Projektgruppe

Intelligentes Kommissionieren: Evolution und Adaption

Endbericht

9. November 2006

Veranstalter
Lehrstuhl 5, Universität Dortmund
Firma IKEA

Betreuer
Prof. Dr. Bernhard Steffen
Dipl. Inform. Ralf Nagel
Dipl. Inform. Sven Jörges

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Der schwedische Möbelhersteller IKEA plant den Neubau eines Lagers in Dortmund-Mengede. Dieses Lager soll europaweit die Möbelhäuser mit Artikeln versorgen, die diese nicht standardmäßig vorrätig haben. Diese Artikel werden nur in kleineren Stückzahlen – nicht palettenweise – benötigt und sollen so auch angeliefert werden. Da diese Produkte vom Hersteller allerdings palettenweise in Dortmund eintreffen werden, müssen sie dort für die einzelnen Möbelhäuser den Lieferungen entsprechend zusammengestellt werden.

Das Zusammenstellen von Waren aus einem bereitgestellten Sortiment nach vorgegebenen Aufträgen wird in der Logistik *Kommissionieren* genannt. In einem Teil des neuen IKEA-Lagers ist eine automatisierte Kommissionierzone geplant, in der das Prinzip der *Ware-zum-Mann*-Kommissionierung angewandt wird. Nach diesem Prinzip werden die Waren zu den Kommissionierern transportiert. Die Kommissionierer haben die Aufgabe, die Artikel entsprechend der Aufträge von den ankommenden Paletten auf sogenannte *Zielpaletten* zu verteilen. Diese Zielpaletten werden dann an die einzelnen Möbelhäuser geliefert. Der genaue Aufbau des Lagers und der Ablauf der Kommissionierung ist in Kapitel 1.2 beschrieben.

Die Besonderheit an diesem neuen Lager ist die Art der Kommissionierung. Der Großteil der Aufgaben wird automatisiert erledigt, lediglich das Umpacken der Waren bleibt den Kommissionierern überlassen. Mit der Modellierung und Simulation der Kommissionierzone hatte die Projektgruppe eine Aufgabe, die es den Teilnehmern ermöglichte, an einem Projekt mitzuwirken, das von einer Firma so auch verwirklicht wird. IKEA übernahm hierbei die Rolle des „Kunden“ und stellte Mitarbeiter als Ansprechpartner zur Verfügung.

1.2 Aufbau und Ablauf

Abbildung 1.1 zeigt einen Ausschnitt aus dem Grundriss des neuen IKEA-Lagers. Am unteren Rand der Abbildung befindet sich das *Hochregallager* (1). In diesen Regalen befinden sich die Paletten, die mit Hilfe von *Regalbediengeräten* entnommen und in die Kommissionierzone übergeben werden. Insgesamt arbeiten 14 vollautomatische Regalbediengeräte in dem Lager, die jeweils zwei Regale bedienen. Jedes dieser Geräte ist verfügbar über eine Ent- und eine Beladezone, die als Schnittstelle

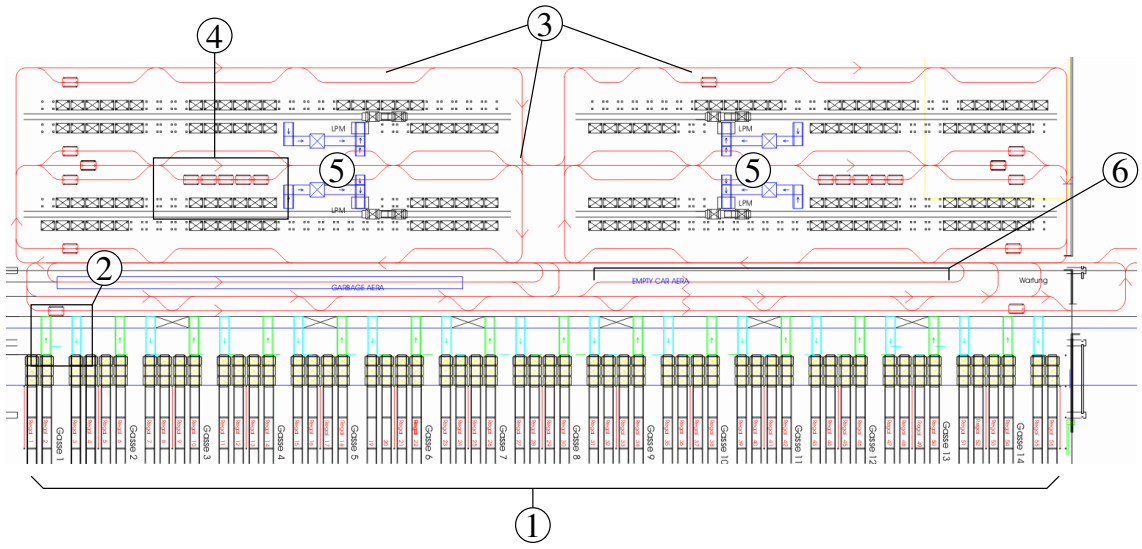


Abbildung 1.1: Die Kommissionierzone.

zur Kommissionierzone dienen. Die Zonen befinden sich jeweils am Anfang einer Gasse (2). An die Ladezonen angeschlossen ist die Kommissionierzone, bestehend aus einem automatisierten *Elektrohängebahnsystem* (3) und 24 Bahnhöfen (4). Abbildung 1.2 zeigt einen der Bahnhöfe im Detail.

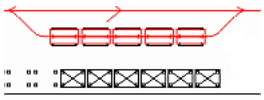


Abbildung 1.2: Ein Bahnhof der Kommissionierzone.

Die Fahrzeuge, die automatisch durch dieses Schienennetz gesteuert werden, können jeweils eine Palette transportieren. Es gibt drei Schienenstränge, in der Zeichnung in horizontaler Richtung dargestellt, an denen die Bahnhöfe angeordnet sind. Am oberen und unteren Schienenstrang befinden sich jeweils sechs Bahnhöfe, am mittleren zwölf, so dass insgesamt 24 Bahnhöfe zur Verfügung stehen. Zusätzlich zu den Bahnhöfen gibt es vier *Leerpalettenmagazine* (5), an denen leere Paletten von den Fahrzeugen entladen werden können. Diese Paletten werden später an den Bahnhöfen wieder verwendet. Bei dem Schienensystem der Elektrohängebahn handelt es sich um ein Einbahnstraßensystem, dessen Fahrtrichung durch die kleinen Pfeile in der Zeichnung vorgegeben ist. Die Fahrzeuge dürfen niemals (auch nicht zum „Ausparken“) die Richtung wechseln. Zunächst sind 120 Fahrzeuge für den Betrieb geplant, diese Zahl kann jedoch noch auf bis zu 180 erhöht werden. Die Anzahl der sich im Einsatz befindenden Fahrzeuge hängt von der Auslastung der Kommissionierzone ab. Nicht benötigte Fahrzeuge können in der sogenannten *empty car area*

(6) abgestellt werden. Die gesamte Kommissionierzone misst 110 mal 24 Meter und befindet sich auf der zweiten Ebene in ca. acht Metern Höhe.

In jedem Bahnhof arbeitet ein Kommissionierer, der für das Umpacken der Waren von den Fahrzeugen auf die *Zielpaletten* zuständig ist. Zusätzlich zu den 24 Kommissionierern gibt es drei sogenannte *Springer*, die jeweils dort einspringen können, wo ein Kommissionierer ausfällt. In einem Bahnhof haben bis zu fünf Fahrzeuge Platz. Von diesen kann der Kommissionierer jeweils das erste feststellen und die Waren entsprechend der Aufträge zusammenstellen. Die Artikel werden von ihm auf die sechs Zielpaletten gepackt. Lampen zeigen ihm dabei an, wohin er welche Waren packen muss. Nachdem eine Zielpalette fertig bepackt ist, wird diese über einen Aufzug automatisch auf die untere Ebene des Lagers befördert. Dort kann sie dann mit einem Stapler zum entsprechenden LKW-Verladeplatz gebracht werden. Eine leere Palette aus dem Leerpalettenmagazin ersetzt die volle Palette am Bahnhof.

Das Lager soll alle 200 Möbelhäuser in Europa beliefern. Die eingehenden Aufträge werden im Zentrallager gesammelt. Es folgt eine Aufteilung in *Batches*, welche die Order von 40 bis 80 Möbelhäusern enthalten. Pro Arbeitstag von 16 Stunden werden voraussichtlich zwei bis vier Batches bearbeitet. Die Verteilung der Aufträge in Batches geschieht offline, so dass später eintreffende Bestellungen der Möbelhäuser erst in der nächsten Batch berücksichtigt werden. Die Auslastung der einzelnen Kommissionierbereiche soll möglichst gleichmäßig sein.

1.3 Aufgabenstellung

Die Aufgabe der Projektgruppe bestand in der Modellierung und der Simulation der Kommissionierzone dieses IKEA Lagers. Anhand der Simulationsmodelle sollten Strategien für die Kommissionierung gefunden und optimiert werden. Zur Erstellung der Modelle standen zwei Werkzeuge zur Verfügung: *jABC* und Uppaal. In Kapitel 2 werden die beiden Werkzeuge näher vorgestellt. Zunächst sollte von zwei Teilgruppen der Projektgruppe parallel jeweils ein Modell mit jedem der Werkzeuge erstellt werden. Bevor die eigentliche Modellierung beginnen konnte, musste sich jeder Teilnehmer in das entsprechende Werkzeug einarbeiten. Zudem musste das Problem genau analysiert und diskutiert werden. Die Modellierung begann mit der Entwicklung der Teilkomponenten des Systems. Die *jABC*-Teilgruppe hat hierzu sogenannte SIBs erstellt, die die zu simulierenden Teilkomponenten im Lager darstellen. Diese Komponenten können später zu einem Modell der Kommissionierzone zusammengebaut werden. In Uppaal bestehen die einzelnen Komponenten aus zeitgesteuerten Automaten, die zu einem Netzwerk zusammengefasst werden.

Nach der Erstellung der Modelle folgte die Evaluierung und Optimierung verschiedener Kommissionierstrategien. Beide Werkzeuge stellen zu diesem Zweck Simulationsumgebungen zur Verfügung. Bei der Modellierung mit dem *jABC* besteht die Möglichkeit, sich beim Erstellen des Modells an den geographischen Gegebenheiten

der Kommissionierzone zu orientieren. Die einzelnen Komponenten werden als Graph im Baustein-Prinzip mit Kanten verbunden. Die Simulation ist bei entsprechender Positionierung der SIBs in dem Graphen sehr übersichtlich und nachvollziehbar. Problematischer ist die Simulation hingegen bei UPPAAL. Die Simulationsumgebung zeigt lediglich die Zustandsübergänge der einzelnen Automaten an. Somit musste zusätzlich eine geeignete Visualisierung für die Simulation mit UPPAAL entwickelt werden.

Zur Entwicklung und Optimierung der einzelnen Strategien und zur Beantwortung von Fragen bezüglich des Lagers und der Kommissionierzone stand der Projektgruppe ein Ansprechpartner von IKEA zur Verfügung. Die Strategien beider Gruppen wurden miteinander verglichen und diskutiert. Da man mit zwei Werkzeugen arbeitet, mit denen das Problem von zwei unterschiedlichen Seiten beleuchtet wird, werden vermutlich verschiedene Ergebnisse und Strategien zur Diskussion stehen. Da es sich um eine praxisorientierte Projektgruppe handelt, wurde während der Entwicklung der Modelle und der Erarbeitung der Strategien viel Wert auf Stabilität und Präzision gelegt.

Das Hochregallager konnte von der Projektgruppe als „Black Box“ angesehen werden. Man konnte davon ausgehen, dass immer genügend Waren vorhanden sind und die Regalbediengeräte schnell genug arbeiten, um die Kommissionierzone mit Paletten zu versorgen. Das Layout des gesamten Lagers lag bereits fest, so dass sich die Projektgruppe ganz auf die Findung und Optimierung von Strategien zur Steuerung der Kommissionierzone konzentrieren konnte.

2 Werkzeuge

2.1 jABC

Das *jABC* ist das verwendete Modellierungswerkzeug. Es basiert auf der Methode der „Lightweight Process Coordination“ (siehe [JAB]). Dabei werden austauschbare Komponenten, die Service Independent Building Blocks (kurz: SIBs), graphisch zu so genannten Modellen zusammengefügt. Damit ist das jABC nicht auf ein Modellierungsgebiet beschränkt, sondern vielseitig einsetzbar.

2.1.1 Architektur des *jABC*-Gesamtsystems

Der Kern wird durch das *jABC*-Framework gebildet. Es setzt sich aus folgenden, wesentlichen Teilen zusammen:

1. dem *jABC*, der Modellierungsanwendung,
2. einer Java 1.5 kompatiblen Entwicklungsumgebung
3. einer CVS-Projektverwaltung

Abbildung 2.1 zeigt das Architekturbild des *jABC*-Frameworks.

jABC

Das *jABC* ist der Editor, mit welchem die Systeme modelliert werden. Die Konzeption sieht vor, dass ein Anwendungsexperte, also ein Benutzer mit System-Spezialkenntnissen, diese mit dem *jABC* modelliert.

Die Oberfläche besteht aus Java-Swing-Komponenten, damit ist ein plattformübergreifender Einsatz in Zusammenhang mit einer Java-Runtime-Enviroment möglich.

Die Benutzeroberfläche des *jABC* wird in folgende Bestandteile unterteilt:

1. Projekt- und SIB-Browser,
2. Inspektoren und
3. Graphzeichenfläche.

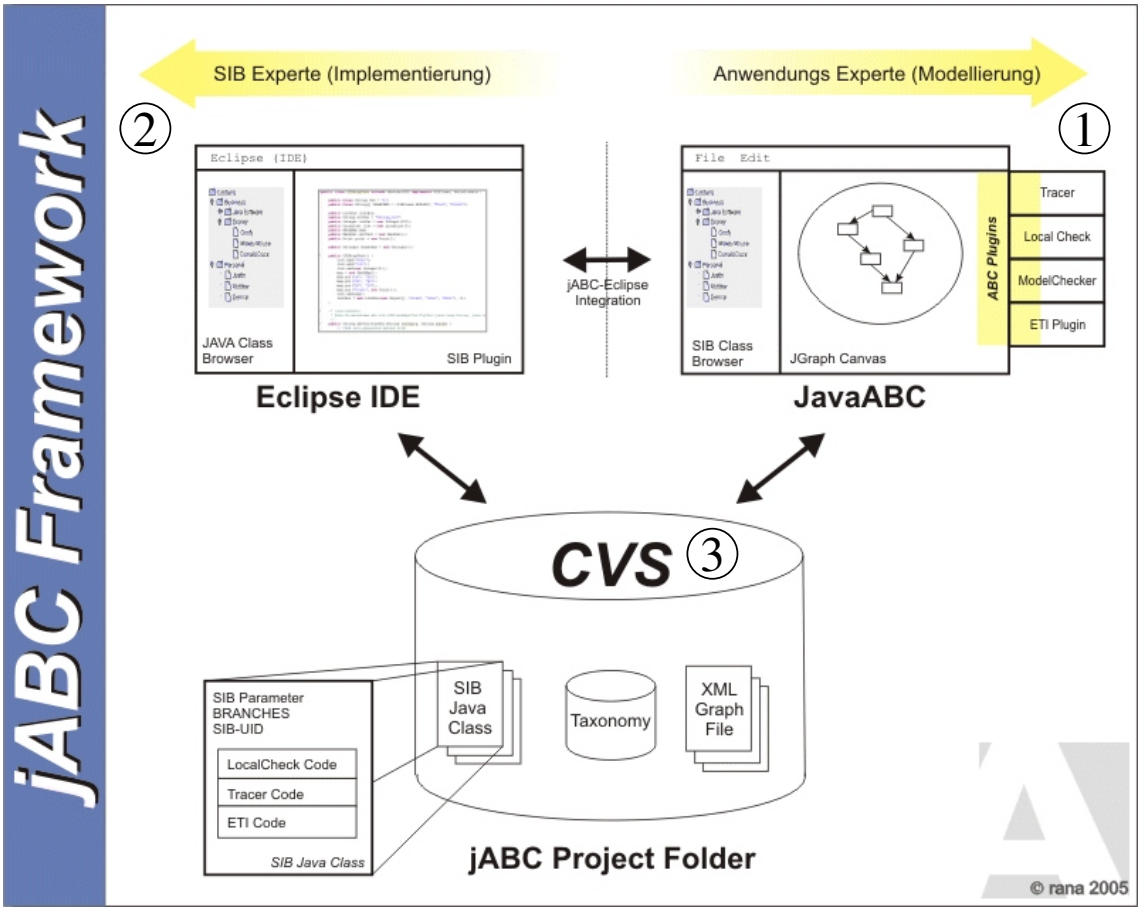


Abbildung 2.1: Architekturbild des *jABC* Frameworks.

Der *Projekt-Browser* zeigt die Liste der definierten *jABC*-Projekte. Jegliche Kontrolle über die Projekte werden in diesem Fenster ausgeführt. Dies sind z.B. Anlegen neuer Projekte, Löschen existierender Projekte, Öffnen eines Projektes etc. Ein Projekt kann einen sogenannten SIB-Pfad enthalten.. Ein Ordner wird zum SIB-Pfad nachdem er aus dem jABC heraus durchsucht wurde und SIB-Java-Klassen in ihm gefunden wurden. Es können beliebig viele SIB-Pfade für ein Projekt definiert werden. Wurden in einem Projekt SIB-Pfade definiert, so werden diese im SIB-Browser angezeigt. Im *SIB-Browser* werden die gefundenen SIB-Java-Klassen entweder als *Taxonomy*- oder als *Filesystem*-Ansicht angezeigt. Die Taxonomiestruktur ist eine vom Filesystem unabhängige Strukturierung der SIBs. Es ist ein Taxonomy-Editor-Plugin erhältlich mit dem diese Strukturierung erzeugt, bearbeitet und abgespeichert werden kann. Ein weiterer SIB-Pfad ist der *Classpath-SIB-Browser*. Dieser zeigt alle SIBs im Java-Klassenpfad an. Dabei handelt es sich in der Regel um die SIBs, welche in den mitgelieferten Plugins enthalten sind. Insbesondere befinden sich dort ein sogenanntes Platzhalter-SIB (Proxy-SIB) und ein SIB welches die Schachtelung

von Graphen ermöglicht (Graph-SIB).

Jede Java-Klasse repräsentiert ein SIB im *jABC*. Ihre Implementierung kann mit jeder Java Entwicklungsumgebung erfolgen, die zu Java 1.5 kompatibel ist. Ist ein SIB implementiert und im *jABC* eingefügt, so kann dieses in der Graphzeichenfläche platziert werden. Die gerichteten Kanten zwischen SIBs können beschriftet werden und so in einen so genannten *Branch* umgewandelt werden. Ein Branch ist damit eine eindeutig identifizierbare Kante.

