dabei sollen im Folgenden die grundlegenden Schritte für die Umsetzung erläutert werden. Für die Umsetzung wurde auf die entwickelte Systemarchitektur aus Abschnitt 3.1 zurückgegriffen. Die Systemarchitektur stellt eine abstrakte Klasse $KrProgramm$ bereit, welche den Aufbau für eine zu implementierende Aufgabe vorgibt. Für die Implementierung der Aufgabe wurde die Klasse $KrProgrammRueckkehr$ angelegt, welche die Klasse $KrProgramm$ erweitert. In dieser Klasse müssen lediglich zwei Methoden realisiert werden. Die erste zu implementierende Methode ist $config()$. In dieser Methode wurden die Informationen für die Eingabedaten angegeben, die der $Fahrzeug$ Agent zum Ausführen der Aufgabe *Rückkehr* benötigt. Außerdem wurde zu jeder dieser Information der Agententypen, der diese Information bereitstellt, angegeben. Die zweite Methode ist $interpretiere()$, in welcher die Aktionen für die berechnete Lösung des Solvers hinterlegt werden müssen. Dafür wurde eine Schleife entwickelt, welche die Lösung elementweise durchläuft und bei Vorkommen einer bestimmten ID die zugehörigen Methodenaufrufe durchführt. Dabei wurden die benötigten Aktionen in dem $Fahrzeug$ Agenten implementiert. Bei der zu implementierenden Aufgabe ist es wichtig, dass der bereits implementierte Auktionsansatz nicht ausgeführt wird. Deshalb wurde die Ausführung des Auktionsansatzes im System unterbunden. Zudem wurden für jede benötigte Information, bei dem jeweiligen Agententyp, die Eingabedaten generiert und geeignet zur Verfügung gestellt. Bei den Agenten $Auktionator$ und $Inventarmanager$ wurden zudem Methoden für die Verwaltung von Job- bzw. Stellplatzanfragen der Fahrzeuge umgesetzt. Der Rest wird von der Systemarchitektur erledigt. Die Systemarchitektur startet die Ausführung des Prozesses für diese Aufgabe Ereignisgesteuert. Dabei wird der Prozess aufgerufen, wenn das Fahrzeug eine Kommissionierstation verlässt. Die Implementierung wurde durch die entwickelte Systemarchitektur stark erleichtert.

4.7.1.2 Evaluierung

Im Folgenden soll die implementierte Aufgabe evaluiert werden. Dafür wird am Anfang der experimentelle Aufbau angegeben und anschließend werden die Ergebnisse der Experimente ausgewertet.

Experimenteller Aufbau

Die Experimente werden mit einem modifizierten Framework aus der Bachelorarbeit [13] durchgeführt. Dabei simuliert das Framework ein ZFT System und wurde mit Hilfe von JADE realisiert. Neben dem implementierten AWM Ansatz wird auch der bereits implementierte Auktionsansatz verwendet. In diesem Ansatz werden die Jobs in einer Auktion vergeben und die Ladungsträger an ihren ursprünglichen Stellplatz zurückbefördert. Die Simulationsparameter sind in der Tabelle 4.11 zu finden. Neben der Anzahl der Fahrzeuge, werden auch zwei verschiedene Weggraphen verwendet (siehe Abbildung 4.27 und 4.26).

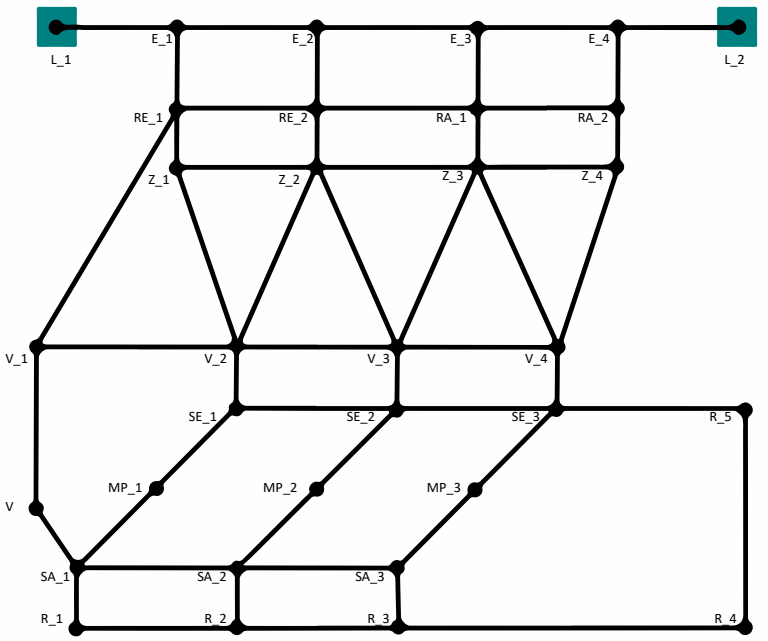


Abbildung 4.26: Der Weggraph 1.

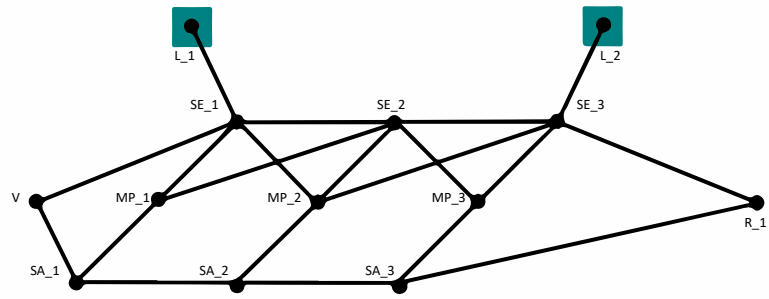


Abbildung 4.27: Der Weggraph 2.