Der Hauptarbeitsbereich ist die Graphzeichenfläche. Hier werden die Modelle erstellt. Dabei gibt es drei Hauptmodi. Der *change mode* dient zur Änderung eines oder mehrerer SIBs. Der *edge mode* wird zur Bearbeitung von Kanten verwendet. Um Kanten oder SIBs zu löschen wird der *erase mode* verwendet.

Die Inspektoren dienen der Anzeige von Eigenschaften der SIBs oder des Modells. Dies ist ebenfalls während Programmausführungen möglich. Die Standardinspektoren sind der *SIB-Inspector* zum Anzeigen von SIB-Parametern und der *Graph-Inspector* zum Anzeigen und Ändern von Modellparametern.

Das *jABC* lässt sich in seiner Funktionalität durch Plugins erweitern.

Java-Entwicklungsumgebung

Jede Java-Entwicklungsumgebung lässt sich für die Erstellung von SIBs und Plugins verwenden. Es wird jedoch die Entwicklungsplattform Eclipse empfohlen, da diese die Möglichkeit der integrierten CVS-Verwaltung sowie der Integration der *jABC*-Umgebung bietet.

2.1.2 SIBs

Minimales SIB

Ein minimales SIB hat folgendes Aussehen:

```
1  package ikea.sib
2  import de.metaframe.common.sib.SIBClass;
3  public class MinimalSIB implements SIBClass{
4      public final String UID = "c0a80226:2d363530363039343434:1130518276868";
5      public MinimalesSIB() {
6          super();
7      }
8      public String getToolTipText(String arg0, String arg1) {
9          return null;
10     }
11     public Object getIcon() {
12         return null;
13     }
14 }
```

Parameter

Alle als `public` deklarierten Attribute der SIB-Klasse werden nach Außen sichtbare Parameter. Im Folgenden ein Beispiel:

```
1 public class EinParameterSIB implements SIBClass {
2     public final String UID = "c0a80226:2d363530363039343434:1130518276869";
3     public String StringParameter = "StringWert";
4     public Integer IntegerParameter = new Integer(1);
5
6     public EinParameterSIB() {
7         super();
8     }
9 }
```

Dieses SIB enthält die Parameter `StringParameter` für Zeichenketten und `IntegerParameter` für ganze Zahlen. Die Wahl des Parametertyps ist frei, jedoch werden hauptsächlich primitive Datentypen im *jABC* unterstützt. Diese können mit Hilfe von vordefinierten Eingabemasken in den Inspektoren modifiziert werden.

Branches

Branches sind gerichteten Kanten zugewiesene Bezeichner. Sie werden in feste und variable Branches eingeteilt. Der Ausdruck `public final String [] BRANCHES` definiert diese festen Branches. Variable Branches dürfen vom Benutzer geändert werden und sind über `public String[] branches` definiert. Im Folgenden ein Beispiel eines SIBs mit festen und variablen Branches:

```
1 public class EinBranchSIB implements SIBClass{
2     public final String UID ="c0a80226:2d363530363039343434:1130518276870";
3     public final String[]BRANCHES = {"links", "rechts"};
4     public String[] branches = {};
5     public EinBranchSIB() {
6         super();
7     }
8 }
```

Die Kontrolle der Branches erfolgt über die Inspektoren.

Proxy-SIB

Unvollständige, inkompatible oder nicht verfügbare SIB-Klassen werden beim Laden im *jABC* durch das *Proxy-SIB* als Platzhalter dargestellt. Das *Proxy-SIB* wird als ein abgerundetes Rechteck mit einem rot gefärbten Stern im Zentrum dargestellt (Siehe Abbildung 2.2).

Abbildung 2.2: Das ProxySIB

Graph-SIB

Das *Graph-SIB* steht repräsentativ für einen weiteren Graphen. Damit lassen sich Graphen ineinander hierarchisch schachteln (Siehe Abbildung 2.3).

Abbildung 2.3: Das GraphIB

2.1.3 Plugins

Durch Plugins kann der Funktionsumfang des *jABC* erweitert werden. Verschiedene Plugins sind zur Zeit verfügbar, das neu erstellte IKEA-Plugin ist ein Teil des Ergebnisses dieser Projektgruppe.

2.1.4 Einsatzgebiete

Aktuelle Einsatzgebiete des *jABC* sind z. B.:

1. Entwicklung von Webanwendungen,
2. Laufzeitumgebung für ausführbare Modelle,
3. Beschreibung und Ausführung von Workflows,
4. Modellverifikation mit Temporallogiken.

2.1.5 jABC-Website

Als nahe Vorlage für diesen Text dienten die Texte der Webseite des *jABC*-Entwicklerteams (siehe [JAB]). Sie enthält weiterführende Informationen sowie Neuigkeiten in der Weiterentwicklung des Werkzeugs.

2.2 Uppaal

2.2.1 Theoretische Grundlagen

Die in Uppaal benutzten zeitgesteuerten Automaten basieren auf den *Timed Safety Automata*. Die dazu folgenden Definitionen sind dem Uppaal-Tutorial ([Beh04], Seite 2f.) entnommen und ins Deutsche übersetzt worden.

Sei C die Menge der Uhren und $B(C)$ die Menge der Konjunktionen von einfachen Bedingungen der Form $x \bowtie c$ oder $x - y \bowtie c$, wobei $x, y \in C$, $c \in \mathbb{N}$ und $\bowtie \in \{<, \leq, =, \geq, >\}$ sei. Ein zeitgesteuerter Automat ist ein endlicher gerichteter Graph, der mit Bedingungen und Zurücksetzungen über positiv reellwertigen Uhren beschriftet ist.

Definition 1 (Zeitgesteuerter Automat) *Ein zeitgesteuerter Automat ist ein Tupel (L, l_0, C, A, E, I). Dabei ist L die Menge der Orte, $l_0 \in L$ der Anfangsort, C die Menge der Uhren und A die Menge der Aktionen, Nebenaktionen und der internen τ-Aktion. $E \subseteq L \times A \times B(C) \times 2^C \times L$ ist eine Menge von Kanten zwischen Orten mit einer Aktion, einer Bedingung und einer Menge von zurückzusetzenden Uhren. Die Abbildung $I : L \to B(C)$ weist den Orten Invarianten zu.*

In der folgenden Definition sei die Wertbelegung u für die Uhren definiert als Funktion $u : C \to \mathbb{R}_{\geq 0}$. Die Schreibweise $u \in I(l)$ bedeute, dass die Belegung u die zur Stelle l zugehörige Invariante $I(l)$ erfüllt.

Definition 2 (Semantik zeitgesteuerter Automaten) *Sei (L, l_0, C, A, E, I) ein zeitgesteuerter Automat. Seine Semantik ist ein beschriftetes Zustandsübergangssystem $\langle S, s_0, \to \rangle$, wobei $S \subseteq L \times \mathbb{R}$ die Menge der Zustände, $s_0 = (l_0, u_0)$ der Anfangszustand und $\to \subseteq S \times \{\mathbb{R}_{\geq 0} \cup A\} \times S$ die Zustandsübergangsrelation ist mit*

- $(l, u) \xrightarrow{d} (l, u + d)$, *wenn* $\forall d' : 0 \leq d' \leq d \Rightarrow u + d' \in I(l)$ *und*
- $(l, u) \xrightarrow{a} (l', u')$, *wenn es ein* $e = (l, a, g, r, l') \in E$ *gibt, so dass* $u \in g$, $u' = [r \mapsto 0]u$ *und* $u' \in I(l')$.

Dabei stellt $u + d$ für ein $d \in \mathbb{R}_{\geq 0}$ die Abbildung dar, die einer Uhr $x \in C$ den Wert $u(x) + d$ zuweist. $[u \mapsto 0]u$ bezeichnet die Uhrenwertbelegung, die jeder Uhr $x \in r$ den Wert 0 zuweist, im übrigen aber mit der Uhrenwertbelegung u übereinstimmt.

Ein *Netzwerk zeitgesteuerter Automaten* besteht aus n zeitgesteuerten Automaten $A_i = (L_i, l_{0,i}, C, A, E_i, I_i), 1 \leq i \leq n$. Ein Ortsvektor hat in diesem Netzwerk die Form $\bar{l} = (l_1, \ldots, l_n)$. Die einzelnen Invariantenfunktionen, die den Orten Invarianten zuweisen, fassen wir zusammen als gemeinsame Funktion $I(\bar{l}) = \bigwedge_i I_i(l_i)$. Für den Vektor, in dem das i-te Element l_i von $\bar{l}$ durch l'_i ersetzt wurde, schreiben wir $\bar{l}[l'_i/l_i]$. Kommen wir damit zur Semantik:

Definition 3 (Semantik eines Netzwerks zeitgesteuerter Automaten) *Sei nun $A_i = (L_i, l_{0,i}, C, A, E_i, I_i)$ ein Netzwerk zeitgesteuerter Automaten. Dessen Semantik ist ein Zustandsübergangssystem $\langle S, s_0, \rightarrow \rangle$, wobei $S = (L_1 \times \cdots \times L_n) \times \mathbb{R}^C$ die Menge der Zustände, $s_0 = (\bar{l}_o, u_0)$ der Anfangszustand und $\rightarrow \subseteq S \times S$ die Zustandsübergangsrelation ist, gegeben durch:*

- $(\bar{l}, u) \rightarrow (\bar{l}, u + d)$, *wenn* $\forall d' : 0 \leq d' \leq d \Rightarrow u + d' \in I(\bar{l})$,
- $(\bar{l}, u) \rightarrow (\bar{l}[l'_i/l_i], u')$, *wenn es einen Übergang* $l_i \xrightarrow{\tau gr} l'_i$ *gibt, so dass* $u \in g$, $u' = [r \mapsto 0]u$ *und* $u' \in I(\bar{l}[l'_i/l_i])$,
- $(\bar{l}, u) \rightarrow (\bar{l}[l'_j/l_j, l'_i/l_i], u')$, *wenn es Zustandsübergänge* $l_i \xrightarrow{c?g_i r_i} l'_i$ *und* $l_j \xrightarrow{c!g_j r_j} l'_j$ *gibt, so dass* $u \in (g_i \wedge g_j)$, $u' = [r_i \cup r_j \mapsto 0]u$ *und* $u' \in I(\bar{l}[l'_j/l_j, l'_i/l_i])$.

2.2.2 Aufbau eines Uppaal-Systems

Ein Uppaal-System entspricht im Wesentlichen einem Netzwerk zeitgesteuerter Automaten. Ein System besteht aus Prozessen, Uhren, Kanälen, Variablen und Konstanten und hat einen definierten Anfangszustand.

Deklarationen

Die Menge der Uhren in Uppaal entspricht der Menge C in der Theorie der zeitgesteuerten Automaten. Ein Kanal `k` in Uppaal ermöglicht die einfachen Synchronisationsausdrücke `k?` und `k!`. Die gesamte Menge der Synchronisationsausdrücke entspricht dabei der Menge der Aktionen A. Konstanten können in Uppaal universeller benutzt werden als in der Theorie, wo sie nur in den Bedingungen der Menge $B(C)$ vorkommen. Zu den Variablen gibt es in der Theorie kein Gegenstück.

Globale und lokale Deklarationen von Uhren, Kanälen, Variablen und Konstanten erfolgen in Uppaal textuell. Die Syntax von *Deklarationen* (siehe Tabelle 2.1) ähnelt dabei der gängiger Programmiersprachen. Wir wollen nun eine Uhr `t`, Ganzzahlvariablen `x` und `y` und einen Kanal `k` deklarieren.

```
clock t;
int x, y := 1;
chan k;
```

Die Variable `y` wird mit dem Wert 1 initialisiert, während Ganzzahlvariablen ohne eine solche Angabe (hier `x`) mit dem Wert 0 initialisiert werden.

Es gibt die Möglichkeit, Felder von Uhren, Ganzzahlvariablen und Kanälen zu deklarieren. Bei Ganzzahlvariablen kann eine Wertbeschränkung mit angegeben werden. Für Kanaldeklarationen können zusätzlich die Präfixe `urgent` und `broadcast`

Deklaration	::=	(*VariablenDekl* | *KonstDekls*)*
KonstDekls	::=	`'const'` *KonstDekl* (`','` *KonstDekl*)* `';'`
KonstDekl	::=	*ID ArrayDekl** [*Initialisierer*]
VariablenDekls	::=	*Typ VariablenDekl* (`','` *VariablenDekl*)* `';'`
VariablenDekl	::=	*ID ArrayDekl** [`':='` *Initialisierer*]
Initialisierer	::=	*Ausdruck*
		| `'{'` *Initialisierer* (`','` *Initialisierer*)* `'}'`
ArrayDekl	::=	`'['` *Ausdruck* `']'`
Typ	::=	*Präfix TypId* [*Bereich*]
Präfix	::=	`'urgent'` | `'broadcast'`
TypId	::=	`'int'` | `'clock'` | `'chan'` | `'bool'`
Bereich	::=	`'['` *Ausdruck* `','` *Ausdruck* `']'`

Tabelle 2.1: Backus-Naur-Form der *Deklaration* (Uppaal-Hilfe [Upp05]). Für die BNF von *Ausdruck.* siehe Tabelle 2.3.

benutzt werden. Im Folgenden sollen beispielhaft ein Feld von Uhren, eine auf Werte von 1 bis 49 beschränkte Ganzzahlvariable, eine zweidimensionale Matrix mit Initialisierung und ein Broadcast-Kanal deklariert werden:

```
clock uhren[3];
int [1,49] lottozahl;
int einheitsmatrix[2][2] := { { 1, 0 }, { 0, 1 } };
broadcast chan tonsignal;
```

Prozesse und Automatenvorlagen

Ein Prozess in Uppaal entspricht einem Automaten A_i in einem Netzwerk zeitgesteuerter Automaten. Ein Prozess ist die Instanz einer Automatenvorlage. Er besteht aus Orten, zu denen Invarianten gehören können, Zustandsübergängen, die mit *Wächter-*, *Synchronisations-* und *Zuweisungsausdrücken* beschriftet sein können, sowie den eben erwähnten lokalen Uhren, Kanälen und Variablen und Konstanten.

Dabei entspricht

- die Menge der Orte eines Prozesses der Menge L_i eines Automaten A_i,
- der Anfangsort dem Ort $l_{0,i}$,
- die Zuschreibung von Invarianten zu Orten der Invariantenzuweisungsfunktion I_i,
- die Menge der Transitionen der Zustandsübergangsmenge E_i,
- die Wächterausdrücke und Invarianten den Elementen der Menge $B(C)$,

- die Synchronisationsausdrücke den Elementen der Menge der Aktionen A und
- die Zuweisungsausdrücke den Elementen der Menge 2^C.

Automatenvorlagen werden in UPPAAL grafisch dargestellt (siehe Abbildung 2.4) und daher auch grafisch „zusammengeklickt“. In der grafischen Darstellung werden Orte durch Kreise repräsentiert. Der Anfangsort ist durch einen zweiten, konzentrischen Kreis markiert. Man erzeugt einen Ort, indem man z. B. mit der mittleren Maustaste auf die freie Vorlagenfläche klickt. Im Kontextmenü eines Ortes lassen sich verschiedene Attribute einstellen. Orte, die auf diese Weise als *Urgent* oder *Committed* markiert

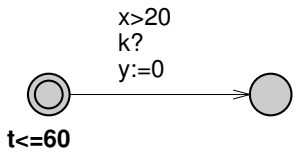


Abbildung 2.4: Ein einfacher Automat, der Teil eines Systems ist.

wurden, tragen zudem die Buchstaben U oder C in ihrem Kreis. Die zu einem Ort gehörenden Invarianten werden als Beschriftung des Kreises angezeigt. Zustandsübergänge werden als Pfeile dargestellt und ihre Attribute direkt an den Pfeil geschrieben. Man erzeugt sie, indem man z. B. mit der mittleren Maustaste zuerst auf den Startort und danach auf den Zielort des zu erzeugenden Zustandsübergangs klickt. Im Kontextmenü des Zustandsübergangs stellt man die Eigenschaften ein. Möchte man, dass bei der Erzeugung eines Prozesses aus einer Vorlage Parameter nötig sind, gibt man einen entsprechenden Ausdruck in ein dafür vorgesehenes Textfeld ein. Es handelt sich dabei um Referenzübergabeparameter. Die Syntax in Backus-Naur-Form befindet sich in Tabelle 2.2.

Parameterliste	::=	[*Parameter* (’;’ *Parameter*)*]
Parameter	::=	*TypId ID ArrayDekl** (’,’ *ID ArrayDekl**)*
		| ’const’ *ID ArrayDekl** (’,’ *ID ArrayDekl**)*

Tabelle 2.2: Backus-Naur-Form des Parameterausdrucks (Uppaal-Hilfe [Upp05]).

Ausdrücke

Die Beschriftungen der Zustandsübergänge und Zustände von Automaten in UPPAAL folgen einer gemeinsamen Syntax, der für *Ausdrücke*. Ihre Backus-Naur-Form kann in Tabelle 2.3 nachgelesen werden.

Ausdruck	::=	ID | NAT |
		Ausdruck `'['` *Ausdruck* `']'` |
		`'('` *Ausdruck* `')'` |
		Ausdruck `'++'` | `'++'` *Ausdruck* |
		Ausdruck `'--'` | `'--'` *Ausdruck* |
		Ausdruck ZuweisungsOp Ausdruck |
		UnärerOp Ausdruck |
		Ausdruck BinärerOp Ausdruck |
		Ausdruck `'?'` *Ausdruck* `':'` *Ausdruck* |
		Ausdruck `'.'` ID |
UnärerOp	::=	`'-'` | `'!'` | `'not'`
BinärerOp	::=	`'<'` | `'<='` | `'=='` | `'!='` | `'>='` | `'>'` |
		`'+'` | `'-'` | `'*'` | `'/'` | `'%'` | `'\&'` |
		`'|'` | `'^'` | `'<<'` | `'>>'` | `'&&'` | `'||'` |
		`'<?'` | `'>?'` | `'and'` | `'or'` | `'imply'`
ZuweisungsOp	::=	`':='` | `'+='` | `'-='` | `'*='` | `'/='` | `'%='` |
		`'|='` | `'='` | `'='` | `'<<='` | `'>>='`

Tabelle 2.3: Backus-Naur-Form von *Ausdruck* (Uppaal-Hilfe [Upp05]).