Das verwendete System enthält, unabhängig vom Weggraphen, drei Kommissionierstationen. Zudem enthält es ein Regal, welches aus 40 Ladungsträgerplätzen besteht und fünf Etagen besitzt. Das System umfasst außerdem 40 Ladungsträger und die Lifte befinden sich am Anfang und am Ende des Regalsystems. In den Experimenten arbeiten die Fahrzeuge eine fest definierte Menge von Aufträgen (siehe Tabelle 4.12) ab. Dabei besteht die Menge sowohl aus kleinen als auch großen Aufträgen und enthält zudem Artikel, welche in mehreren Aufträgen benötigt werden. Die beiden Ansätze werden aufgrund der benötigten

Zeit für die Bearbeitung der Auftragsmenge verglichen. Es werden alle möglichen Parameterkombinationen evaluiert.

Parameter	Werte
Methode	AWM Ansatz, Auktionsansatz
Anzahl Fahrzeuge	1, 3, 5, 8, 10
Weggraph	Weggraph1, Weggraph2

Tabelle 4.11: Simulationsparameter für die Evaluierung.

Auftrag	Die benötigten Artikel für den Auftrag
1	35, 14, 12, 37, 3, 10, 1
2	5, 11, 1, 31, 26, 20, 14, 24, 2
3	10, 8, 9, 25, 11
4	7, 36, 35, 8, 11, 25
5	24, 3, 10
6	15, 5, 18, 23, 6, 40, 8, 36
7	15, 10, 35, 33, 23, 28, 38, 2, 5, 18
8	16, 22, 40, 25, 24, 23, 31, 30, 12, 36
9	25
10	4, 37, 38, 27, 39, 36
11	19, 21, 25, 8, 31, 3, 33, 39
12	8, 13, 27, 28, 15, 38, 20, 16, 3, 26
13	4, 1, 30, 10
14	2, 18, 15, 10, 11, 8
15	13, 15, 9
16	3, 13, 30, 20, 18, 11, 28, 31
17	19, 12, 29
18	9, 12, 32, 40, 13, 15, 7, 29, 35, 36
19	21, 37, 26, 17, 5
20	17, 21, 5, 22, 30, 18, 37, 31, 20
21	4, 32
22	15, 32, 10, 37, 23, 28, 5, 31, 3
23	20
24	12
25	31

Tabelle 4.12: Die Menge von zu erledigender Aufträge.

Ergebnisse

Die Simulationsergebnisse sind in Abbildung 4.28 zu finden. Im Folgenden werden die Ergebnisse detaillierter betrachtet. Die Ergebnisse sind abhängig vom Weggraphen visualisiert worden.

Bei dem *Weggraph*1 sinkt die Abarbeitungszeit der Aufträge, wenn die Anzahl der Fahrzeuge erhöht wird. Dabei wird deutlich, dass der AWM Ansatz in allen Simulationen bessere Zeiten erzielt als der Auktionsansatz. Jedoch wird auch deutlich, dass mit steigender Anzahl der Fahrzeuge, die Zeitunterschiede zwischen den Ansätzen abnehmen. Bei bis zu fünf Fahrzeugen besitzen beide Weggraphen dieselbe Steigung zwischen den Zeiten, von einer Anzahl der Fahrzeuge zu der nächsten. Die Abarbeitungszeiten sind im

*Weggraph*2 bei bis zu fünf Fahrzeugen geringer, als im Vergleich zu *Weggraph*1. Das liegt an dem kürzeren Weg von dem Regal zu den Kommissionierstationen und zurück. Dabei ist die Entwicklung der Zeit nach fünf Fahrzeugen interessant. Denn dann steigt die Bearbeitungszeit im *Weggraph*2 mit steigender Anzahl der Fahrzeuge. Dies liegt an dem kleineren Weggraphen und den daraus resultierenden Staus, die sich vor den Liften und Kommissionierstationen bilden.

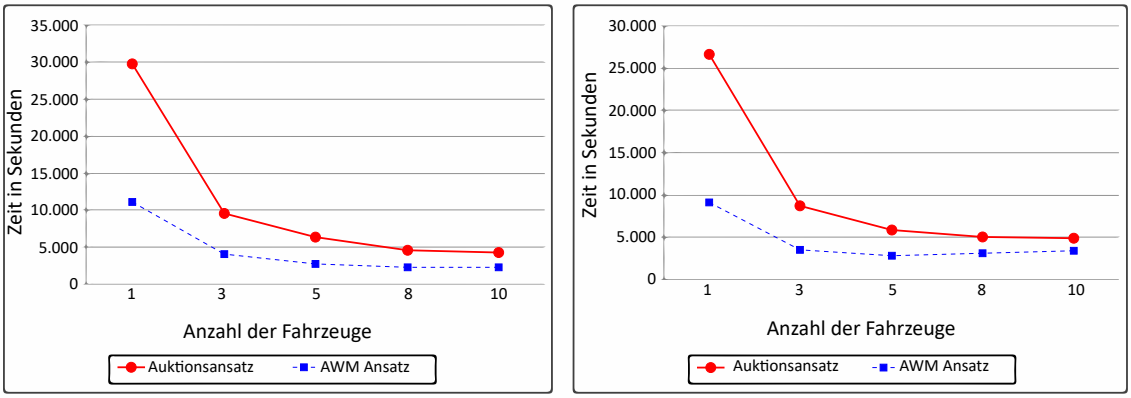


Abbildung 4.28: Die Evaluierungsergebnisse der Aufgabe Rückkehr. Links die Ergebnisse mit dem Weggraph1 und Rechts die Ergebnisse mit dem Weggraph2.

Im Folgenden werden die Zeiten für die Berechnungen der beiden Ansätze verglichen. Dafür wurde das Framework aus der Bachelorarbeit [13] um eine Zeitmessung für beide Ansätze erweitert. Der AWM Ansatz startet die Zeit, wenn ein Fahrzeug, für das die Zeitberechnung durchgeführt wird, eine Kommissionierstation verlässt und endet nachdem das Fahrzeug seine Berechnungen für dieses Programm durchgeführt hat. Der Auktionsansatz startet die Zeit mit der ersten Auktion an dem das Fahrzeug, für das die Zeitberechnung durchgeführt wird, teilnimmt und endet nachdem das Fahrzeug eine Auktion gewonnen hat. Die Zeiten wurden bei beiden Ansätzen jeweils 20 mal ermittelt und anschließend die durchschnittlichen Zeiten berechnet. Dabei benötigt der Auktionsansatz durchschnittlich 9005 Millisekunden für die Bestimmung des nächsten Jobs für ein Fahrzeug. Der AWM Ansatz benötigt für seine Berechnungen 9956 Millisekunden. Dabei ist zu beachten, dass das AWM Programm für das *Grounding* durchschnittlich 8 Millisekunden benötigt. Das *Solving* wird sogar in einer nicht messbaren Zeit erledigt. Dabei wird nicht messbar hier als kleiner oder gleich einer Millisekunde definiert.

Fazit

Der AWM Ansatz hat in allen Experimenten die Menge der Aufträge schneller abgearbeitet als der Auktionsansatz. Dabei ist jedoch zu beachten, dass mit steigender Anzahl der Fahrzeuge die Zeitunterschiede zwischen den Ansätzen stark abnehmen. Zudem können zu viele Fahrzeuge und zu kleine Weggraphen schnell zu einer Verschlechterung der Abarbeitungszeiten führen. Die beiden Ansätze benötigen durchschnittlich circa dieselbe Zeit zur Ermittlung eines neuen Jobs und Platzes. Der AWM Solver berechnet seine Lösung schnell, jedoch wird durch die Architektur und das verwendete Framework deutlich mehr Zeit für die Berechnung benötigt.

4.7.2 Deadlock-Erkennung

Im Folgenden wird für die Aufgabe *Deadlock-Erkennung* (siehe Abschnitt 4.2) die Implementierung und Evaluierung vorgestellt.

4.7.2.1 Implementierung

Die Aufgabe *Deadlock-Erkennung* wurde mit Hilfe des Frameworks aus der Bachelorarbeit [13] umgesetzt. Dabei sollen im Folgenden die grundlegenden Schritte der Umsetzung erläutert werden. Für die Umsetzung wurde auf die entwickelte Systemarchitektur aus Abschnitt 3.1 zurückgegriffen. Dabei wird wie in der Beschreibung zur Implementierung der Aufgabe *Rückkehr* (siehe Abschnitt 4.7.1.1) eine Klasse angelegt, welche die Klasse $KrProgramm$ erweitert. In dieser Klasse wurden die zwei abstrakten Methoden $config()$ und $interpretiere()$ implementiert. In der Methode $config()$ wurden die Informationen für die Eingabedaten angegeben, die der *Lokalisator* Agent zum Ausführen der Aufgabe *Deadlock-Erkennung* benötigt. Außerdem wurde zu jeder dieser Information der Agententypen, der diese Information bereitstellt, angegeben. In der Methode $interpretiere()$ wurde eine Schleife angelegt, welche die Lösung elementweise durchläuft und bei Vorkommen einer bestimmten ID den zugehörigen Methodenaufruf durchführt. Zudem wurden die benötigten Aktionen in den jeweiligen Agenten implementiert. Der Rest wird von der Systemarchitektur erledigt. Die Systemarchitektur startet die Ausführung des Prozesses zeitbasiert. Das bedeutet, dass der Prozess zur Berechnung einer Lösung alle zehn Sekunden aufgerufen wird.

4.7.2.2 Evaluierung

Im Folgenden wird die Evaluierung für die Aufgabe *Deadlock-Erkennung* durchgeführt. Dafür wird wie bei der Evaluierung der Aufgabe *Rückkehr* zunächst der experimentelle Aufbau angegeben und anschließend werden die Ergebnisse der Experimente ausgewertet.

Experimenteller Aufbau

Der Experimentelle Aufbau ist fast identisch zu dem Aufbau in Abschnitt 4.7.1.2. In den Experimenten wird das AWM Programm zur Deadlock-Erkennung für zwei verschiedenen Methoden der Job Verteilung evaluiert. Die verwendeten Methoden sind dieselben, wie in dem experimentellen Aufbau aus Abschnitt 4.7.1.2. Im Unterschied zu dem Aufbau wird in den Experimenten nicht die Abarbeitungszeit, sondern die Anzahl der erzeugten Deadlocks ermittelt.