Im Beispiel von Abbildung 2.4 ist die Beschriftung `t<=60` eine *Invariante*, `x>20` ein *Wächter*, `k?` eine *Synchronisation* und `y:=0` eine *Zuweisung*. Für die vier Arten von Ausdrücken gelten eine Reihe von Einschränkungen (vgl. [Beh04], 4f.):

Wächter: Ein Wächter (Guard) ist ein *Ausdruck*, der folgende Bedingungen erfüllt. Er muss frei sein von Seiteneffekten, darf also den Zustand von Variablen nicht verändern. Dabei muss der Wächter zu einem booleschen Ausdruck ausgewertet werden können. Der Ausdruck darf nur Uhren, Ganzzahlvariablen und Konstanten (oder Felder derselben) referenzieren. Vergleiche von Uhren oder Uhrdifferenzen dürfen nur mit Ganzzahlausdrücken stattfinden. Wächter über Uhren sind notwendigerweise Konjunktionen.

Synchronisation: Eine Synchronisationsbeschriftung ist entweder von der Form `Ausdruck!` oder `Ausdruck?`, oder sie ist leer. Auch hier muss der Ausdruck frei von Seiteneffekten sein. Er muss zu einem Kanal ausgewertet werden können und darf dabei nur Ganzzahlvariablen, Konstanten und Kanäle einbeziehen.

Zuweisung: Eine Zuweisung ist eine kommaseparierte Liste von Ausdrücken mit Seiteneffekten. Die Ausdrücke dürfen nur Uhren, Ganzzahlvariablen und Konstanten referenzieren. Uhren dürfen dabei nur Ganzzahlwerte zugewiesen werden.

Invariante: Eine Invariante ist ein Ausdruck, der folgende Bedingungen erfüllt. Er ist eine Konjunktion von Ungleichungen der Form `x<a` oder `x<=a`, in der nur

Uhren, Ganzzahlvariablen und Konstanten referenziert werden dürfen. Zudem muss der Ausdruck frei von Seiteneffekten sein.

Prozesserzeugung

Die fertigen Automatenvorlagen werden im Bereich *Process assignments* den Prozessen „zugewiesen“. Beispielsweise wird nachfolgend ein Prozess `Prozess` aus der Vorlage `Vorlage` erzeugt und ein Prozess `NochEinProzess` aus der Vorlage `AndereVorlage` unter Angabe zweier Parameter:

```
Prozess := Vorlage();
NochEinProzess := AndereVorlage(3, 4);
```

Die Syntax der Prozesszuweisungsliste kann in Tabelle 2.4 nachgelesen werden.

PZList	::=	*PZ**
PZ	::=	*ID* ':=' *ID* '(' *ArgListe* ');'
ArgListe	::=	[*Ausdruck* (',' *Ausdruck*)*]

Tabelle 2.4: Backus-Naur-Form der *Prozesszuweisungsliste* (Uppaal-Hilfe [Upp05]).

Systemerzeugung

Zum Schluss wird das System definiert, indem auf das Schlüsselwort `system` folgend die zugehörigen Prozesse aufgezählt werden. Die Backus-Naur-Form der Syntax befindet sich in Tabelle 2.5.

Sys	::=	'system' *ID* (',' *ID*)* ';'

Tabelle 2.5: Backus-Naur-Form der *Systemdefinition* (Uppaal-Hilfe [Upp05]).

2.2.3 Semantik eines Uppaal-Systems

Anfangszustand

Alle Variablen haben zu Beginn den angegebenen Initialisierungswert und sonst den Wert 0. Die Anfangswerte von globalen Variablen können von bereits definierten globalen Variablen abhängen oder konstant sein. Ebenso verhält es sich mit Parametern, die Prozessen übergeben werden. Lokale Variablen eines Prozesses können von globalen Variablen, dem Prozess übergebenen Parameterwerten und vorher definierten lokalen Variablen abhängen. Für Array-Elemente gilt das Gleiche wie für Variablen. Jeder Prozess befindet sich in seinem als Anfangszustand markierten Ort. Alle Uhren haben zu Beginn den Wert 0.

Systemzustandsübergänge

Ein System ändert seine Zustände durch Systemzustandsübergänge, die einen, zwei oder mehrere Prozesse mit je einer Prozesstransition beinhalten können, abhängig vom Vorhandensein eines Synchronisationsausdrucks und des dabei benutzten Kanals. Eine Prozesstransition heißt ausführbar, wenn ihr Wächterausdruck erfüllt ist, ihr Zuweisungsausdruck ausführbar ist und der Zielort eine gültige Invariante trägt.

Ein Systemzustandsübergang mit genau einer Prozesstransition ist möglich, wenn diese Transition keinen Synchronisationsausdruck enthält.

Ein Zustandsübergang, der genau zwei Transitionen zweier verschiedener Prozesse umfasst, ist möglich, wenn die eine Transition mit `kanal!` und die andere Transition mit `kanal?` beschriftet ist und beide Prozesstransitionen ansonsten ausführbar sind. Dies nennt sich *binäre Synchronisation*.

Ein Systemzustandsübergang mit einer oder mehreren teilnehmenden Prozesstransitionen ist möglich, wenn genau eine der Prozesstransitionen ausführbar ist, einen Synchronisationsausdruck `kanal!` trägt, `kanal` dabei ein Broadcast-Kanal ist und alle übrigen beteiligten Prozesstransitionen ausführbar sind und einen Synchronisationsausdruck `kanal?` tragen. Jede ausführbare Prozesstransition mit einem Synchronisationsausdruck `kanal?` muss bei einem solchen Systemzustandsübergang mitmachen. Dies nennt sich *Broadcast-Synchronisation*.

Die Auswertung der Zuweisungsausdrücke erfolgt in der Reihenfolge Sender, Empfänger, d. h. der Zuweisungsausdruck der Prozesstransition mit Beschriftung `kanal!` wird zuerst ausgeführt, danach der Zuweisungsausdruck der Transition mit Beschriftung `kanal?`.

Zeit

Die Uhren eines Uppaal-Systems laufen synchron. Ihre Werte werden nicht explizit gespeichert, sondern durch Intervallgrenzen abgesteckt. Ebenso wird für jede Kombination zweier Uhren das Intervall abgespeichert, in dem sich die Differenz ihrer Werte befindet. Zeitfortschritt kann erzwungen werden, indem Wächter- und Invariantenausdrücke so gesetzt werden, dass die Transition, die den Wächterausdruck trägt oder zu dem Zustand mit der Invariante führt, erst nach Ablauf endlicher Zeit ausführbar ist. Ein Systemzustandsübergang mit Zeitfortschritt ist möglich, wenn während der Zeit, die vergehen muss, bis alle Bedingungen erfüllt sind, kein ungültiger Systemzustand eintritt, indem etwa Invarianten anderer Prozesse unerfüllt sind.

Spezielle Prozesszustände

Befinden sich einer oder mehrere Prozesse des Systems in Zuständen, die als *Committed* markiert sind, muss der nächste Systemzustandsübergang eine Transition dieser Prozesse beinhalten. Bei dem Zustandsübergang darf keine Zeit vergehen. Dadurch

lassen sich Abläufe, die mehrere Prozesstransitionen benötigen, leicht atomar gestalten.

Wenn sich ein Prozess des Systems in einem als *Urgent* markierten Zustand befindet, darf ebenfalls keine Zeit verstreichen. Das bedeutet, dass u. a. die Prozesstransitionen, die für ihre Durchführung Zeitfortschritt verlangen, nicht ausführbar sind. Da eine Bevorzugung der Prozesse, die in einem Urgent-Zustand sind, nicht stattfindet, kann man die Urgent-Markierung als Abschwächung der Committed-Markierung betrachten.

Prozesszustände, deren Invarianten nicht erfüllt sind, können als nicht existierend angesehen werden. Ein Prozess, der sich in einem Zustand mit nicht erfüllter Invariante befindet, muss ohne Zeitvergehen einen Zustand mit gültiger Invariante erreichen.

Ein weiterer Sonderfall ist die Synchronisation mit einem Urgent-Kanal. Wenn ein entsprechender Systemzustandsübergang möglich ist, darf keine Zeit vergehen.

Sind von einem Systemzustand aus keine Übergänge möglich, handelt es sich um einen Deadlock-Zustand.

2.2.4 Model-Checking

Der Model-Checker in Uppaal verwendet zur Formulierung der Eigenschaften die 1991 von Rajee Alur definierte Timed Computation Tree Logic (TCTL). Diese erweitert die herkömmliche CTL um die Möglichkeit, das Zeitverhalten von Automaten mittels so genannter *clocks* zu beschreiben. Dafür muss allerdings in Kauf genommen werden, dass nur vier der zehn CTL-Basis-Operatoren unterstützt werden: **AG, AF, EG** und **EF**. Hinzu kommt der neue Operator $\rightarrow$, der formal definiert ist durch $\mathrm{AG}(f \Rightarrow \mathrm{AF}\, g)$ und nötig wird, da Uppaal keine Kombination von Pfadquantoren erlaubt. Hinsichtlich der Zustandsformeln hat Uppaal keine Beschränkungen. Die Syntax dieser an Java und C angelehnten Notation ist in der Tabelle 2.3 erläutert.

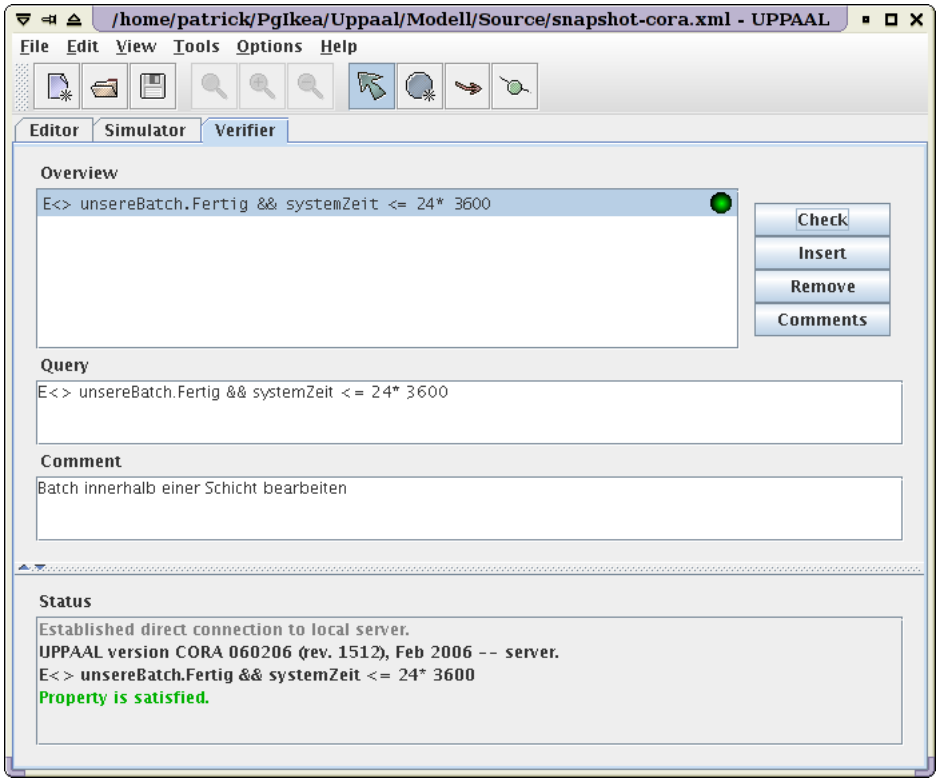


Abbildung 2.5: Der Uppaal-Model-Checker.

Die Eingabe der Anforderungen, *Queries* genannt, erfolgt im Uppaal-Client über den Reiter *Verifier*, der sich in drei Bereiche aufteilen lässt (Abbildung 2.5). Im oberen Teil befindet sich eine Übersicht über die für das System spezifizierten Anfragen, die sich mittels der nebenliegenden Knöpfe *Löschen* und *Überprüfen* lassen. Im mittleren Teil des Reiters lässt sich die in der Übersicht ausgewählte Eigenschaft editieren, wobei zu jeder Formel ein Kommentar erfasst werden kann, der sich anstatt der Formel im Übersichtsteil anzeigen lässt. Zur Kontrolle der Ergebnisse einer Model-Checking-Anfrage dient das Statusfenster im unteren Teil. Schlägt die Verifikation einer Anfrage fehl, so erstellt Uppaal, bei gesetzter Option, einen Trace, der im Simulator angezeigt und analysiert werden kann.

Als Vorteil von Uppaal ist zu nennen, dass der Model-Checker ein separater Prozess ist, der sich sowohl in der Kommandozeile, als auch ausgelagert auf einem leistungsstärkeren Rechner bei Linux-Systemen ausführen lässt. Hierzu müssen zunächst das Modell und die Anfragen abgespeichert werden. Anschließend lässt sich in der Kommandozeile durch den Aufruf

```
$ verifyta [ Optionen ] modell.xml queries.q
```

prüfen, ob die Anforderungen an das Modell erfüllt sind. Mit Hilfe der Optionen lassen sich die Zustandsraumkonstruktion und -reduktion, die Suchreihenfolge sowie

die Art des Gegenbeispiels beeinflussen, was natürlich im Client über die entsprechenden Menüs ebenso möglich ist.

2.2.5 Einsatzmöglichkeiten von Uppaal

Uppaal eignet sich hervorragend für die Modellierung hochparalleler Systeme, da es zeitgesteuerte Automaten einsetzt. Auf diesen Automaten sind dann Model-Checking-Anfragen möglich. Hieraus lassen sich im Wesentlichen sowohl die Vor- als auch Nachteile von Uppaal ableiten: Beim Model-Checking eines Systems werden immer **alle** möglichen Abläufe betrachtet. Durch die Formulierung des Modells und der Anfragen in CTL-Logik können Eigenschaften formal verifiziert bzw. falsifiziert werden. Diese Stärke ist allerdings auch eine Schwäche, denn natürlich ist mit größter Sorgsamkeit darauf zu achten, dass alle Eigenschaften korrekt modelliert werden und dass die Abstraktion mit der Realität übereinstimmt. Auch die Hürde, natürlichsprachliche Anfragen korrekt zu formalisieren, ist zu nehmen.

Ein weiterer bedeutender Nachteil aller benutzten Uppaal-Versionen ist, dass bei Synchronisationen von Prozessen keine Parameter übergeben werden können. Diese sind aber bei der Modellierung des IKEA-Lagers essentiell, denn z. B. bei der Kommunikation von Prozessen, die Schienen modellieren, muss die Identifikationsnummer des einfahrenden Fahrzeuges übergeben werden. Die Lösung des Problems bringt hier ein Trick, der an vielen Stellen Einzug in das Uppaal-Modell erhalten hat: Zwei Prozesse `a` und `b` sollen sich über einen Kanal `chan` synchronisieren. Der rufende Prozess `a` setzt, wenn er die Transition ausführt, die den Synchronisationsausdruck enthält, gleichzeitig einen Parameter `p`. Der lauschende Prozess `b` übernimmt `p`, wenn er seine Synchronisationstransition verlässt. Da beide Prozesse Zugriff auf `p` haben müssen, ist es technisch nicht anders möglich, als dass `p` eine globale Variable ist. Eine Synchronisation ist aber immer atomar. Daher muss nicht darauf geachtet werden, dass kein anderer Prozess `p` zwischenzeitlich überschreibt.

Die Änderungen, die Uppaal im Verlauf unserer Projektgruppe erfahren hat, sollen nun kurz aufgezeigt werden.

Uppaal 3.4

Zu Beginn der Projektgruppe im Oktober 2005 war Uppaal 3.4 die stabile Hauptversion. Für den Betrieb seiner grafischen Oberfläche wird eine Java-1.4-Laufzeitumgebung benötigt. Es fehlt die Möglichkeit, programmiersprachliche Konstrukte in Modellen einsetzen zu können. Wünschenswert wären auch Datenstrukturen (`structs`), Schleifen sowie Zeiger oder Referenzen.

Uppaal 3.6 alpha, beta, CORA

Seit November 2005 gab es erste Alpha-Versionen von Uppaal 3.6, denen ab März 2006 Beta-Versionen folgten. Das in der Projektgruppe zwischenzeitlich benutzte Uppaal CORA basierte jeweils auf diesen Versionen. Für den Betrieb der aufpolierten und nun zu Syntaxhervorhebungen fähigen grafischen Oberfläche ist nun Java 1.5 erforderlich.

Interessante neue Möglichkeiten beim Entwurf von Modellen im Vergleich zur Version 3.4 sind:

- Deklaration von Typen (`typedef`)
- Deklaration von Datensatztypen (`struct`)
- Metavariablen (`meta`)
- Funktionen, For- und While-Schleifen, If-Verzweigung

Die Deklaration von zusammengefassten Typen und die Möglichkeit, einem solchen Strukturtypen durch Typdeklaration einen neuen Namen zu geben, erhöhen die Übersichtlichkeit. Werte von Metavariablen werden bei der Beurteilung, ob zwei Zustände gleich sind, außer Acht gelassen. Beispielsweise haben die Werte von Übergabevariablen nur für den Moment einer Transition Gültigkeit und sind danach unwichtig. Hier kann eine Deklaration zur Metavariablen den Zustandsraum verkleinern. Der Zustandsvektor eines Systems enthält solche Variable jedoch, so dass sich dadurch nicht zusätzlich Speicherplatz einsparen lässt.

Funktionen bieten die Möglichkeit, Programmcode zu kapseln. Die in ihnen möglichen Schleifen und Verzweigungen erlauben komplexe Berechnungen auf beliebig vielen Variablen in nur einem Zustandsübergang. Bisher wäre für eine Schleife ein Ort (*committed*) mit einer Transition zum selben (Schleife) sowie einer Transitionen zu einem Nachfolgeort (Abbruch) nötig gewesen. Jeder Schleifendurchlauf hätte einen Systemzustandsübergang und damit insgesamt hohen Speicherverbrauch bedeutet. Stattdessen wird nun eine Funktion deklariert, die bei Ausführung einer Transition durch deren Update-Teil aufgerufen werden kann.

„Uppaal CORA“ zeichnet sich zusätzlich durch die Verwendung von Kosten aus. Als Grundlage für CORA dienen sogenannte *Linearly Priced Timed Automata* (kurz: *LPTA*), die die im Original verwendeten Timed Safety Automata um eine Kostenvariable erweitern. Diese Variable `cost` wird dabei zu Beginn mit 0 initialisiert und kann in den Zuweisungsausdrücken der Kanten ausschließlich inkrementiert werden. Eine Verwendung von `cost` in Guards ist nicht zulässig, da hierdurch das Model-Checking unentscheidbar würde. Des Weiteren wurde in CORA die Variable `remaining` eingeführt. Diese gibt eine Untergrenze der noch benötigten Kosten an und kann genauso wie `cost` über die Zuweisungsausdrücke der Kanten gesetzt

werden. Beide Variablen ermöglichen ein effizienteres Model-Checking bei Verwendung der neuen Suchreihenfolgen *Best First* und *Random Best Depth First*. Außerdem wird so dem Anwender ermöglicht, dem Model-Checking durch eine geeignete Kostenfunktion eine „Hilfestellung“ zu geben und es damit zu steuern. Nach einigen Experimenten mit Kostenfunktionen, die leider nicht den erhofften Vorteil brachten, konzentrierte sich die Projektgruppe wieder auf die Weiterentwicklung des bis dahin entstandenen Modells und seine Ausführung mit Uppaal 3.6.