Ergebnisse

Die Simulationsergebnisse sind in Abbildung 4.29 zu finden. Im Folgenden werden die Ergebnisse detaillierter betrachtet. Dabei visualisiert die Abbildung 4.29 die Simulationsergebnisse abhängig von den verwendeten Weggraphen.

Der $Weggraph2$ ist im Vergleich zum $Weggraph1$ deutlich kleiner und provoziert damit stärker Deadlocks. Dies kann an den Ergebnissen in Abbildung 4.29 nachvollzogen werden. Dabei generiert nur der AWM Ansatz Deadlocks. Bei dem bereits implementierten Auktionsansatz, gibt es auch bei zehn Fahrzeugen noch kein Deadlock. Das liegt daran, dass bei dem Auktionsansatz alle Fahrzeuge einer Route folgen, die von dem Regal direkt zu den

Kommissionierstation führt und wieder zurück zum Regal. Bei dem AWM Ansatz kann es dazu kommen, dass ein Fahrzeug von einer Kommissionierstation direkt zu einer weiteren Kommissionierstation fährt und damit den strikten Ablauf der Fahrzeuge durchbricht. Bei dem AWM Ansatz ist unabhängig vom Weggraphen zu erkennen, dass die Anzahl der Deadlocks monoton steigt. Das ist jedoch keine große Überraschung, da mit steigender Anzahl der Fahrzeuge auch das Risiko für Deadlocks steigt.

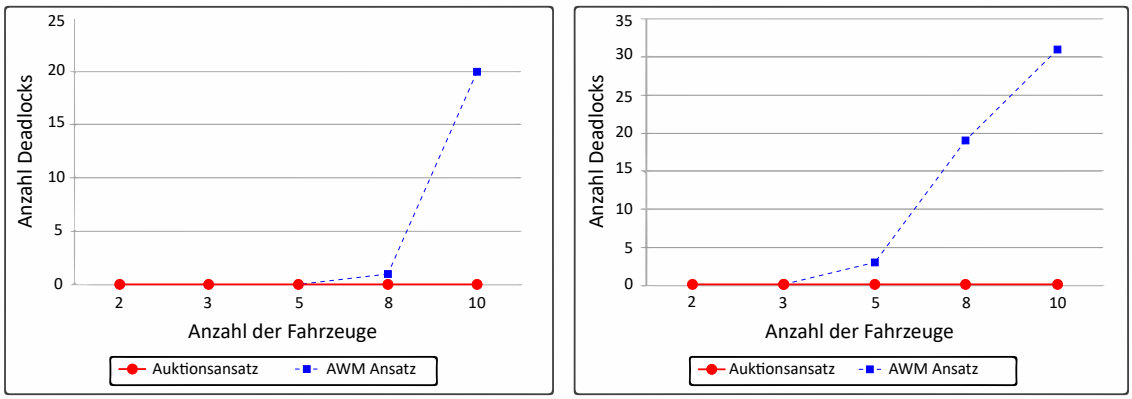


Abbildung 4.29: Die Evaluierungsergebnisse der Aufgabe *Deadlock-Erkennung*. Links die Ergebnisse mit dem Weggraph1 und rechts die Ergebnisse mit dem Weggraph2.

Im Folgenden wird die Zeit für die Berechnung der Deadlock-Erkennung untersucht. Dafür wurde das Framework aus der Bachelorarbeit [13] um eine Zeitmessung für die Aufgabe erweitert. Die Zeitmessung startet, wenn der Lokalisator das Programm ausführt und endet entweder nachdem festgestellt wurde, dass es keinen Deadlock gibt oder nachdem ein am Deadlock beteiligtes Fahrzeug beendet wurde. Die Zeit wurde 20 mal ermittelt und anschließend die durchschnittliche Zeit berechnet. Dabei stellt sich heraus, dass der Ansatz für die Überprüfung und Auflösung eines Deadlocks durchschnittlich 20256 Millisekunden benötigt. Schuld an dieser recht langen Zeit hat das verwendete Framework. Dabei wird der Prozess zur Berechnung häufig in den Hintergrund geschoben und der Agent, welcher den Prozess bearbeitet, erledigt andere Aktionen. Dabei ist zu beachten, dass das AWM Programm zur Erkennung von Deadlocks nur durchschnittlich 8 Millisekunden für das Grounding benötigt. Außerdem wird die Berechnung der Lösung in einer nicht messbaren Zeit erledigt. Dabei wird nicht messbar hier als kleiner oder gleich einer Millisekunde definiert.

Fazit

Zusammenfassend kann festgehalten werden, dass die Anzahl der Deadlocks stark von der gewählten Art der Job Verteilung abhängig ist. Zudem zeigen die Experimente, dass kleinere Weggraphen und mehr Fahrzeuge zu mehr Deadlocks führen. Durch die Experimente wurde deutlich, dass der Auktionsansatz nur in speziellen Situationen Deadlocks erzeugt. Jedoch sollte es auch in diesem Ansatz eine Deadlock-Erkennung geben, da im Falle eines Deadlocks der Betrieb stark eingeschränkt ist. Wenn der AWM Ansatz verwendet wird, muss eine Deadlock-Erkennung und Auflösung existieren, da es bei diesem Ansatz zu einer Vielzahl von Deadlocks kommen kann. Die eigentliche Berechnung des AWM Programms zur Deadlock-Erkennung liefert eine Lösung bereits nach wenigen Millisekunden, jedoch beansprucht der komplette Prozess im Framework deutlich mehr Zeit.

Kapitel 5

Abschließende Bewertung der AWM Implementierungen

In diesem Kapitel soll der Mehrwert der Antwortmengenprogrammierung gegenüber dem bereits bestehenden Ansatz ermittelt werden. Dabei wird der bestehende Ansatz durch das Framework aus der Bachelorarbeit [13] repräsentiert. Durch die Ermittlung des Mehrwerts soll die Frage geklärt werden, ob AWM im ZFT System sinnvoll ist und in welchen Aspekten dieser Mehrwert entsteht. Dafür werden zunächst sowohl die positiven als auch die negativen Eigenschaften der Antwortmengenprogrammierung betrachtet (siehe Abschnitt 5.1). Danach werden einige Punkte vorgestellt, in denen AWM einen Mehrwert gegenüber dem bereits bestehenden Ansatz bietet (siehe Abschnitt 5.2).

5.1 Bewertung der AWM Implementierungen

In dieser Bewertung sollen sowohl die positiven als auch die negativen Eigenschaften der Antwortmengenprogrammierung betrachtet werden. Dabei werden in diesem Abschnitt auch Eigenschaften betrachtet, die nicht im direkten Zusammenhang mit dem ZFT System stehen. Bei den Eigenschaften sind auch einige subjektive Punkte zu finden, die während der Erstellung der Arbeit besonders signifikant waren. In dieser Arbeit wurde der für die Antwortmengenprogrammierung beliebte Solver *clingo* für die Berechnungen von Antwortmengen verwendet. Aus diesem Grund wird die Bewertung der AWM Solver auch anhand von *clingo* durchgeführt.

5.1.1 Positive Eigenschaften der AWM Implementierungen

Die Antwortmengenprogrammierung ist ein Ansatz der deklarativen Programmierung. Dabei weist die deklarative Programmierung gegenüber der imperativen Programmierung mehrere Vorzüge auf. Ein Vorzug ist, dass auch Anwender die keine Programmiererfahrung haben, AWM Programme entwickeln können. Das liegt an der leicht erlernbaren Syntax der Antwortmengenprogrammierung und den intuitiven Aufbau der Regeln [15]. AWM erlaubt zudem die Verwendung von problemspezifischen Wissen und problemspezifischer Sprache, wodurch die Experten ihr Fachwissen leicht modellieren können [12]. Ein weiterer wichtiger Punkt ist, dass die entwickelten AWM Programme oft kürzer sind als vergleichbare impe-

rative Programme, wodurch die Verständlichkeit des Codes zunimmt und die Fehlerrate abnimmt. Aufgrund der geringen Größe der modellierten AWM Programme, können diese leicht verifiziert und kontrolliert werden [23]. Zudem ist die Reihenfolge sowohl im Regelrumpf als auch zwischen unterschiedlichen Regeln in der Antwortmengenprogrammierung nicht relevant [12], was gerade für Anwender ohne Programmiererfahrung eine große Hilfe bei der Modellierung ist. Diese Eigenschaft grenzt AWM stark von *Prolog* ab. Außerdem ist AWM im Gegensatz zu *Prolog* in der Lage, negative Informationen mit Hilfe der strikten Negation zu modellieren [23]. Ein weiterer Vorteil der Antwortmengenprogrammierung ist die differenzierte Modellierbarkeit von Unsicherheit, fehlendem Wissen und definitiven Nichtwissen durch zwei verschiedene Negationsoperatoren [23]. Zusammenfassend ist die schnelle, einfache, mächtige und benutzerfreundliche Modellierung ein Hauptvorteil von AWM.

Ein weiterer Hauptvorteil der Antwortmengenprogrammierung ist die Bereitstellung des effizienten und kostenlosen Solvers *clingo*. Dabei zeichnet sich *clingo* dadurch aus, dass er seine Berechnungen schnell und nachvollziehbar durchführt. Außerdem gibt es für den Solver online gute Literatur und Foren (siehe z.B. [16]). Zudem gibt es einen Youtube Kanal, welcher die Antwortmengenprogrammierung mit *clingo* anschaulich erklärt. In der Literatur von *clingo* sind die verwendeten Methoden und Algorithmen angegeben, die der Solver für die Berechnung der Antwortmengen verwendet (siehe [16]). Das ist ein großer Vorteil, da dadurch der Benutzer die Funktionsweise des Solvers gut nachvollziehen und bei der Modellierung seines Problems berücksichtigen kann. *Clingo* bietet zudem eine Möglichkeit die benötigten Eingabedaten direkt aus einer Datenbank zu laden [16, S.117]. Dies ist ein hilfreiches Feature, da die Eingabedaten oft in einer Datenbank vorliegen. Der Solver *clingo* bietet für die Modellierung eines Problems außerdem Methoden an, welche die Modellierung vereinfachen und effizient gestalten. Dabei gibt es zum Beispiel Funktionen zur Minimierung eines Kriteriums, das Bilden einer Summe für eine Menge von Werten und das Zählen von Literalen mit bestimmten Eigenschaften [16, S.16-23]. Die Ausgaben von dem Solver sind informativ, dabei erhält der Anwender neben der Lösung unter anderem auch Informationen zu der Berechnungsdauer. Es können zudem ausführliche Statistiken für das Grounding und für das Solving ausgegeben werden [16, S.118/134]. Ein weiterer interessanter Punkt der Antwortmengenprogrammierung ist, dass Heuristiken für die Ermittlung von Lösungen verwendet werden können [16, S.137]. Diese Heuristiken führen in manchen Fällen zu einer deutlichen Performancesteigerung.