Uppaal 4.0

Im Mai 2006 wurde die neue stabile Hauptversion als Uppaal 4.0 (anstatt 3.6) veröffentlicht. Neu sind hier Prioritäten für Kanäle und Prozesse und die zufällige Zuordnung von Werten eines bestimmten Bereiches zu einer Variablen in einer Transition. Mit Hilfe der Priorisierung lassen sich in Systemen, in denen viele gleichartige Prozesse gleichzeitig einen Zustandsübergang vollziehen könnten, die durch die verschiedenen Reihenfolgen entstehenden Ablaufmöglichkeiten eliminieren. Bei unserem Ikea-Modell erhält beispielsweise das erste Fahrzeug eine höhere Priorität als das zweite, so dass zu einer Zeit, in der beide fahren könnten, aufgrund der Priorisierung zunächst das erste Fahrzeug und dann das zweite fährt. Die Möglichkeit, während einer Transition Zufallszahlen zu erhalten, wäre zu Beginn der Projektgruppe sehr hilfreich gewesen. Zufallsentscheidungen wie Zuordnungen von Palettenaufträgen zu Bahnhöfen geschehen derzeit durch die (zufällige) Auswahl gleichberechtigter Prozesse als Synchronisationspartner. Hier hätte vermutlich die Anzahl der Prozesse deutlich eingespart werden können.

3 jABC-Modell

3.1 Spezifikation

3.1.1 Einleitung

Das in Kapitel 2.1 vorgestellte *jABC* bildet die Grundlage für das entwickelte Tool zur Simulation von Kommissionierungsprozessen in dem in Kapitel 1.1 beschriebenen Lager.

Dabei werden die verwendeten SIBs als Strecken- und Funktionselemente eines Elektrohängebahnsystems interpretiert. Diese speziell definierten SIB-Typen sowie Regeln für ihren Einsatzzweck und gegenseitige Kombination erlauben eine mächtige und wirklichkeitsnahe Modellierung.

Die Eingabe der Simulationsdaten und Analyse erfolgt über eigenständige Programme, die über geeignete Schnittstellen mit dem *jABC*-Plugin kommunizieren.

3.1.2 Anforderungen

Ziel

Das unmittelbare Ziel des Programms besteht in der Evaluierung von Zeit- und Mittelkosten (das sind z. B. die eingesetzten EHB-Fahrzeuge, Paletten usw.), die zur Durchführung von *Kommissionierungsprozessen* benötigt werden.

Aufgrund des in der Einleitung beschriebenen Prozessablaufs erwartet das IKEA-Simulationswerkzeug folgende Eingabedaten:

- Ein Schienennetzmodell,
- die *Auftrags-Batch*, also die zu kommissionierenden Paletten,
- die Palettengrößen der einzelnen Artikel sowie
- Parameter der Schienen und Fahrzeuge.

Das *Schienennetz* wird auf der *Zeichenfläche* des *jABC* modelliert. Das IKEA-Plugin definiert alle für die allgemeine Modellierung erforderlichen elementaren *Streckentypen* einschließlich sämtlicher vorkommender *Stationstypen* und Regeln für deren Verwendung. Somit wird dem Benutzer des Tools eine große Modellierungsfreiheit geboten, die sich nicht nur auf das von IKEA vorgegebene Schienennetz be-

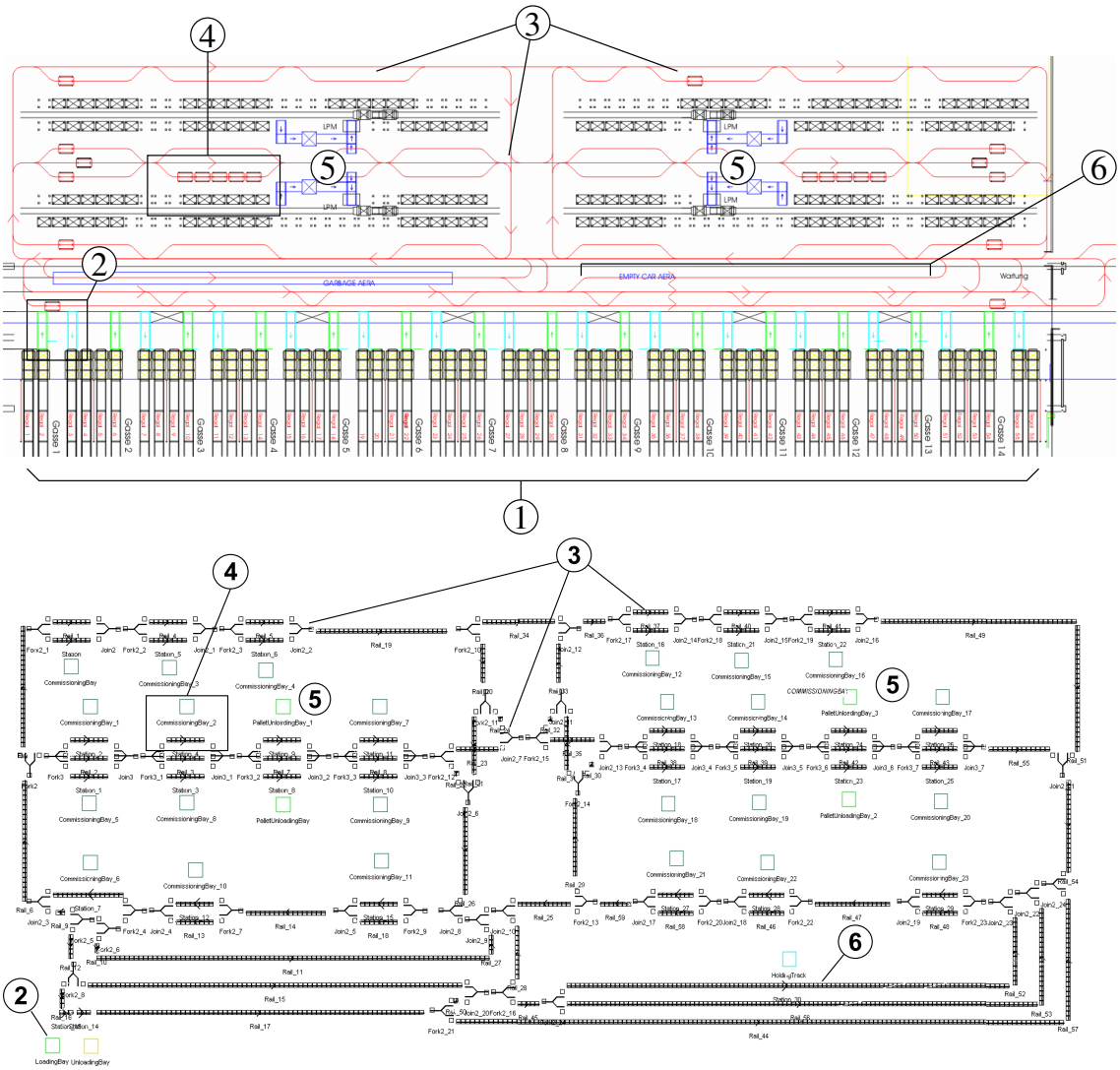


Abbildung 3.1: Das IKEA Lager als CAD Zeichnung und *jABC* Modell

schränkt. Im Rahmen des Projekts wird die Simulation auf ein Modell angewendet, das exakt der von IKEA vorgegeben CAD-Zeichnung des Streckensystems entspricht.

Unter Berücksichtigung der oben spezifizierten Eingabedaten berechnet das IKEA-Simulationswerkzeug eine mögliche Reihenfolge der Kommissionierung der Batch. Dabei werden die Kommissionieraufträge in der gegebenen Reihenfolge, also nach dem *FIFO-Prinzip*, bearbeitet. Aufgrund des modularen Aufbaus des *jABC* kann dieses Prinzip jedoch leicht durch einen alternativen Algorithmus ersetzt werden.

Bei der Simulation selbst kann der Benutzer zwischen der performanteren, rein rechnerischen und der grafischen Simulation wählen. Bei der rechnerischen Simulation erfolgt keine grafische Darstellung des aktuellen Systemzustands. Ergebnisse

ergeben sich hierbei nur durch die Auswertung des erstellen Logs. Bei der grafischen Simulation hingegen wird stets der aktuelle Systemzustand abgebildet, sodass z. B. einzelne Fahrzeugbewegungen über die entsprechenden Inspektoren nachvollzogen werden können. Ein Wechsel zwischen den beiden Simulationsarten ist dabei stets möglich.

Determinismus

Die eingesetzten programminternen Algorithmen arbeiten deterministisch. Das bedeutet, dass anhand gleicher Eingabedaten und unter gleichen Voraussetzungen die gleichen Ergebnisse erzielt werden. Somit ist die Vergleichbarkeit der erzielten Ergebnisse gegeben.

3.1.3 Design-Annahmen

Um die Komplexität eines Großlagers mit Elektrohängebahnnetz möglichst präzise und wirklichkeitsnah abzubilden und gleichzeitig das entwickelte System möglichst einfach und überschaubar zu gestalten, wurden vereinfachende Design-Annahmen getroffen. Diese richten sich nach den tatsächlichen Gegebenheiten des geplanten IKEA-Lagers.

Simulation der statischen Streckenelemente

Das Schienennetzmodell, auf dem das Simulationswerkzeug angewendet wird, entspricht dem physikalischen Streckensystem, das von IKEA vorgegeben wurde. Das gegebene CAD-Modell wurde in elementare Strecken-Grundtypen aufgeteilt. Diese werden von dem entwickelten Simulationswerkzeug durch unterschiedliche SIB-Typen repräsentiert (siehe auch Kapitel 3.2).

Durch die Platzierung dieser speziell definierten SIBs auf der Zeichenfläche des *jABC* kann ein beliebiges Schienennetz modelliert werden. Hierzu erhalten alle Fahrstrecken-SIBs eine Positionierungsausrichtung, die bei der Modellierung auf der Zeichenfläche passend gewählt werden kann. Das sind die vier Richtungen `North`, `South`, `East`, `West`.

Geschwindigkeit

Um *Geschwindigkeiten* in das Simulationswerkzeug zu integrieren, werden vier konstante Geschwindigkeitsstufen unterschieden. Diese werden direkt an die im Modell verwendeten Fahrstrecken-SIBs gekoppelt. Ein EHB-Fahrzeug, welches sich auf einem der Fahrstrecken-SIBs befindet, kann höchstens das schnellste auf dem SIB erlaubte Tempo fahren. Folgende Geschwindigkeitsstufen werden im Simulationswerkzeug zur Auswahl angeboten (aufsteigend geordnet):

- reserved,
- with personal risk,
- normal und
- fast.

In der grafischen Simulation wird eine Bewegungveränderung der EHB-Fahrzeuge erst dann angezeigt, wenn das gegebene EHB-Fahrzeug eine Blockstelle passiert.

Zeitverzögerungskonstanten für Load, Unload etc.

Da bestimmte Vorgänge wie z. B. das Verladen der Ware vom Hochregallager auf ein EHB-Fahrzeug eine bestimmte Zeitdauer in Anspruch nehmen, müssen diese auch in der Simulation berücksichtigt werden. Zur Vereinfachung wird unabhängig von der zu verladenden Ware von konstanten Verzögerungszeiten bei Lade- und Entladevorgängen ausgegangen. Diese stellen einen Mittelwert aus den vorgegebenen Erfahrungsdaten dar.

3.2 SIB-Arten und ihre Verwendung

3.2.1 Einleitung

Das IKEA-Plugin basiert – wie bereits erwähnt – auf zwei Typen von SIBs. Zum einen existieren SIBs zur Realisierung der einfachen Fahrstrecke und zum anderen werden funktionale SIBs (`Kommissionierplatz`, `EHB-Fahrzeugstation`, `Ladestation`, `Entladestation`, `Leerpalettenentladestation`) bereitgestellt. Der SIB-Graph des *jABC* veranschaulicht auf übersichtliche Weise die zusammengefügte Fahrstrecke, bringt selbst jedoch keinerlei Funktionalität mit. Die SIBs bilden hierbei lediglich den Datenspeicher, der entsprechende Methoden zum Auslesen und Setzen der einzelnen Attribute bereitstellt. Die nachfolgenden Abschnitte erläutern die Aufgaben der einzelnen SIBs im Detail.

3.2.2 Design-Annahmen

Zur performanten und einfacheren Implementierbarkeit müssen einige Design-Annahmen getroffen werden.

Vereinfachend wird angenommen, dass das gesamte Schienennetz in Blockstellen eingeteilt wird. Jedes EHB-Fahrzeug befindet sich damit nicht mehr absolut auf der Strecke, sondern lediglich innerhalb einer bestimmten Blockstelle. Es lässt sich damit nicht mehr sagen, wo exakt sich ein EHB-Fahrzeug befindet, sondern nur noch,

in welchem Intervall es sich aufhält. Dadurch ist eine Implementierung des Schienennetzes mittels eines Arrays möglich, wobei jede Array-Position eine Blockstelle repräsentiert.

Durch diese Einteilung in Blockstellen bietet sich eine Realisierung des Zeitfaktors mittels diskreter Zeitintervalle an. Diese bedingen eine weitere Design-Annahme bezüglich der Geschwindigkeit.

Würde jeder Streckenabschnitt eine beliebige Geschwindigkeit der Fahrzeuge zulassen, so würde die Berechnung eines geeigneten Zeitintervalls unverhältnismäßig schwierig (nähere Erläuterungen: siehe Kapitel 3.6). Daher kann die Geschwindigkeit eines Streckenabschnitts nur aus einer vorher definierten Menge gewählt werden.

3.2.3 Fahrstrecken-SIBs

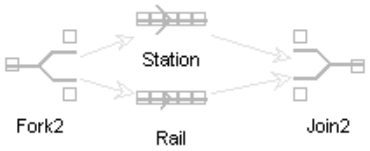


Abbildung 3.2: Ein Stück Gleisstrecke, gebildet aus Fahrstrecken-SIBs

Die Fahrstrecken-SIBs sind vergleichbar mit den Bauelementen einer Modelleisenbahn. Es ist möglich, sie in beliebiger Konstellation zu kombinieren und damit jedes gewünschte Streckennetz abzubilden. Ähnlich einer Modelleisenbahn stehen Geraden (`Rail`), Verzweigungen (`Forks/Joins`) und Bahnhöfe (`Stations`) zur Verfügung. Zur grafischen Veranschaulichung kann die Darstellung der SIBs im Graphen des *jABC* in 90 Grad Schritten gedreht werden. Damit ist eine realistischere und intuitiv zu erkennende Abbildung der Realität möglich. Dazu bietet das *jABC* die Möglichkeit, einige Eigenschaften direkt in den SIBs zu verändern.

Um die Blockstellen eines jeden SIBs abzubilden, erhält jeder SIB eine Instanz der Datenstruktur `Status`. Diese ermöglicht es, jede Blockstelle mit genau einem EHB-Fahrzeug zu belegen beziehungsweise diese über die Strecke zu bewegen (siehe Kapitel 3.4). Die Anzahl der Blockstellen eines SIBs und damit auch implizit die Länge des Gleisstückes können direkt vom Nutzer verändert werden.

Weiterhin besitzt jedes Fahrstrecken-SIB ein Attribut für Geschwindigkeit, das mit einem Wert aus einer zuvor definierten Menge von Geschwindigkeiten (vgl. auch Kapitel 3.2.2) belegt werden kann.

Des weiteren beinhaltet jedes Fahrstrecken-SIB eine *Routing-Tabelle*, welche Informationen über erreichbare Ziele und die dafür einzuschlagende Fahrtrichtung enthält (siehe Abschnitt 3.5). Die Bedeutung und Eigenschaften der einzelnen SIBs sind im Folgenden näher beschrieben.

Rail

Das `Rail` symbolisiert das gerade Gleisstück des Modells. Ein `Rail`-SIB kann nur von einem Vorgänger-SIB erreicht werden und kann selbst wiederum nur einen Nachfolger besitzen. Da in einem `Rail`-SIB folglich keine Verzweigungen möglich sind, enthält es nur ein Element vom Typ `Status`, der von den Fahrzeugen von vorne nach hinten „durchfahren“ wird.

Station

Das `Station`-SIB symbolisiert den Bahnhof des Modells. Der Grundaufbau entspricht dabei dem eines `Rail`-SIBs, wobei es einige zusätzliche Eigenschaften besitzt. Lediglich in der Station ist es möglich, EHB-Fahrzeuge anzuhalten, um weitere Aktionen durchführen zu können. Damit bieten die `Station`-SIBs die einzige Schnittstelle für die später erläuterten Funktions-SIBs.

EHB-Fahrzeuge können nur an der ausgehenden Position angehalten werden. Alle nachfolgenden EHB-Fahrzeuge rücken bis zum gestoppten EHB-Fahrzeug auf und bleiben dann ebenfalls stehen.

Join

Die `Join`-SIBs symbolisieren die zusammenführenden Weichen des Modells. Dabei wird unterschieden zwischen der Zusammenführung von zwei auf einen beziehungsweise von drei auf einen Schienenstrang. Dies wird realisiert durch zwei unterschiedliche Implementierungen und demzufolge mit zwei verschiedenen `Join`-SIBs (`Join2`, `Join3`).

Im Gegensatz zu den vorgenannten SIBs ist hier die Anzahl der Blockstellen fest vorgegeben. Dies begründet sich darin, dass sich die Weiche in die einfahrenden und in den ausfahrenden Bereiche aufteilt. Dabei erreicht ein EHB-Fahrzeug zuerst einen der einfahrenden Bereiche und wird dann mit Hilfe des Fahralgorithmus entsprechend in den ausfahrenden Bereich weitergeleitet. Die drei/vier Bereiche sind intern durch je einen Status der Länge eins realisiert. Ein eventueller Rückstau würde sich somit sofort auf die vorausgehenden SIBs ausbreiten.

Fork

Die `Fork`-SIBs symbolisieren die verzweigenden Weichen des Modells. Ähnlich zu den Join-SIBs wird auch hier zwischen der Verzweigung von einem auf zwei beziehungsweise von einem auf drei Schienenstränge unterschieden. Auch hier gibt es aufgrund der unterschiedlichen Implementierungen zwei unterschiedliche Fork-SIBs (Fork2, Fork3). Die Anzahl der Blockstellen in den Fork-SIBs ist ebenfalls fest vorgegeben.