Da AWM eine Entwicklung der Wissensrepräsentation ist, ist sie besonders in wissensintensiven und kombinatorischen Anwendungen nützlich [20]. Dabei haben die meisten imperativen Programmiersprachen keine Möglichkeit effizient mit wissensintensiven Anwendungen umzugehen. Eine weitere positive Eigenschaft von AWM ist, dass ein Anwender sich mehrere Lösungen für ein modelliertes Problem ausgeben lassen und diese anschließend bewerten kann. Damit kann ein Experte eine begründete Entscheidung auf Grundlage der berechneten Lösungen treffen. Aus diesem Grund kann AWM auch als ein unterstützendes Werkzeug zur effizienteren Lösungsfindung für Experten betrachtet werden.

5.1.2 Negative Eigenschaften der AWM Implementierungen

Die Antwortmengenprogrammierung weist auch Eigenschaften auf, welche bei der Verwendung negativ auffallen. Diese Eigenschaften werden im Folgenden beschrieben. Zunächst

kann festgestellt werden, dass der Solver *clingo* nicht benutzerfreundlich ist. Gerade für technikunversierte Anwender ist der Umgang mit dem Solver umständlich. Dabei wäre eine Programmierumgebung sinnvoll, welche dem Anwender beim Erstellen der AWM Programme hilft. Ein hilfreiches Feature wäre zum Beispiel die Autovervollständigung von AWM Kommandos. Auch die direkte Ausführung und Ausgabe des modellierten Problems in einer benutzerfreundlichen Programmierumgebung würde die Verbreitung und Akzeptanz von *clingo* weiter steigern.

Aus meiner persönlichen Sicht ist die kritischste Eigenschaft der Antwortmengenprogrammierung, dass die meisten modellierten Programme ohne Feintuning nicht effizient sind und dadurch vergleichsweise deutlich mehr Zeit für ihre Berechnungen benötigen. Diese Tatsache kann anhand des folgenden Beispiels nachvollzogen werden. In dem Beispiel (siehe [16, S.153]) wurde ein intuitives und ein performantes AWM Encoding für das n-Queens Problem entwickelt und die Berechnungszeiten für das 50- und 100- Queens Problem verglichen. In der Tabelle 5.1 sind die Berechnungszeiten aufgelistet.

Encoding Typ	**Anzahl (n)**	**Berechnungszeit in Sekunden**
intuitiv	50	5,42
performant	50	0,05
intuitiv	100	>300
performant	100	0,1

Tabelle 5.1: Berechnungszeiten n-Queens Problem

Für das Feintuning benötigt der Benutzer jedoch weiterführende AWM und *clingo* Kenntnisse. Das verfehlt dann den Einsatzzweck, dass auch technikunversierte Anwender ihre Probleme selbst modellieren sollen. Neben dem Feintuning führt eine große Anzahl von Eingaben in den meisten Fällen zu einer großen Berechnungsdauer, was auch bei der Erstellung der Encodings für die Aufgaben aus dem Kapitel 4 beobachtet werden konnte. Diese Eigenschaft ist gerade in dem Aspekt der Echtzeitfähigkeit im Rahmen der Industrie 4.0 kritisch zu betrachten. Des Weiteren fehlen der Antwortmengenprogrammierung hilfreiche Datenstrukturen wie zum Beispiel Listen, welche nur mit Hilfe einer aufwendigen Behelfslösung erreicht werden können. Gerade bei der Modellierung der Aufgabe *Minimiere Strafen* (siehe Abschnitt 4.3) wurde das Fehlen der Listen in AWM bewusst. Ein weiterer negativer Aspekt ist, dass die Ermittlung von geeigneten AWM Aufgaben nicht trivial ist, da viele Aufgaben mit Hilfe von effizienteren Methoden gelöst werden können. Auch wenn der AWM Ansatz viele Aufgaben lösen kann, sollten die Benutzer beachten, dass sie nach effizienteren Methoden suchen oder diese selbst entwickeln sollten. Ansonsten wird direkt auf die einfache AWM Modellierung zurückgegriffen, ohne sich Gedanken über eine effizientere Methode zu machen.

5.2 Mehrwert der AWM Implementierungen gegenüber dem bestehenden Ansatz

Da es sich bei der Antwortmengenprogrammierung um einen Ansatz der deklarativen Programmierung handelt, ergeben sich für den Anwender viele Vorteile. Ein Vorteil ist, dass

die Programmierung mit AWM leicht zu erlernen ist und die Programme somit von den Experten selbst geschrieben werden können. Damit kann das Expertenwissen direkt umgesetzt werden. Außerdem sind die entwickelten AWM Programme oft kürzer als vergleichbare Programme für den bestehenden Ansatz, wodurch die Verständlichkeit des Codes zunimmt und die Fehlerrate abnimmt. Ein weiterer Mehrwert von AWM ist, dass die entwickelten Programme leicht verifiziert und kontrolliert werden können [23]. Das liegt an der vergleichsweise leicht verständlichen Syntax und den daraus folgenden intuitiven Regeln.