Im Gegensatz zu den zusammenführenden Weichen verfügen verzweigende Weichen jedoch über nur einen einfahrenden Bereich und über zwei beziehungsweise drei

ausfahrende Bereiche. Ein einfahrendes EHB-Fahrzeug wird durch ein Zusammenspiel von Fahralgorithmus und Routing an den entsprechenden ausfahrenden Bereich weitergeleitet. Die Bereiche sind je mit einem Status der Länge eins realisiert.

3.2.4 Funktions-SIBs

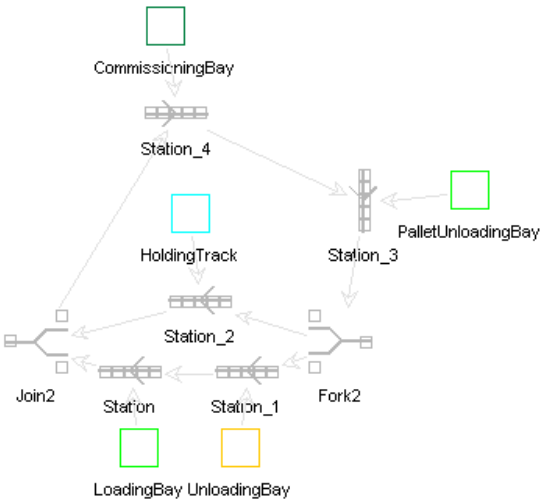


Abbildung 3.3: Ein Stück Gleisstrecke, versehen mit Funktions-SIBs

Während Fahrstrecken-SIBs das Schienennetz abbilden und die darauf fahrenden EHB-Fahrzeuge beinhalten, dienen die Funktions-SIBs der Interaktion zwischen Fahrstreckenelementen und Hochregallager, Kommissionierplatz, EHB-Fahrzeugstation und Leerpalettenentladestation. Die Schnittstellen zum Schienennetz werden hier ausschließlich durch Station-SIBs repräsentiert. Dabei ist eine eindeutige Zuordnung zwischen Funktion-SIBs und Station-SIBs vonnöten.

LoadingBay

Das `LoadingBay`-SIB symbolisiert die Beladeschnittstelle zum Hochregallager. Hier werden die *Artikel-Paletten* (homogene Paletten, die von EHB-Fahrzeugen transportiert werden können und von denen kommissioniert wird) gepuffert. Jede Artikel-Palette enthält intern eine Zielliste, die beschreibt, wo und in welcher Reihenfolge diese Artikel-Palette kommissioniert werden sollte. Über die Klasse `Datahandler` werden die entsprechenden Artikeln in den `LoadingBay`-SIBs bereitgestellt. Nach dem FIFO-Prinzip wird immer die am längsten bereitstehende Ware zuerst übergeben.

Erreicht ein EHB-Fahrzeug ohne Ladung die ausfahrende Position des zugehörigen Station-SIBs, so wird dieses für die Dauer des Ladevorgangs angehalten und eine

der wartenden Ladungen an das EHB-Fahrzeug übergeben. Nach dem Beladen fährt das EHB-Fahrzeug gemäß des Ziels der Artikel-Palette einen Bahnhof an.

HoldingTrack

Das `HoldingTrack`-SIB symbolisiert die EHB-Fahrzeugstation des Modells. Hier befinden sich alle leeren EHB-Fahrzeuge und warten auf ihr nächstes Ziel. Über den Datahandler wird das Ziel an das EHB-Fahrzeug übergegeben.

UnloadingBay

Im Gegensatz zum LoadingBay-SIB symbolisiert das `UnloadingBay`-SIB die Entladeschnittstelle zum Hochregallager. Hier werden die nach der Kommissionierung angebrochenden Artikel-Paletten gepuffert. Über den Datahandler werden die entsprechenden Paletten wieder im Hochregallager abgelegt. Erreicht ein EHB-Fahrzeug mit angebrochener Ladung planmäßig ein UnloadingBay, so wird dieses für die Dauer des Entladevorgangs angehalten und die angebrochene Ladung an UnloadingBay übergeben. Nach der Entladung fährt das EHB-Fahrzeug zum LoadingBay, falls der Puffer für Artikel-Paletten im LoadingBay nicht leer ist, ansonsten fährt es zum HoldingTrack.

PalletUnloadingBay

Das `PalletUnloadingBay`-SIB symbolisiert die Palettenentnahmestation des Modells. Hier werden die leeren Artikel-Paletten von einem EHB-Fahrzeug entnommen und verworfen.

Erreicht ein Fahrzeug die Entnahmeposition der zugehörigen Station, so wird dieses dort für die Dauer der Entnahme angehalten und die noch geladene leere Palette entfernt. Nach der Entnahme der leeren Artikel-Paletten fährt das EHB-Fahrzeug direkt zum `HoldingTrack`.

CommissioningBay

Das `KommissioningBay`-SIB symbolisiert einen Kommissionierplatz des Modells. Es besteht aus den entsprechenden Kommissionierpaletten, die derzeit an diesem Arbeitsplatz kommissioniert werden. Die maximale Kapazität an Paletten ist auf sechs voreingestellt. Erreicht ein EHB-Fahrzeug die Entnahmeposition innerhalb der zugehörigen Station, so wird es dort für die Dauer der Entnahme angehalten und die entsprechende Teilladung entnommen und den Kommissionierpaletten zugeführt. Sobald eine Palette fertig kommissioniert wurde, wird diese durch eine neue ersetzt.

3.3 Das IKEA-Plugin

3.3.1 Einleitung

Für die funktionale Erweiterung des Modells wurde ein eigenes Plugin entwickelt, das als optionales Modul im *jABC* eingebunden werden kann. Dieses Plugin stellt alle im folgenden Abschnitt genannten Funktionen zur Verfügung.

3.3.2 Funktionalitätsbeschreibung

Erzeugen einer Batch

Zur Erzeugung der zu simulierenden Batch wurde ein eigener Generator implementiert. Dieser erzeugt anhand der gewählten Parameter eine Batch, die an das Modell übergeben wird. Um einen Vergleich zwischen dem Uppaal und *jABC*-Modell ziehen zu können, wurde die hierarchische Struktur der Uppaal-Batch übernommen. Diese setzt sich aus LKW-Aufträgen, die sich wiederum in Palettenaufträge unterteilen lassen, zusammen. Für jede einzelne Hierarchiestufe lässt sich die Anzahl pro höhergelegener Stufe individuell festlegen. Außerdem kann eine prozentuale Abweichung angegeben werden, um eine realistischere Verteilung der Batch zu erhalten. Eine nachträgliche Änderung der erzeugten Batch ist im Programm nicht vorgesehen.

Um eine erzeugte Batch zwischen den beiden Modellen auszutauschen, wird eine Laden-/Speicher-Funktionalität zur Verfügung gestellt, welche den Austausch über XML-Dateien ermöglicht.

Das Austauschformat ist durch die folgende DTD spezifiziert.

```
1 <!ELEMENT WAREHOUSE (BATCH,HBR)>
2 <!ELEMENT BATCH (TRUCKORDER)*>
3 <!ELEMENT TRUCKORDER (PALLETORDER)*>
4 <!ELEMENT PALLETORDER (POSITION)*>
5 <!ELEMENT POSITION (ITEM,QUANTITY)>
6 <!ELEMENT ITEM (#PCDATA)>
7 <!ELEMENT QUANTITY (#PCDATA)>
8 <!ELEMENT HBR (ITEMQTY)>
9 <!ELEMENT ITEMQTY (#PCDATA)>
```

Überprüfungs- und Init-Funktionen

Um die Vollständigkeit der Kantenbeschriftungen zu gewährleisten bzw. herzustellen, besteht die Möglichkeit alle *Branches* automatisch beschriften zu lassen. Hierbei werden die Kanten der Forks und Joins und die Kanten zwischen Funktions-SIBs und Station-SIBs entsprechend beschriftet. Dies kann jedoch zur Folge haben, dass

die Beschriftung der Weichen nicht mehr intuitiv erscheint, da das Plugin das Layout des Modells nicht berücksichtigt.

Des weiteren kann eine einfache Verifikation des Modells durchgeführt werden. Diese prüft, ob alle SIBs über die korrekte Anzahl an ein- und ausgehenden Kanten verfügen und ob Kanten existieren, die nicht von einem SIB ausgehen bzw. in einem SIBs enden. Dabei erfolgt allerdings keine logische Prüfung des Modells, etwa ob die Reihenfolge der Fahrtstrecken-SIBs korrekt ist oder ob die Fahrstrecke einen geschlossenen Kreis bildet.

Nachdem die Modellierung abgeschlossen ist, kann das Modell initialisiert werden. Während der Initialisierung erfolgt zum einen die in Kapitel 3.5 beschriebene Berechnung der benötigten Routinginformationen, aber auch die Instanziierung der Fahrzeuge, etc. Die Simulation kann daher nur auf einem zuvor initialisierten Modell durchgeführt werden.

Simulation

Die Simulation wird durch den Benutzer gesteuert, d. h. er kann sie beliebig starten, schrittweise ausführen oder anhalten. Des weiteren kann die Simulationsgeschwindigkeit und die Bildaktualisierungsrate der Fahrzeugbewegung frei über den Inspektor der *Globalen Einstellungen* variiert werden. Dieses vereinfacht die visuelle Analyse am Modell und über die Inspektoren.

Inspektoren

Die Inspektoren des Plugins lassen sich in zwei Typen unterteilen, nämlich *Eingabe-* und *Ausgabeinspektoren*.

Die Eingabeinspektoren werden beim Starten des *jABC* durch das IKEA-Plugin aktiviert und nach erfolgreicher Initialisierung des Modells wieder deaktiviert. Sie bieten die Möglichkeit an, die Daten einzugeben, die für die Simulation des IKEA-Modells benötigt werden. Dabei bezieht sich der Inspektor immer auf das aktuell gewählte SIB. Die beiden Eingabeinspektoren sind der *Car Positioning Inspector* und der *Create Order Pallet Inspector*.

Der *Car Positioning Inspector* ermöglicht dem Anwender, die EHB-Fahrzeuge auf der Fahrtstrecke frei zu platzieren. Hierzu können alle Bufferpositionen des aktuell gewählten SIBs einzeln markiert und somit mit einem Fahrzeug belegt werden. Änderungen werden dabei sofort an das SIB übergeben und sind für den Anwender sichtbar.

Mit Hilfe des *Create Order Pallet Inspector* können zusätzliche Palettenaufträge erzeugt werden. Hierdurch lässt sich eine erzeugte Batch nachträglich individualisieren. Diese Funktion dient jedoch lediglich als Ergänzung zum Batch-Generator und sollte nicht zur Erzeugung einer kompletten Batch benutzt werden.

Die Ausgabeinspektoren dienen der Analyse des aktuellen Zustands des IKEA-Modells während der laufenden Simulation. Daher stehen sie erst nach der Initialisierung des Modells zur Verfügung. Es existieren die folgenden Ausgabeinspektoren: *Car Inspector*, *Load Inspector*, *Unload Inspector*.

Der *Car Inspector* zeigt den aktuellen Zustand der EHB-Fahrzeuge. Das vom Benutzer gewählte Fahrzeug wird dabei farblich im Modell gekennzeichnet, um eine Verfolgung zu vereinfachen. Die aktuelle Ladung und Fahrziele können wiederum im Inspektor selbst abgelesen werden.

Der *Load Inspector* dient zum Anzeigen der bereitgestellten *Cargos* in *Load*-SIBs (s. Kapitel 3.2.4). *Cargos* werden in diesem Inspektor ebenfalls in einem Baum dargestellt. Wählt der Anwender ein *Cargo* aus, werden die Informationen des *Cargos* angezeigt.

Laden/Speichern von Belegungen

Da das *jABC* standardmäßig nur die SIBs, aber nicht die durch das Plugin benötigten Objekte speichern kann, wurde hierfür eine eigener Programmteil entwickelt. Dieser speichert bzw. lädt einen aktuellen Systemzustand mit Hilfe einer XML-Datei. Konkret setzt sich der aktuelle Systemzustand zusammen aus:

- den Paletten in den Be- und Entladestationen des Hochregallagers,
- den zu kommissionierenden Palleten der einzelnen Kommissionierbahnhöfe,
- den auf der Strecke befindlichen Fahrzeugen,
- den im Hochregallager vorhandenen Anbruchpaletten und
- den (un-)zugewiesenen Aufträgen der Batch.

Um inkonsistente Belegungen zu verhindern, verwendet das Plugin zwei Mechanismen. Zum einen wird beim Speichern einer Belegung das zugehörige Modell mit hinterlegt, sodass die Belegung auch nur auf dieses Modell wieder geladen werden kann. Zum zweiten kann das Speichern nur auf initialisierten Modellen erfolgen, da so gewährleistet wird, dass bereits ein gültiger Systemzustand existiert.

Batch Logging und Trace Play

Um eine Analyse der rein rechnerischen Simulation zu ermöglichen, können alle Vorgänge extern protokolliert werden. Da aber nur Aktionen protokolliert werden können, wird zu Beginn der Simulation die aktuelle Belegung gespeichert. Um die zum Protokoll passende Belegung identifizieren zu können, wird diese im ersten Datensatz der Datei hinterlegt.

Die protokollierten Vorgänge, die während der Simulation geloggt werden, sind im Einzelnen:

- Fahrzeugbewegungen auf einer Schiene bzw. zwischen zwei Schienen,
- Be- und Entladen eines Fahrzeugs,
- Kommissionieren eines Artikels vom Fahrzeug,
- Entfernen einer fertig kommissionierten Palette,
- Entladen einer Leerpalette im Leerpalettenmagazin und
- Wiedereinlagern einer Palette im Hochregallager.

Anhand dieser Informationen und der gespeicherten Belegung ist es möglich eine simulierte Batch erneut abzuspielen. Hierzu wurde der sogenannte *Trace Player* implementiert. Dieser ermöglicht, ähnlich wie bei der richtigen Simulation, eine (schrittweise) Ausführung der Aktionen und zusätzlich den Sprung an eine frei wählbare Stelle der Simulation. Da dieses Replay mit denselben Objekten arbeitet wie die originale Simulation, ist es zum einen möglich die Inspektoren wie in einer richtigen Simulation zu nutzen und zum anderen an einer beliebigen Stelle weiterzusimulieren. Der wirkliche Vorteil der Analyse mit dem Trace Player im Vergleich zur visuellen Simulation ist jedoch die *Undo*-Funktion. Hierdurch ist es jederzeit möglich Aktionen zurückzuspringen und somit wichtige Vorgänge erneut zu betrachten ohne die gesamte Trace erneut abspielen zu müssen.

3.3.3 Implementierung

Suchen und Registrieren des neuen Plugins erfolgt problemlos durch die Plugin-Suchfunktion des *jABC*. Nach dem Registrieren des neuen Plugins startet *jABC* das Plugin automatisch mit. Das IKEA-Plugin ist eine Java-Klasse, die das Interface `de.metaframe.common.plugin.Plugin` von *jABC* implementiert. Die Implementierung der eigentlichen Funktionalitäten des IKEA-Plugins werden im folgenden beschrieben.

Erzeugen einer Batch

Beim `Erzeugen einer Batch` wurde bei der Implementierung das Software-Muster des *Model-View-Controller* eingesetzt. Dadurch ist die Trennung zwischen `GUI-BatchGenerator`, `OrderController` und `BatchFactory` möglich. Die GUI-Klasse ist dafür zuständig, die Oberfläche darzustellen, während die Controller-Klasse dafür sorgt, auf die Aktionen zu reagieren. Die Klasse für das Erzeugen der Objekte hat die Aufgabe, die gewünschten Objekte gemäß den Parametern, die man in der GUI einstellen kann, zu erzeugen. Bezüglich des Batchgenerators werden die Palettenaufträge, die die Positionen enthalten, erzeugt. Die Anzahl der Palettenaufträge, der Artikel und der Positionen wird in den Textfeldern der GUI eingegeben.

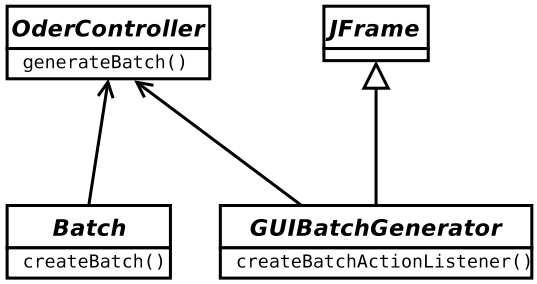


Abbildung 3.4: Batch Generator

Simulation

Bevor man das erstellte Modell in *jABC* simuliert, sollte die folgende Reihenfolge eingehalten werden. Als erstes werden Fahrzeuge auf Strecken-SIBs gesetzt. Dieses erfolgt durch den *CarInsertInspektor*. Anschließend wird eine Batch durch den *BatchGenerator* erzeugt. Als letzte Vorbereitung werden die Parameter des Modells eingestellt. Hier handelt es sich um die Parameter, die zur Berechnung der Fahrgeschwindigkeit und der Dauer der Belade-/Entlade-/Kommissionierungsvorgänge benutzt werden. Jeder Parameter hat einen voreingestellten Wert, der in einer Properties-Datei gespeichert wird. Die Properties-Datei kann mit Hilfe eines Einstellungsdialoges geöffnet und verändert werden.

Es gibt drei Simulationsarten. Diese sind Starten, schrittweise Ausführen und Anhalten. Alle drei Simulationsarten werden grundsätzlich durch die Methode `IKEABaseClass.getInstance().singleRound()` realisiert. Diese sorgt für die Simulation einer einzigen Runde der Simulation. Im Einzelnen sind das folgende Aufgaben:

- Aufrufen der von der Klasse DataHandler bereitgestellten static Methode `DataHandler.refillCommissioningBays()`. Diese sorgt für das Erzeugen der Kommissionierungsaufträge an den Kommissionierstellen. Wie die Kommissionieraufträge erzeugt werden ist im Abschnitt 3.7 erklärt.

- Aufrufen der von der Klasse DataHandler bereitgestellten static Methode `DataHandler.createCargoFromPallet()`. Diese sorgt für das Erzeugen der für die Kommissionierung benötigten Cargos und das Einfügen der Cargos in die Funktions-SIBs (*LoadingBays*). Wie die Cargos erzeugt werden ist in Abschnitt 3.7 beschrieben.

- Aufrufen der Methode `Simulation.getInstance().SimulateRound()`. Diese sorgt für die Bewegung der EHB-Fahrzeuge im Modell. Wie die EHB-Fahrzeuge auf den Strecken-SIBs bewegt werden ist in Abschnitt 3.4 erklärt.

Inspektoren

Inspektoren müssen das Interface `de.metaframe.client.inspector.Inspector` implementieren und können durch die Methode `InspectorPane.getInstance().addInspector(EigenerInspektor)` ins *jABC* geladen werden.

Laden/Speichern von Belegungen

Beim *Laden/Speichern von Belegungen* werden die Zustände, die das Modell komplett beschreiben, gespeichert und wieder ins Modell geladen.

Die Rückgabe aller benötigten Objekte erfolgt durch Get-Methode der jeweiligen Klassen z. B. `IKEABaseClass.getInstance().getCars()`. Zum Setzten dieser Objekte stehen die entsprechenden Set-Methoden zur Verfügung. Alle diese Instanzen werden in einer XML-Datei gespeichert. Zum Transformieren bzw. Rücktransformieren eines Objektes ins XML-Format wird dabei die Bibliothek `XStream` [JST] benutzt.

3.4 Der Fahralgorithmus

3.4.1 Einleitung

Dieses Kapitel beschäftigt sich damit, wie die Fahrzeuge während der Simulation über die Strecke bewegt werden. Dabei wird sowohl auf die realen Gegebenheiten in einem Hochregallager mit elektrischer Hängebahn eingegangen, wie auch die gemachten Abstraktionen erklärt, die verwendet wurden, um die Eigenschaften in die Simulation zu übertragen.

3.4.2 Designannahmen

Betrachtet man ein klassisches Schienensystem, wie beispielsweise bei einer Eisenbahn oder eben einer Elektrohängebahn, so sind einige Dinge implizit klar. Züge auf einem einspurigen Gleis können sich nicht gegenseitig überholen, da es sonst einen Zusammenstoß geben würde. Auch lässt sich der Abstand mit bloßem Auge relativ leicht erkennen. Leider ist der Erhalt dieser Eigenschaften in der Simulation nicht ganz so einfach. Hier müssen dieselben Eigenschaften abgebildet werden, wobei die Zusammenhänge zu allen anderen Komponenten der Simulation, wie z. B. der Realisation von Zeit, erhalten bleiben müssen. Die Simulation verläuft nicht in Echtzeit, sondern mittels sogenannter *Ticks*. Dabei steht ein einzelner Tick für eine fest definierte Zeiteinheit, die dem größten gemeinsamen Teiler aller vorkommenden Zeiten entspricht. Zeiten kleiner als ein solcher Tick werden durch die Simulation nicht betrachtet. Dies führt dazu, dass die Position der Fahrzeuge nicht absolut auf der Strecke angegeben wird, sondern diese sich immer in einer sogenannten *Blockstelle*

befinden. Innerhalb einer Blockstelle kann sich immer nur ein einzelnes Fahrzeug befinden. Dabei ist die Länge einer einzelnen Blockstelle frei konfigurierbar. Wählt man also einen niedrigen Wert für die Länge einer Blockstelle, wird die Simulation entsprechend realer, es führt jedoch zu einer Verlangsamung der Simulation. Hinzu kommt, dass die Simulation keine Mindestabstände kennt. Der Abstand, der zwischen zwei Fahrzeugen herrschen muss, ist ebenfalls durch die Blockstellen gegeben. Innerhalb einer Blockstelle bewegt sich ein Fahrzeug nicht. Es kennt lediglich die Anzahl an Ticks, die es benötigt um die Blockstelle zu durchfahren, und bewegt sich nach Ablauf dieser Ticks dorthin.

3.4.3 Die Strecke

Die eigentliche Strecke wird mittels einzelner *SIBs* verschiedenen Typs zusammengesetzt. Dabei gibt es neben den verschiedenen Weichen für zwei oder drei Gleise zum einen Bahnhöfe und zum anderen gerade Gleise. Diese müssen nicht immer gleichlange Gleise beschreiben, sondern ihre Länge ist individuell parametrisierbar. Für die Fahrbewegungen werden jedoch nur Abschnitte gleicher Länge benötigt, welches durch die Unterteilung in Blockstellen realisiert wird. Dabei kann der Nutzer die Länge einer einzelnen Blockstelle selbst festlegen. Stellt beispielsweise ein SIB einen 10m langen und ein anderes SIB einen 12,5 m langen Gleisabschnitt dar und beträgt die Länge einer Blockstelle 2,5m, so wird das erste SIB in 4 Blockstellen unterteilt, während sich das zweite SIB in 5 Blockstellen aufteilt. Weichen besitzen grundsätzlich eine Blockstelle für jedes eingehende, sowie eine Blockstelle für jedes ausgehende Gleis. Gleichzeitig dienen bestimmte Blockstellen als Trigger. Erreicht ein Fahrzeug die vorderste Blockstelle eines Bahnhofs, so werden Aktionen wie das Be- und Entladen eines Fahrzeugs automatisch angestoßen. Doch wie werden die einzelnen Fahrzeuge nun mit der richtigen Geschwindigkeit durch diese Blockstellen bewegt?

3.4.4 Die Initialbelegung

Die Verteilung der einzelnen Fahrzeuge auf die Strecke geschieht durch den Anwender. Dieser hat vor der Initialisierung die Möglichkeit, seine Fahrzeuge nach eigenen Vorstellungen auf der Strecke zu verteilen. Dabei wird jede Blockstelle initial mit Null vorbelegt, welches gleichbedeutend mit einer freien Blockstelle ist. In allen Blockstellen, die der Anwender belegt, werden die Nullen durch einen negativen Zahlenwert ersetzt. Erst bei der Initialisierung wird die Belegung mit dem eindeutigen Wert, der ein Fahrzeug identifiziert, überschrieben. Auf diese Weise ist es möglich, vor der Initialisierung gesetzte Fahrzeuge auch wieder zu entfernen. Gleichzeitig mit der Initialisierung werden die Fahrzeuge mit ihren initialen Tickwerten belegt (siehe Abschnitt 3.6). Allerdings müssen nicht nur die Fahrzeuge gesetzt, sondern auch die Fahrstrecke muss initialisiert werden. Während sich die Übergänge auf einer geraden

Strecke oder in einem Bahnhof eindeutig zuordnen lassen, müssen die einzelnen Verbindungen zwischen den Blockstellen der einzelnen SIBs bzw. innerhalb der Weichen *errechnet* werden.

Damit ein Modell auch nach dem Speichern und dem erneuten Laden wieder in dem Zustand ist wie zuvor, müssen die einzelnen Kanten des *jABC*s mit Bezeichnern versehen werden. Dies ist in der Simulation mittels sogenannter Ports gelöst. Dabei gibt es drei Typen von Ports, IN-Ports für eingehende Kanten, OUT-Ports für ausgehende Kanten und INT-Ports für interne Kanten innerhalb von Weichen. Eine Auflistung aller SIB-Typen und ihrer zugehörigen Kanten zeigt Tabelle 3.1.

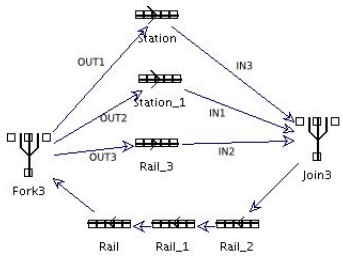
SIB-Typ	Ports
Rail	IN1,OUT1
Station	IN1,OUT1
Fork2	IN1,OUT1,OUT2,INT1,INT2
Fork3	IN1,OUT1,OUT2,OUT3,INT1,INT2,INT3
Join2	IN1,IN2,OUT1,INT1,INT2
Join3	IN1,IN2,IN3,OUT1,INT1,INT2,INT3

Tabelle 3.1: Übersicht über die verschiedenen Ports

Die Vergabe der internen Ports (INT) ist dabei lediglich nötig, um eine effiziente Speicherung der Strecke zu ermöglichen. Es wird eine Datenstruktur mit eindeutigen Schlüsseln benötigt. Dies ist jedoch dann nicht mehr möglich, wenn von einer Blockstelle aus mehrere Ziele erreicht werden können (verzweigende Weichen). Der Anwender selbst bekommt von diesen Ports nichts mit. Die Beschriftung der Kanten mit den entsprechenden IN und OUT-Ports dagegen kann jedoch wahlweise vor dem Start der Simulation manuell vorgenommen werden. Verzichtet der Benutzer auf eine automatische Beschriftung und nimmt diese manuell vor, so kann die Bezeichnung der Ports dem Modell angepasst werden und die grafische Abbildung während der Simulation ist verständlicher. Es müssen auch nicht alle Kanten beschriftet werden. Lediglich die Kanten, bei denen es eine Wahlmöglichkeit bei der Vergabe der Ports gibt, ist eine Beschriftung notwendig. Das sind zum einen die Kanten, die eine zusammenführende Weiche (JOIN) als Ziel haben oder aber die von einer verzweigenden Weiche (FORK) ausgehen. Sind alle Daten abgelegt, kann mittels einer einfachen Anfrage an eine Hashmap aus einem SIB und seinem zugehörigen Port die entsprechende Nachfolgekonstellation erzeugt werden.
Die erzeugte Datenstruktur wird in Abbildung 3.5 beispielhaft verdeutlicht.

3.4.5 Die Fahrbewegung

Mit dem Abschluss der Initialisierung sind alle Blockstellen entsprechend miteinander verbunden und es existieren keine Platzhalter mehr anstelle echter Fahrzeuge.



Vorgänger		Nachfolger	
SIBName	SIBPort	SIBName	SIBPort
Fork3	OUT1	Station	IN1
Fork3	OUT2	Station_1	IN1
Fork3	OUT3	Rail_3	IN1
Station	OUT1	Join3	IN3
Station_1	OUT1	Join3	IN1
Rail_3	OUT1	Rail_3	IN2
Join3	OUT1	Rail2	IN1
Rail_2	OUT1	Rail_1	IN1
Rail_1	OUT1	Rail	IN1
Rail	OUT1	Fork3	IN1

Abbildung 3.5: Links: Eine einfache Strecke, die Kantenbeschriftungen sind zur Veranschaulichung sichtbar gemacht. Rechts: Die erzeugte Tabelle nach der Initialisierung, die SIB-internen Übergänge wurden der Einfachheit halber weggelassen.

Nun kann mit der eigentlichen Simulation begonnen werden. Dazu wird jede Runde eine gewisse Menge an Ticks simuliert (für nähere Erläuterungen, siehe Abschnitt 3.6). Erreicht in einer Runde ein Fahrzeug die Grenze von null Ticks oder unterschreitet sie sogar, so hat es seine Fahrzeit aufgebraucht und kann in die nächste Blockstelle einfahren. Jetzt kann es jedoch vorkommen, dass die vorherige Blockstelle bereits durch ein anderes Fahrzeug belegt ist. In diesem Fall muss ermittelt werden, ob die blockierende Hängebahn in dieser Runde ebenfalls null oder weniger Ticks erreicht hat und wenn ja, ob sie nicht ebenfalls blockiert wird. Daher ist der Fahralgorithmus in verschiedene Stufen unterteilt:

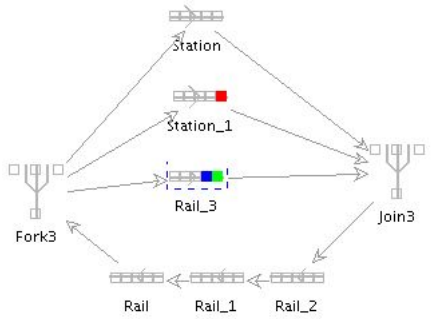
```
checkForMoves();
createExecutionOrder();
calculateTimeForBlockedCars();
moveTheCars();
resetLists();
updateHoldingTrack();
```

Dabei werden in der ersten Phase all die Fahrzeuge ermittelt, die sich diese Runde bewegen möchten. Da die Fahrzeuge in einer Tabelle aufsteigend nach Ihren Ticks sortiert vorliegen, ist dies mit minimalem Zeitaufwand möglich. In der zweiten Phase wird eine Ordnung auf den Fahrzeugen erstellt. Dabei werden nur die Fahrzeuge betrachtet, die bereits im vorherigen Schritt ermittelt wurden. Nun werden die Fahrzeuge bestimmt, die nicht durch ein anderes Fahrzeug blockiert werden. Diese können direkt in die Ausführungsliste übernommen werden. Dann werden alle die

EHB-Fahrzeuge ermittelt, die zuvor durch andere Fahrzeuge blockiert wurden, die sich nun in der Ausführungsliste befinden. Diese werden der Ausführungsliste angehängt, sodass ihre Bewegung nach der ihres Hindernisses stattfindet. Dieser Prozess wird so lange wiederholt, bis alle Fahrzeuge bewegt werden können oder aber keine weiteren Bewegungen mehr möglich sind (innerhalb einer Iteration wird kein Fahrzeug mehr befreit). Fahrzeuge, die nun immer noch blockiert werden, können sich diese Runde sicher nicht mehr bewegen.

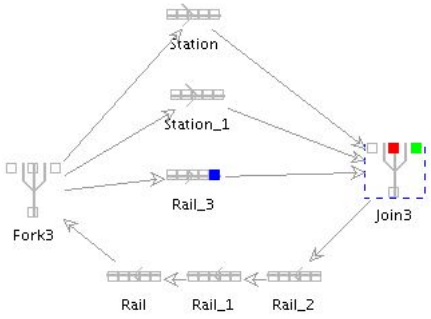
```
  if #Blocked == 0 : return;
  do{
    #ElementsBlockedBeforeExecution := BlockedMoveList.size();
    for each element in BlockedMoveList {
         if (CanMoveList.contains(element.successor)){
               BlockedList.remove(element);
               CanMoveList.add(element);
          }
     }
//break if no elements are moved from BlockedList to CanMoveList
 } while (BlockedMoveList.size() <  #ElementsBlockedBeforeExecution);
```

Damit diese jedoch nicht jede Runde wieder angefragt werden, bekommen sie eine gewisse *penalty* aufgeschlagen. Dies geschieht in Phase drei. Dabei wird für jedes Fahrzeug nachgesehen, wie lange die Blockstellen, die es gerne befahren möchte noch blockiert ist. Als penalty bekommt es dann genau diesen Wert zugewiesen. Es wird durch den Fahralgoritmus bzw. die Simulation also erst wieder betrachtet, wenn auch sein Vorgänger sich bewegen wird. Auf diese Weise wird der Overhead deutlich reduziert und die Performance der Simulation deutlich gesteigert. Nachdem nun alle Fahrzeuge abgehandelt wurden, die sich diese Runde nicht bewegen konnten, muss nun die Liste der Fahrzeuge, die sich bewegen können, von vorne nach hinten abgearbeitet werden. Dies kann nun in liearer Zeit erfolgen. Danach gilt es noch die entsprechenden Listen wieder zurückzusetzen und alles für die nächste Runde vorzubereiten. Damit ist der eigentliche Fahralgorithmus abgeschlossen. Eine Einschränkung gibt es jedoch noch.



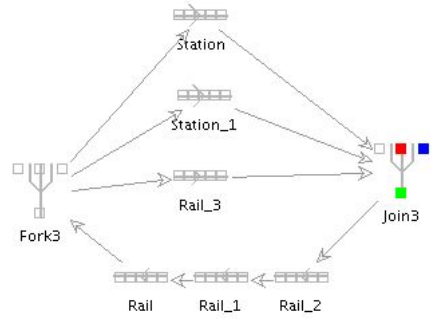
Das linke Bild präsentiert eine mögliche Fahrsituation. Die Streckenvernetzung entspricht der aus dem Beispiel für die Initialisierung.

Fahrzeug	gewünschte Position	Ticks
rot	Join3;IN1	10
grün	Join3;IN2	10
blau	Rail_3;OUT1	10



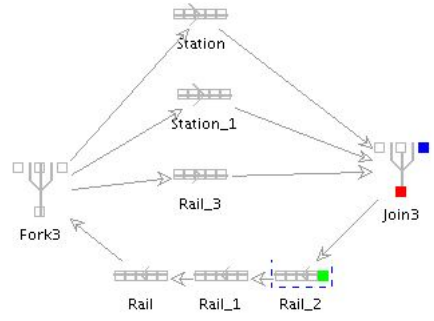
Da alle Fahrzeuge sich in dieser Runde bewegen konnten, gab es keine Fahrzeuge die blockiert wurden. Alle Fahrzeuge konnten ihr nächstes Ziel erreichen.

Fahrzeug	gewünschte Position	Ticks
rot	Join3;OUT1	10
grün	Join3;OUT1	10
blau	Join3;IN3	10



An dieser Stelle gibt es einen Konflikt an einer zusammenführenden Weiche. Rot und Grün wollen gleichzeitig einfahren, der Algorithmus entscheidet, dass Grün bevorzugt wird. Rot ist nun blockiert und bekommt den neuen Tickwert von Grün als *penalty*.

Fahrzeug	gewünschte Position	Ticks
rot	Join3;OUT1	10
grün	Rail_2;IN1	10
blau	Join3;OUT1	10



Auch diesmal gibt es einen Konflikt. Jetzt zwischen Rot und Blau. Diesmal darf Rot fahren und Blau wird mit den neuen Ticks von Rot als *penalty* belegt. Grün kann ungehindert fahren.

Fahrzeug	gewünschte Position	Ticks
rot	Rail_2;IN1	10
grün	Rail_2;IN1(2)	10
blau	Join3;OUT1	10

Abbildung 3.6: Ein einfaches Beispiel für eine mögliche Fahrsituation. Alle Fahrzeuge fahren mit gleicher Geschwindigkeit.

3.5 Das Routing

3.5.1 Einleitung

Das Routing ist dafür zuständig, die EHB-Fahrzeuge zu ihren Zielen zu leiten. Realisiert wird dies über Routing-Tabellen. Dabei wird für jedes Fahrstrecken-SIB eine eigene Routing-Tabelle erzeugt, welche dann bei der späteren Simulation ausgelesen werden kann.

Die Ziele selbst sind in den EHB-Fahrzeugen gespeichert und werden an verzweigenden Weichen (Fork) ausgelesen, da hier eine Entscheidung über die einzuschlagende Richtung getroffen werden muss (dies geschieht mit Hilfe des Fahralgorithmus). Gelangt also ein EHB-Fahrzeug an eine solche verzweigende Weiche, wird ein Ereignis erzeugt, welches über den Routing-Algorithmus und unter Angabe des Fahrzeugs den *Next-Hop* (siehe Kapitel 3.5.3) und damit die einzuschlagende Richtung zurückliefert.

An zusammenführenden Weichen (Join), normalen Schienen (Rail) und Bahnhöfen (Station) ist eine solche Entscheidung nicht zu treffen, da es hier jeweils nur eine Möglichkeit gibt, das EHB-Fahrzeug weiterzuleiten. Fährt ein EHB-Fahrzeug in einen Bahnhof ein, wird kontrolliert, ob es sein Ziel erreicht hat. Wenn dies der Fall ist, wird das Fahrzeug gegebenenfalls ent- bzw. beladen. Ist dies erfolgreich geschehen, wird es anhand seines nächsten Zieles weitergeleitet. Wurde die gesamte Fracht entladen, so wird das EHB-Fahrzeug zur nächsten Leerpalettenentladestation weitergeleitet. Hat ein Fahrzeug weder Fracht noch gespeicherte Ziele, so wird es automatisch zum Wartegleis (HoldingTrack) geroutet.