Einen weiteren großen Mehrwert liefert der Umgang mit unvollständigen und unsicheren Informationen. Denn die Antwortmengenprogrammierung kann im Vergleich zu dem bestehenden Ansatz gut mit dieser Art von Informationen umgehen. Dabei bietet AWM einfache Möglichkeiten für die Modellierung von Unsicherheiten und Ausnahmeregeln. Außerdem findet die Antwortmengenprogrammierung ihre Anwendung in komplexen Aufgaben, die Expertenwissen benötigen [23, S.10]. Der aktuelle Ansatz bietet für die Umsetzung dieser Art von Aufgaben keine effiziente Möglichkeit. Jedoch haben viele Planungsaufgaben, die im ZFT System benötigt werden, eine hohe Komplexität und können damit effizient mit AWM gelöst werden. Auch die Verwendung der reaktiven Antwortmengenprogrammierung bietet einen Mehrwert gegenüber dem bestehenden System. Durch diesen Ansatz kann eine Echtzeitfähigkeit des Systems erreicht werden, welche in der Industrie 4.0 zwingend erforderlich ist [6]. Dabei kann festgestellt werden, dass der aktuelle Ansatz keine Option für die Echtzeitfähigkeit bietet.

Kapitel 6

Fazit und Ausblick

Die Antwortmengenprogrammierung wurde bereits konzeptionell für ausgewählte Planungsaufgaben in zellularen Fördertechnik Systemen angewendet (siehe [24]). Jedoch besteht weiterhin die offene Fragestellung bezüglich der Identifikation von weiteren geeigneten Planungsaufgaben für AWM im ZFT System. Für die Beantwortung dieser Fragestellung konnten in dieser Arbeit erfolgreich fünf geeignete Aufgaben für ZFT Systeme identifiziert werden, die mit Hilfe der Antwortmengenprogrammierung gelöst wurden. Für die identifizierten Aufgaben wurden effiziente AWM Encodings erstellt, welche anschließend ausführlich validiert wurden. Zudem wurden in dieser Arbeit zwei der identifizierten Aufgaben mit Hilfe des Frameworks aus der Bachelorarbeit [13] umgesetzt. Die beiden Aufgaben wurden mit Hilfe einer Nutzwertanalyse, zur rationalen Entscheidungshilfe, ermittelt. Für diese beiden ausgewählten Aufgaben wurden Evaluierungen durchgeführt, bei denen der AWM Ansatz mit dem bereits bestehenden Ansatz auf Grund eines festgelegten Kriteriums verglichen wurde. Dabei konnten für die AWM Ansätze jeweils positive Ergebnisse bei den Evaluierungen festgestellt werden. Eine weitere offene Fragestellung, die in dieser Arbeit beantwortet wurde, ist die Integration von AWM in die Architektur von zellularen Fördersystemen. Dafür wurde in dieser Arbeit eine Systemarchitektur zur Verwendung von Wissensrepräsentation innerhalb eines Multiagentensystems entwickelt, umgesetzt und verifiziert. Diese Architektur ermöglicht die Verwendung unterschiedlicher Implementationen aus dem Bereich der Wissensrepräsentation. Außerdem wurde der Mehrwert der Antwortmengenprogrammierung gegenüber dem bereits bestehenden Ansatz ermittelt. Dabei konnten mehrere signifikante Vorteile für die Antwortmengenprogrammierung identifiziert werden.

Es kann festgestellt werden, dass AWM ein interessanter und geeigneter Ansatz für ein ZFT System ist. Da in dieser Arbeit aus zeitlichen Gründen nicht alle identifizierten Aufgaben umgesetzt und evaluiert werden konnten, wäre es interessant die noch nicht umgesetzten Aufgaben dieser Arbeit umzusetzen und anschließend zu evaluieren, sodass die theoretisch identifizierten Vorteile der Aufgaben in der Praxis überprüft werden können. Eine weitere interessante Aufgabe wäre der Vergleich einer identifizierten Aufgabe mit unterschiedlichen Ansätzen aus dem Bereich der Wissensrepräsentation aufgrund eines bestimmten Kriteriums, wie zum Beispiel der Berechnungszeit. Eine solche Aufgabe wäre durch die entwickelte Systemarchitektur leicht möglich.

Abbildungsverzeichnis

Abkürzungsverzeichnis

ID	Eindeutige Identifikation
IML	Fraunhofer-Institut für Materialfluss und Logistik
JADE	Java Agent Development Framework
KI	Künstliche Intelligenz
KR	Knowledge representation
UML	Unified Modeling Language
ZFT	Zellulare Fördertechnik

Literaturverzeichnis

[1] *Deadlock (Informatik).* https://de.wikipedia.org/wiki/Deadlock_(Informatik), 2017. Zugriff: 13.07.2017.

[2] *Fundierte Entscheidungen treffen mit der Nutzwertanalyse.* https://projekte-leicht-gemacht.de/blog/pm-methoden-erklaert/nutzwertanalyse/, 2017. Zugriff: 14.08.2017.

[3] *Glossar zur Intralogistik.* http://intralogistik.tips/glossar-zur-intralogistik/, 2017. Zugriff: 16.10.2017.