3.5.2 Design-Annahmen

Als grundsätzliche Design-Annahmen müssen auch beim Routing einige Eigenschaften als gegeben vorausgesetzt werden.

Die Bezeichner der einzelnen Fahrstrecken-SIBs müssen eindeutig sein. Sie dienen während des Routings zur eindeutigen Identifizierung der Ziele und der zu passierenden Fahrstrecken-SIBs, um zu den entsprechenden Zielen gelangen zu können. Weiterhin wird angenommen, dass das Streckennetz ein zusammenhängendes Schienennetz ist, auf dem von jedem Punkt aus jeder beliebige andere Punkt des Streckennetzes erreichbar ist. Als Grundlage dient hierbei das IKEA-Lagermodell.

Alle Fahrstrecken-SIBs werden als „Einbahnstraßen-SIBs“ vorausgesetzt. Es ist einem EHB-Fahrzeug grundsätzlich nur möglich, ein Fahrstrecken-SIB in *eine* Richtung zu durchfahren, wobei die zu durchfahrende Richtung unveränderlich ist.

3.5.3 Funktionalitätsbeschreibung

Initialisierung der Routing-Tabellen

In jeder Routing-Tabelle wird für jedes Ziel eine eigene Liste (Pfad) angelegt, in der jeder *Hop* auf dem Weg zu diesem Ziel in korrekter Reihenfolge gespeichert wird. Das Ziel selbst stellt hierbei den ersten Eintrag und der *Next-Hop* den letzten Eintrag dar. Es wird zu jeder Zeit sichergestellt, dass sich für jedes Ziel nur ein einziger Eintrag in jeder Routing-Tabelle befindet. Zur Initialisierung der Routing-Tabellen wird zunächst der zugrundeliegende Graph geladen. Dieser wird nun mit dem Depth-First-Search-Algorithmus (*DFS*) durchlaufen.

Es wird ein beliebiger Knoten als Startknoten gewählt. Von diesem Startknoten aus wird der DFS gestartet. Stößt der DFS dabei auf einen noch nicht besuchten Knoten, so wird mit dessen Nachfolger fortgefahren. Bei einer verzweigenden Weiche spielt es keine Rolle, welcher der Nachfolger zuerst betrachtet wird. Gelangt der DFS an einen bereits besuchten Knoten A, so kehrt der Algorithmus zurück zum vorher betrachteten Knoten B und aktualisiert dessen Routing-Tabelle mit der Routing-Tabelle des Knotens A.

Aktualisierung einer Routing-Tabelle

Die Aktualisierung einer Routing-Tabelle erfolgt immer, wenn der DFS einen Rekursionsrücksprung ausführt. Die Routing-Tabelle des Nachfolgers wird dann als Übergabewert an den Vorgänger überreicht und dessen Routing-Tabelle damit abgeglichen. Hierbei wird für jedes Ziel, welches in der übergebenen Routing-Tabelle vorhanden ist, kontrolliert, ob in der aktuellen Routing-Tabelle bereits ein Pfad zu diesem Ziel existiert. Zu diesem Zweck muss für jeden Pfad der Routing-Tabelle lediglich der erste Eintrag kontrolliert werden, da dieser immer das Ziel repräsentiert.

Ist ein Ziel in beiden Routing-Tabellen vorhanden, so wird überprüft, welcher von beiden die geringere Streckenlänge aufweist. Zusätzlich findet hier eine Überprüfung statt, welcher der beiden Pfade durch weniger Bahnhöfe verläuft. Diese Überprüfung hat Vorrang vor der kürzeren Route, da aus Effizienzgründen möglichst nie durch einen Bahnhof geroutet werden sollte, um dem realen System möglichst nahe zu kommen. Für das IKEA-Projekt in Dortmund-Mengede lässt sich dieser Umstand jedoch nicht komplett ausschließen, da bei Be- und Entladestationen im Hochregallager teilweise mehrere Bahnhöfe durchfahren werden müssen, um zum gewünschten Ziel zu gelangen. Dies geschieht aufgrund der Architektur der Anlage. Da im Hochregallagerbereich mehrere Bahnhöfe direkt aufeinander folgen, werden hier einige Bahnhöfe lediglich durchfahren, ohne dass eine Interaktion mit den EHB-Fahrzeugen stattfindet (*Transitbahnhof*).

Stellt sich der aktuelle Pfad als der „bessere“ heraus, bleibt die Routing-Tabelle unberührt. Ansonsten wird der Pfad durch den neuen Pfad ersetzt. Diese Prozedur stellt sicher, dass zu jedem Zeitpunkt immer nur ein Pfad zu einem bestimmten Ziel

in der Routing-Tabelle enthalten ist.

Aufgrund dieser Realisierung bezüglich der Streckenlängen, welche in den Tabellen mit abgespeichert werden, ist die übliche Umsetzung mit Hilfe einer Matrix an dieser Stelle nicht sinnvoll.

Wurden alle Knoten auf die Weise abgearbeitet, so gelangt der DFS zurück zum Startknoten. Ist auch dessen Routing-Tabelle aktualisiert, ist die Grundinitialisierung der Routing-Tabellen aller SIBs damit abgeschlossen.

Optimierung der Routing-Tabellen

Aufgrund von Schleifen im Schienennetz kann es vorkommen, dass nicht alle Ziele in allen Routing-Tabellen bekannt sind. Um dies zu korrigieren, wird nun eine Optimierung der Routing-Tabellen vorgenommen. Aus Effizienzgründen gibt es hierfür die Möglichkeit, eine von drei Optimierungsstufen auszuwählen (wählbar in den globalen Variablen der Simulation). Die Unterscheidung der drei Optimierungsstufen zeichnet sich wie folgt ab:

- 1. Optimierungsstufe: Allen Forks sind alle Ziele bekannt
- 2. Optimierungsstufe: Allen Forks und allen Zielen sind alle Ziele bekannt
- 3. Optimierungsstufe: Allen Fahrstrecken-SIBs sind alle Fahrstrecken-SIBs bekannt

Hierbei reicht für den Ablauf einer „normalen“ Simulation die erste Optimierungsstufe aus. Ist es allerdings gewünscht, zur Simulationszeit an einem erreichten Ziel online zu ermitteln, welches der verbleibenden Ziele eines EHB-Fahrzeugs das nächstbeste Ziel darstellt, so ist es notwendig, dass in diesem aktuellen Ziel die Entfernungen aller Ziele vorhanden sind. Dies wird mit der zweiten Optimierungsstufe gewährleistet. Mit der dritten Optimierungsstufe wird es möglich, auf eventuelle Stausituationen zu reagieren. Tritt ein Stau auf, so kann unmittelbar am Ende des Staus mit Hilfe der Routing-Tabelle ermittelt werden, welche Fahrstrecken-SIBs betroffen (nicht mehr zu erreichen) sind. Eine entsprechende Behandlung dieser SIBs kann dann vorgenommen werden. In der aktuellen Version der Simulation wird lediglich die erste Optimierungsstufe genutzt, da eine Behandlung der beiden genannten Szenarien bislang nicht vorgesehen ist und somit einer fortführenden Arbeit unterliegen würde.

Die Realisierung der Bekanntgabe von in Routing-Tabellen unbekannten Zielen geschieht wie folgt. Ist ein Ziel unbekannt, werden in der entsprechenden Routing-Tabelle alle bekannten zusammenführenden Weichen (Join) auf das gewünschte Ziel hin untersucht. Sollte hier mehr als ein Pfad als Ergebnis geliefert werden, so wird wieder nach den zuvor genannten Kriterien der „bessere“ Pfad (min. Anzahl Bahnhöfe auf Route und kürzester Pfad) gewählt und nun zur Routing-Tabelle hinzugefügt.

Durch die getroffenen Design-Annahmen bezüglich des zusammenhängenden Graphen, in dem jeder Punkt von jedem Punkt aus erreichbar sein muss, ist garantiert, dass diese Prozedur einen erfolgreichen Pfad zurückliefert.

Der Hintergrund hierzu sei im Folgenden beschrieben:
Ist ein Ziel weder in der aktuellen Routing-Tabelle noch in einer der bekannten Joins dieser Routing-Tabelle vorhanden, so bedeutet dies, dass das Ziel vor dem aktuellen Knoten bezüglich des Startknotens liegt (Achtung: der Startknoten für die Routing-Initialisierung und für das $jABC$ müssen nicht identisch sein!). Hieraus folgt unmittelbar, dass der kürzeste Weg, um zu diesem unbekannten Ziel zu gelangen, über den Startknoten führt. Daher wird für ein solches Ziel der Pfad zum Startknoten an den Pfad vom Startknoten zum entsprechenden Ziel angehängt und in der aktuellen Routing-Tabelle gespeichert.

Der Startknoten selbst besitzt eine vollständige Routing-Tabelle, so dass hier jedes Ziel auf jeden Fall bekannt ist. Dies ist sichergestellt, da er der letzte Knoten ist, der beim DFS-Durchlauf aktualisiert wird, und daher ein vollständiges Abbild des Graphen enthält.

Betrachtung der vier Fahrstreckentypen im Detail

Die vier Grundtypen des Schienennetzes bestehen aus der normalen Schiene, dem Bahnhof, der zusammenführenden Weiche und der verzweigenden Weiche. In Bezug auf das Routing spielen die ersten drei keine Rolle für die Zielfindung, da hier immer nur ein Nachfolger in Frage kommt. Es werden jedoch alle Typen in die Routing-Tabellen mit aufgenommen, da dies zur Aktualisierung der Tabellen nötig ist. Nur so ist eine Kostenermittlung eines Pfades und die Überprüfung auf enthaltene Bahnhöfe möglich.

Die Routing-Tabelle der verzweigenden Weiche muss selbstverständlich erstellt werden, da aus ihr die Entscheidung über die einzuschlagende Richtung eines EHB-Fahrzeugs auszulesen ist.

Die Unterscheidung der Weichen bezüglich der Anzahl ihrer ein- oder ausgehenden Gleise spielt für den Algorithmus ebenfalls keine Rolle. Es muss an dieser Stelle lediglich für den DFS-Durchlauf sichergestellt sein, dass bei einer verzweigenden Weiche jede Verzweigung betrachtet wird und dass bei einer vereinigenden Weiche in die Richtung zurückgeleitet wird, aus der man zuletzt gekommen ist (korrekter Rekursionsrücksprung). In der vorliegenden Simulation wurde dies für Weichen mit bis zu drei Ein- bzw. Ausgängen implementiert.

Next-Hop-Ermittlung

Während der laufenden Simulation wird an normalen Schienen und an zusammenführenden Weichen einfach an den Nachfolger weitergeleitet. An Bahnhöfen wird eine gesonderte Betrachtung des EHB-Fahrzeugs vorgenommen, welche zur Be- bzw.

Entladung dient, auf die an dieser Stelle aber nicht weiter eingegangen wird. An verzweigenden Weichen wird bei Bedarf (Eintreffen eines EHB-Fahrzeugs) anhand des Zieles aus der entsprechenden Routing-Tabelle der Next-Hop ausgelesen und das Fahrzeug an diesen weitergeleitet.

3.5.4 Beispiel der Initialisierung der Routing-Tabellen am einfachen Schienennetz

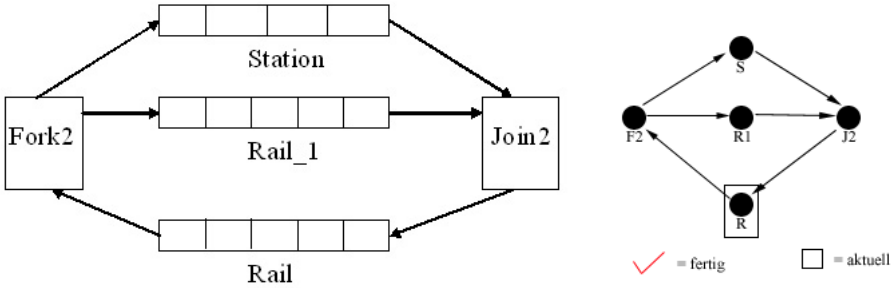


Tabelle 3.2: Schienennetz mit abstraktem Graph für den DFS-Algorithmus.

Die Initialisierung der Routing-Tabellen eines einfachen Schienennetzes (siehe Tabelle 3.2) wird hier nun beispielhaft dargestellt.

Als Startknoten wird das SIB `Rail` ausgewählt. Als Erstes durchläuft der DFS-Algorithmus den Graphen so lange, bis er auf einen bereits besuchten Knoten trifft. Zunächst betrachtet der Algorithmus also `Fork2`. Da dieses SIB noch nicht betrachtet wurde, wählt der Algorithmus zufällig eine Kante aus, in unserem Beispiel nehmen wir die Kante zu `Station`. Da `Station` noch nicht besucht wurde, wird der Nachfolger `Join2` betrachtet, und dieser liefert `Rail` (siehe Abbildung 3.7).

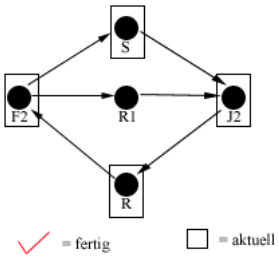
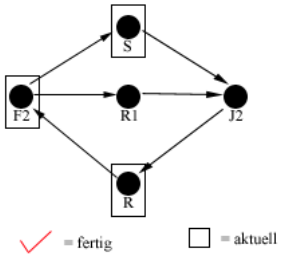


Abbildung 3.7: DFS-Fortschritt bis zum ersten bereits besuchten Knoten.

Mit `Rail` betrachten wir zum ersten mal ein SIB, welches bereits besucht wurde. Wir leiten die Routing-Tabelle von `Rail` an `Join2` zurück. Da die Routing-Tabelle

von `Rail` noch keine Einträge enthält, fügt `Join2` lediglich einen Eintrag zu seinem jetzt bekannten Nachfolger `Rail` in seine Routing-Tabelle ein und übergibt diese Routing-Tabelle nun seinem Vorgänger, also `Station`. Dieser nimmt die Routing-Tabelle, gleicht sie mit seiner eigenen (noch leeren) ab und fügt `Join2` als zusätzlichen Eintrag hinzu. Die bis hierhin erstellten Routing-Tabellen sind in Tabelle 3.3 dargestellt.

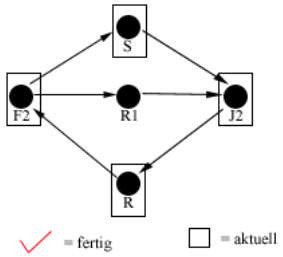
Es sei an dieser Stelle nochmals erwähnt, dass in jedem Pfad einer Routing-Tabelle die erste Position ein Ziel und die letzte Position den Next-Hop zu diesem Ziel darstellt.



Rail	**Join2**	**Station**	
-	Pfad 0:	Pfad 0:	Pfad 1:
	Rail	Rail	Join2
		Join2	

Tabelle 3.3: Fortschritt des DFS und der Routing-Tabellen.

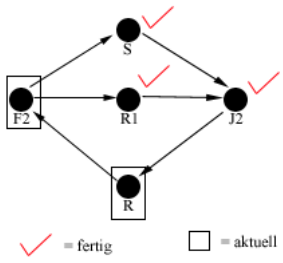
`Station` liefert seine Routing-Tabellen jetzt an `Fork2`, deren Einträge dann wie in Tabelle 3.4 aussehen.



Fork2		
Pfad 0:	Pfad 1:	Pfad 2:
Rail	Join2	Station
Join2	Station	
Station		

Tabelle 3.4: Fortschritt des DFS und Routing-Tabelle von `Fork2`.

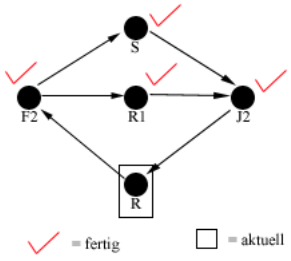
Da `Fork2` noch nicht alle ausgehenden Kanten abgearbeitet hat, verweist es nun auf `Rail_1` und dieses wiederum auf `Join2`. Da `Join2` bereits besucht wurde, verweist es nicht wieder an `Rail` sondern liefert sofort seine Routing-Tabelle an `Rail_1` zurück. Die unverändert bleibende Routing-Tabelle von `Join2` und die nun aktualisierte Routing-Tabelle von `Rail_1` zeigt Tabelle 3.5.



Join2	**Rail_1**	
Pfad 0:	Pfad 0:	Pfad 1:
Rail	Rail	Join2
	Join2	

Tabelle 3.5: DFS Fortschritt mit entsprechenden Routing-Tabellen-Einträgen von `Join2` und `Rail_1`.

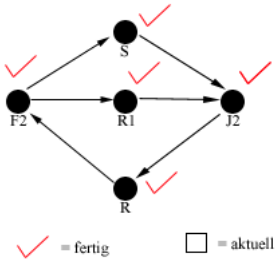
`Rail_1` wurde nun ebenfalls bereits besucht und liefert seine Routing-Tabelle an `Fork2` weiter, welches nun seine bereits bestehende Tabelle mit der neuen abgleichen muss. Da an dieser Stelle in beiden Tabellen ein Eintrag für das Ziel `Rail` existiert, werden die beiden Pfade miteinander verglichen. Obwohl der bereits gespeicherte Pfad über `Station` nicht länger ist, wird nun der neue Pfad über `Rail_1` ausgewählt, da dieser Pfad weniger (hier keine) Bahnhöfe enthält. Daher wird der alte Pfad mit dem neuen Pfad von `Rail_1` überschrieben. Sobald alle Pfade auf diese Weise überprüft wurden, ergibt sich für das Routing von `Fork2` die Tabelle 3.6.



Fork2			
Pfad 0:	Pfad 1:	Pfad 2:	Pfad 3:
Rail	Join2	Station	Rail_1
Join2	Rail_1		
Rail_1			

Tabelle 3.6: DFS-Fortschritt und die Routing-Tabelle von `Fork2` nach der Aktualisierung.

Als letztes liefert `Fork2` seine Tabelle an `Rail` zurück, welche nun ebenfalls aktualisiert wird. Damit wurde jedes Streckenelement betrachtet und `Rail` enthält nun sämtliche Einträge für jedes beliebige Ziel (siehe Tabelle 3.7).



Rail				
Pfad 0:	Pfad 1:	Pfad 2:	Pfad 3:	Pfad 4:
Rail	Join2	Station	Rail_1	Fork2
Join2	Rail_1	Fork2	Fork2	
Rail_1	Fork2			
Fork2				

Tabelle 3.7: DFS nun vollständig und die Routing-Tabelle von `Rail` nach der Aktualisierung.