[4] *Online ASP.* https://github.com/grote/Online-ASP/blob/master/doc/guide/restrictions.tex, 2017. Zugriff: 27.06.2017.

[5] *Systemarchitektur.* http://www.ias.uni-stuttgart.de/common/vhb/html/15_43.htm, 2017. Zugriff: 07.08.2017.

[6] *Was ist Industrie 4.0?* https://www.weingarten.ihk.de/innovation/Industrie_4_0/Was_ist_Industrie_4_0/1942100, 2017. Zugriff: 30.08.2017.

[7] *Was ist lua?* https://www.lua.org/about.html, 2017. Zugriff: 05.06.2017.

[8] *Zellulare Transportsysteme.* https://www.iml.fraunhofer.de/de/abteilungen/b1/maschinen_und_anlagen/Forschungsprojekte/Multishuttle_Move_Fahrerloses_Transportsystem.html, 2017. Zugriff: 20.04.2017.

[9] C. Beierle and G. Kern-Isberner, *Methoden wissensbasierter Systeme*, Springer Vieweg, 5 ed., 2014.

[10] F. Bellifemine, F. Bergenti, et al., *Multi-Agent Programming: Languages, Platforms and Applications*, Springer US, Boston, MA, 2005.

[11] P. Blackburn, J. Bos, et al., *Learn Prolog Now!*, vol. 7 of Texts in Computing, College Publications, 2006.

[12] G. Brewka, T. Eiter, et al., *Answer set programming at a glance*, Commun. ACM, 54 (2011), pp. 92–103.

[13] J. Chojnicki, *Basis-Modellierung und Implementierung eines Multiagentensystems autonomer Fahrzeuge in der Logistik*, Bachelorarbeit, Technische Universität Dortmund, 2016.

[14] E. G. Coffman, M. Elphick, et al., *System deadlocks*, ACM Comput. Surv., 3 (1971), pp. 67–78.

[15] A. Dovier, A. Formisano, et al., *An empirical study of constraint logic programming and answer set programming solutions of combinatorial problems*, Journal of Experimental & Theoretical Artificial Intelligence, 21 (2009), pp. 79–121.

[16] M. Gebser, R. Kaminski, et al., *Answer Set Solving in Practice*, Morgan and Claypool Publishers, 2012.

[17] ——, *Clingo = ASP + control: Preliminary report*, CoRR, abs/1405.3694 (2014).

[18] G. A. Kaminka, M. Fox, et al., *ECAI 2016 - 22nd European Conference on Artificial Intelligence*, vol. 285 of Frontiers in Artificial Intelligence and Applications, IOS Press, 2016.

[19] T. Kirks, J. Stenzel, et al., *Cellular transport vehicles for flexible and changeable facility logistic systems*, in Logistics Journal: Proceedings, 2012.

[20] V. Lifschitz, *What is answer set programming?*, in Proceedings of the 23rd National Conference on Artificial Intelligence, vol. 3 of AAAI'08, Chicago, Illinois, 2008, AAAI Press, pp. 1594–1597.

[21] E. Roitzsch, *Analytische Softwarequalitätssicherung in Theorie und Praxis*, Edition Octopus, Verlag-Haus Monsenstein und Vannerdat, 2005.

[22] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd ed., 2009.

[23] S. Schieweck, G. Kern-Isberner, et al., *Intralogistik als anwendungsgebiet der antwortmengenprogrammierung - potentialanalyse*, May 2016.

[24] ——, *Using answer set programming in an order-picking system with cellular transport vehicles*, Proceedings of the IEEE International Conference on Engineering and Engineering Management, 2016.

[25] R. Silhavy, P. Silhavy, et al., *Applied Computational Intelligence and Mathematical Methods: Computational Methods in Systems and Software 2017*, no. Bd. 2 in Advances in Intelligent Systems and Computing, Springer International Publishing, 2017.

[26] G. Westermann, *Kosten-Nutzen-Analyse: Einführung und Fallstudien*, ESV basics, Schmidt, Erich, 2012.

[27] M. Wooldridge, *An Introduction to MultiAgent Systems*, John Wiley and Sons Ltd, Chichester, UK, 2nd ed., 2009.

In mehreren Abbildungen wurden Bilder mit Copyright verwendet. Im Folgenden wird für jedes Bild der Quellnachweis sowie die Abbildungen, in welchen das Bild vorkommt, angegeben.

[Bild1] *Bundesvereinigung Logistik e. V.*: `https://www.bvl.de/en/duisburg/press-centre/pressemitteilung-lesen?pid=397&global_event_id=1`. Zugriff: 14.09.17

[Bild2] *Bundesvereinigung Logistik e. V.*: https://www.bvl.de/en/duisburg/press-centre/pressemitteilung-lesen?pid=397&global_event_id=1. Zugriff: 14.09.17

[Bild3] *Pro Guard*: https://proguardcoatings.com/find-a-distributor/. Zugriff: 14.09.17

[Bild4] *Aha-Soft*: https://www.iconfinder.com/icons/336049/link_management_meeting_network_structure_people_contacts_relations_social_graph_user_group_icon. Zugriff: 14.09.17

- Bild1 : Abbildung 3.4 - 3.6, 3.9, 4.1, 4.7 - 4.11, 4.17 - 4.25
- Bild2 : Abbildung 3.5 - 3.6
- Bild3 : Abbildung 3.5 - 3.6
- Bild4 : Abbildung 3.4