3.6 Zeitfaktoren

3.6.1 Einleitung

Anforderung an die Simulation ist es, eine Aussage darüber treffen zu können, ob eine gegebene Batch innerhalb einer Zeitschranke realisierbar ist. Folglich spielt der Faktor Zeit eine besondere Rolle und erfordert dementsprechend besondere Aufmerksamkeit. Die folgenden Abschnitte befassen sich mit der Analyse der verschiedenen Zeitfaktoren und deren Realisierung im Modell.

3.6.2 Zeitfaktor im Lagersystem

Bei der Analyse des Lagersystems lässt sich erkennen, dass sich die Zeiten in zwei große Abschnitte unterteilen lassen. Neben global geltenden Faktoren, wie z. B. der Zeit für das Be- und Entladen oder das Auslagern einer Palette aus dem Hochregallager, gibt es für die Fahrzeugbewegungen auf der Strecke Zeitfaktoren, die für jedes Fahrzeug individuell festgelegt werden können.

Globale Faktoren

Im vorliegenden Lagersystem findet ein Großteil der Vorgänge automatisiert und häufig auch in wiederkehrenden, teilweise relativ kurzen Abständen statt. Um beispielsweise eine Palette aus dem Hochregallager auszulagern und diese zur Übergabestation zu transportieren, benötigt das Regalbediengerät im Mittel immer die gleiche Zeitspanne. Um diese Vorgänge in der Simulation abzubilden, kann der Anwender global Zeiten für diverse Vorgänge festlegen. Diese Vorgabe wird auf alle Vorgänge gleichen Typs übertragen. Legt der Benutzer beispielsweise die Zeit, die benötigt wird, um einen einzelnen Artikel zu kommissionieren, auf t Sekunden fest, so dauert

der Kommissioniervorgang eines einzelnen Artikels an *allen* Kommissionierstationen t Sekunden.

In der Simulation werden folgende Faktoren berücksichtigt:
Zeit, die vergeht

- vor Beginn des Kommissioniervorgangs (Sicherung des Fahrzeugs),
- nach Beendigung des Kommissioniervorgangs (Sicherung der Ladung),
- um eine Palette in das HRL einzulagern/auszulagern,
- beim Be-/Entladevorgang eines Fahrzeugs,
- um eine Leerpalette zu Entladen,
- um einen einzelnen Artikel zu kommsissionieren.

Individuelle Faktoren

Im Gegensatz zu den gerade erwähnten globalen Faktoren können bei den individuellen Faktoren beliebig viele verschiedene Werte auftreten. Jeder Fracht kann eine bestimmte Geschwindigkeit zugewiesen werden. So kann z. B. eine verschlossene Kiste schneller befördert werden als eine Sammlung loser Regalbretter. Daher ist es möglich, jedes Fahrzeug mit einer individuellen Geschwindigkeit auf dem Schienensystem fahren zu lassen. Leere Fahrzeuge bewegen sich alle mit derselben Geschwindigkeit. Die Geschwindigkeit eines Fahrzeugs ist jedoch nicht alleine von seiner Fracht abhängig, sondern kann auch durch die Strecke beeinflusst werden. So gibt es z.B. Streckenabschnitte, bei dessen Befahrung Gefahr für Personen bestehen kann und das Fahrzeug diese deshalb nur langsam durchfahren darf. Die aktuelle Geschwindigkeit eines Fahrzeugs ergibt sich daher aus dem Minimum der durch die Fracht vorgeschriebenen Geschwindigkeit und der auf der Strecke erlaubten Geschwindigkeit.

3.6.3 Design-Annahmen

Aufgrund der Komplexität des Lagersystems ist es nahezu unmöglich alle, Zeitfaktoren umfassend in der Simulation abzubilden. Daher beinhaltet die Simulation lediglich die charakteristischsten Vorgänge. Weiterhin werden viele der Vorgänge lediglich durch Näherungswerte beschrieben. Beispielsweise wird die Zeit, die ein Fahrzeug benötigt vom Moment des Erreichens der Kommissionierstation, bis zu dem Zeitpunkt, an dem es seine Fahrt fortsetzt, durch folgende Formel abgeschätzt:

$$t_{\text{vor Beginn}} + t_{\text{einzelner Kommissioniervorgang}} * \#\text{Artikel} + t_{\text{nach Beendigung}}$$

Dabei beschreibt $t_{\text{vor Beginn}}$ die Zeit die vergeht um das Fahrzeug in der Station zu sichern und gegebenenfalls noch verpackte Ware zu öffnen. $t_{\text{einzelner Kommissioniervorgang}} * \#\text{Artikel}$ beschreibt die Zeit des eigentlichen Kommissioniervorgangs, um die gegebene Anzahl von Artikeln zu entladen. Zuletzt beschreibt $t_{\text{nach Beendigung}}$ die Zeitspanne, um die Ware wieder zu sichern und das Fahrzeug wieder frei zu geben. Dies zeigt deutlich, dass sich in dem Modell Abweichungen zum realen System ergeben können. Es lässt sich weder simulieren, dass verschiedene Kommissioniervorgänge unterschiedlich lange dauern, noch dass die Sicherung der Ladung unterschiedlich viel Zeit in Anspruch nehmen kann.

Während bei den Fahrzeuggeschwindigkeiten eine individuelle Anzahl möglich ist, beschränkt sich die mögliche Anzahl der Streckengeschwindigkeiten in der Simulation auf vier.
Unterscheidung zwischen Geschwindigkeit auf

- einer Schnellfahrstrecke
- einem normalen Streckenabschnitt
- einem Streckenabschnitt mit Personengefährdung
- einem Streckenabschnitt für gesonderte Anforderungen

3.6.4 Zeitsimulation mit Ticks

Zur Integration von Zeit in der Simulation haben wir uns für die Implementierung mittels diskreter Zeit entschieden. Auf diese Weise können Zeitintervalle, auf denen keine Aktionen stattfinden, durch die Simulation übersprungen werden. Dies lässt sich am Beispiel der Fahralgorithmik zeigen. Ein Fahrzeug befindet sich hierbei nicht absolut an einer bestimmten Position, sondern innerhalb einer Blockstelle. Dabei ist lediglich interessant, zu welchem Zeitpunkt ein Fahrzeug in eine Blockstelle einfährt und nach welcher Zeitspanne es die Blockstelle wieder verlässt. Der Zeitraum zwischen diesen beiden Ereignissen ist für die Simulation nicht relevant und kann daher übersprungen werden. Hierzu musste eine fest definierte Zeiteinheit in die Simulation eingeführt werden, der *Tick*. Dabei wird ein Tick zu Beginn der Simulation mit einem konstanten Zeitwert belegt. Dazu müssen alle im System vorkommenden Geschwindigkeiten und Zeiten auf die Einheit Sekunden umgerechnet werden. Dann lässt sich der Wert eines Ticks mittels des Algorithmus zur Berechnung eines größten gemeinsamen Teilers nach Euklid berechnen. Der auf diese Weise berechnete Tick stellt hierbei die kleinste benötigte Einheit während der Simulation dar.

Die einzigen Objekte der Simulation, für welche Zeit verwaltet wird, stellen die Fahrzeuge dar. Sie erhalten nach jeder ihrer Fahrbewegungen einen Basiswert, der die Zeit, die für das Durchfahren der nächsten Blockstelle benötigt wird, darstellt.

Hinzu kommen Aufschläge für eventuelle Aktionen. Befindet sich das Fahrzeug z.B. in einer Kommissionierzone um dort entladen zu werden, so bekommt es die für den Entladevorgang benötigte Zeit auf den Basiswert aufaddiert. Um zu Beginn einer jeden Runde feststellen zu können, wie viel Zeit simuliert werden kann, werden alle Fahrzeuge in einer Liste aufsteigend nach ihren Ticks sortiert. Damit gibt das erste Element dieser Liste immer die Zahl Ticks an, die es zu simulieren gilt.

3.6.5 Bestimmung eines Ticks

Grundlage für die Berechnung eines Ticks ist, das zuvor alle Zeiteinheiten einheitlich auf Sekunden umgerechnet werden. Während die Zeiten für Be- und Entladevorgänge bereits in Sekunden vorliegen, müssen die verschiedenen Fahrzeuggeschwindigkeiten noch von m/s umgerechnet werden. Dabei dient die Länge einer Blockstelle als Berechnungsgrundlage. Dies geschieht mittels der einachen Formel $t_{Blockstelle} = \frac{s_{Blockstelle}}{v_{Fahrzeug(Strecke)}}$. Nachdem alle Zeiten vorliegen, kann nun die eigentliche Umrechnung in Ticks erfolgen.

```
EUCLID(a,b)
  solange b <> 0
    wenn a > b
        dann a --> a - b
        sonst b --> b - a
  return a
```

3.7 Vorgänge im Lager

3.7.1 Design-Annahmen

Als zentraler Bestandteil der Simulation wurden auch im Bereich der Ladungsverwaltung entsprechende Abstraktionen eingeführt. Um die gesamten Prozesse besser simulieren zu können, wurden bei der Simulation der *Kommissionierzonen*, *Be-/Entladezonen*, beim *Hochregallager* sowie bei der *Ladung* selbst einige Annahmen, welche die Realität nachstellen und gleichzeitig schnell zu simulieren sind, gemacht.

Die einzelnen Bereiche werden jeweils durch ein entsprechendes SIB, gekoppelt an ein Bahnhof-SIB, gekennzeichnet. Das Hochregallager wird durch einzelne, an einer *Lade-/Entladezone* angeschlossene *Hochregallagerzeile* repräsentiert. Diese Darstellung geschieht intern und ist im Modell nicht direkt ersichtlich.

Wichtige Bereiche des Gesamtlagers

Durch die SIBs ergibt sich eine logistische Aufteilung des Lagers in folgende Teile, die beim Transfer der Ladung wichtige Rollen spielen:

Die Be-/Entladezone

Sie ist der Ein- bzw. Austrittspunkt für Ladung vom bzw. zum Schienennetz aus einer *Hochregallagerzeile*. Hier setzt das Regalbediengerät die Ladung für ein Fahrzeug ab, bzw. nimmt sie wieder von einem Fahrzeug auf.

Der Kommissionierbereich

Im *Kommissionierbereich* werden die Paletten, die aus dem Hochregallager ausgelagert und durch je ein Fahrzeug hierher befördert werden, auf die Kommissionierpaletten kommissioniert. In jedem Bahnhof können bis zu fünf Fahrzeuge zur Entnahme geparkt werden, wobei die Entnahme der Waren nach dem FIFO-Prinzip erfolgt. In jedem Kommissionierbereich stehen gleichzeitig sechs Kommissionierpaletten zur Verfügung. Das eigentliche Kommissionieren erfolgt manuell. Die angeforderten Artikel werden solange bereitgestellt, bis die benötigten Warenmengen entnommen sind.

3.7.2 Der Weg der Ladung

Jede Ladung durchläuft, nachdem durch die Batch vorgegeben wurde, welche Ladung erzeugt werden soll, den gleichen Ablauf:

- Die Ladung wird erzeugt und anschließend aus der entsprechenden *Hochregallagerzeile* ausgelagert,
- Im entsprechenden *LoadingBay-SIB* zur Abholung bereitgestellt,
- Anschließend wird die Ladung abgeholt und es erfolgt ein Transport durch ein Fahrzeug,
- Nach erreichen wird in den entsprechenden *Kommissionierbereich* ein (Teil-)Abladen vollzogen,
- Danach erfolgt der Rücktransport von Anbruchpaletten zum entsprechenden *U*nLoadingBaySIB oder die Abgabe der leeren Palette am *Empty-Pallet-SIB*,
- Sollte das Fahrzeug noch Ladung tragen, wird diese Restladung der Wiedereinlagerung in der entsprechenden *Hochregallagerzeile* zugeführt.

Aufschlüsseln der Batch in Palettenaufträge

Die *Batch* wird von der zentralen Lagerverwaltungsinstanz, dem *DataHandler*, in Palettenaufträge aufgeteilt. Die Batch-Aufträge werden einzeln als Kommissionierpaletten ins System eingepflegt. Danach erfolgt die Abarbeitung in folgenden Schritten:

- Ein Programmteil ruft die Verteilung von einer Palette auf die *CommissioningBay-SIB*s auf.
- Für jede verteilte Palette wird der entsprechende Batch-Auftrag in die einzelnen Artikel-Posten zerlegt.
- Die Posten werden in einer internen Datenstruktur gesammelt. Eine Zuordnung von Ladung zu einer bestimmten Palettenposition bleibt erhalten.
- Gleiche Artikelsorten auf unterschiedlichen Kommissionierpaletten werden zusammengefasst.
- Die entsprechende Einzelwarenpaletten, also Paletten mit nur einer Artikelsorte, werden in der entsprechenden *Hochregallagerzeile* erzeugt. Hierbei werden zuerst im Lager vorhandene Anbruchpalleten wieder ausgelagert. Sollten keine Anbruchpaletten mehr vorhanden sein, so werden neue Paletten gemäß der Artikelbasismenge erzeugt.

Kommissionierungsprozess

Der Kommissionierungsprozess beschreibt die Verteilung der Ladung auf die einzelnen Kommissionierpaletten. Jede Palette besitzt eine Identifikationsnummer, welche mit der Ladung gespeichert wird. Für jede ausgelagerte Artikelpalette steht also bereits fest, welche Kommissionierpaletten durch sie beliefert werden. Anhand dieser Merkmale erfolgt eine eindeutige Zuordnung von Ladung zu Kommissionierpaletten.

3.7.3 Implementierung

Die Initialisierung

Während der Initialisierung werden die *Be-/Entladezone*n erstellt, die internen Datenstrukturen werden aufgebaut und die Artikelbasismengen werden aus einer vorher erstellten XML-Datei geladen.

Be-/Entladezone

Sie besteht aus zwei hintereinander angeordneten Bahnhöfen. Der in Fahrtrichtung vordere Bahnhof ist mit einem *UnloadingBay-SIB* gekennzeichnet. Dies ist der Eintrittspunkt für Ladung in die entsprechende *Hochregallagerzeile*. Der in der Realität vorhandene Platz in einer Einlagerungsstation wird durch die Anzahl der für Ladung vorhandenen Plätze im *UnloadingBay-SIB* repräsentiert. Die genauen Beschreibungen der beiden SIBs sind in Kapitel 3.2 zu finden.

Der hintere Bahnhof ist mit einem *LoadingBay-SIB* versehen. Dies ist der Ladepunkt für Ladung aus der entsprechenden *Hochregallagerzeile*. Auch hier wird der real vorhandene Platz durch die Plätze im *LoadingBay-SIB* dargestellt.

Ist ein UPPAAL-Modell geladen worden, so wird die *Be-/Entladezone* angepasst an den EBZ-Kasten des Austauschmodells erstellt (hier befinden sich noch weitere SIBs zwischen Be- und Entladepunkt).

Hochregallagerzeile

Diese wird nicht durch ein SIB repräsentiert, sondern automatisch bei der Initialisierung der *Be-/Entladezone*n erstellt. Je einer *Be-/Entladezone* wird eine *Hochregallagerzeile* zugeordnet. Diese ist als Black-Box ausgeführt, das heißt, Ladung wird mit einer durchschnittlichen Verzögerung Ein- und Ausgelagert (im Einfach- sowie im Doppelspiel). Weitere interne Vorgänge werden nicht für die Simulation berücksichtigt. Gleichzeitig wird davon ausgegangen, dass für jede Ware beliebig viele neue Paletten zur Verfügung stehen. Jede Runde wird jeder *Hochregallagerzeile* mitgeteilt, wie viele Ticks vergangen sind. Daraus wird berechnet, welche Ein-/Auslagerungsvorgänge als nächstes stattfinden bzw. stattgefunden haben.

Das Beladen in der Be-/Entladezone

Dieses verläuft anhand folgenden Musters:

- Es wird geprüft ob sich ein Fahrzeug an Pufferposition 0 des *LoadingBay-SIB*s befindet
- Besitzt dieses Fahrzeug noch keine Ladung?
- Falls ja, wird das Fahrzeug mit der ersten Ladung in der Warteliste beladen
- Die entsprechenden Ticks werden auf die Fahrzeugticks addiert
- Abschließend wird die Warteliste aktualisiert.

Das Entladen in der Be-/Entladezone

Dieses verläuft anhand folgenden Musters:

- Es wird geprüft, ob sich ein Fahrzeug an Pufferposition 0 des *UnloadingBay-SIB*s befindet
- Das Ziel muss dem *UnloadingBay-SIB* entsprechen
- Das Fahrzeug muss Ladung besitzen
- Sind obige Vorraussetzungen erfüllt, so wird das Fahrzeug entladen

- Die entsprechenden Ticks werden auf die Fahrzeugticks addiert
- Danach wird die Warteliste aktualisiert

Kommissionierbereich

Dieser wird durch einen Bahnhof, der mit einem *CommissioningBay-SIB* versehen wird, gekennzeichnet.

Das Entladen im Kommissionierbereich

findet statt, wenn folgende Kriterien erfüllt sind:

- Das Fahrzeug befindet sich an Pufferposition 0.
- Das Ziel der Ladung ist der Bahnhof, in dem sich das Fahrzeug befindet.
- Die entsprechenden Paletten für die Ladung sind vorhanden.

Damit kann der Entladevorgang eingeleitet werden. Die Ladungen werden transferiert und die Ticks für das Abladen dieser Menge berechnet. Die Ticks des Fahrzeugs werden um diesen errechneten Wert erhöht.

Nach jeder Entladung eines Fahrzeugs in einem Bahnhof wird geprüft, ob die restlichen Waren auf der Palette dieses Fahrzeugs an einem weiteren Bahnhof benötigt werden. Wenn dies nicht der Fall ist, wird die Palette als eine angebrochene Palette zurück zum Lager transportiert.

Jedes *CommissioningBay-SIB*s enthält eine Liste in der die Artikel und Mengen, welche die angeschlossene Kommissionierstation benötigt, vermerkt sind. Die auf einem Fahrzeug vorhandene Ladung wird mit dieser Liste und der Liste der zu bedienenden Paletten abgeglichen. Bei Bedarf wird das Fahrzeug entladen. Die entladene Menge wird entsprechend aus der Wunschliste ausgetragen. Erfüllte Ladeaufträge werden in einer zweiten Liste verwaltet. Diese enthält alle bereits erfüllten Ladeanforderungen. Damit wird der Überblick über bereits erfüllte und noch zu erfüllende Ladeanforderungen erhalten.

Leerpalettenabgabe

Die Entsorgung der leeren Paletten geschieht in dem so genannten *Empty-Pallet-SIB*. Dort wird die leere Palette aus dem Fahrzeug entfernt und das Fahrzeug zum Warten auf weitere Aufträge in den *Holding-Track* geschickt.

3.8 Tests

3.8.1 Motivation

Normalerweise müssen eine Menge Tests während und nach der Programmentwicklung durchgeführt werden. Das Ziel besteht darin, die Qualität des Programms zu beurteilen und eventuell zu verbessern. Der Nachweis der Korrektheit bedarf eines vollständigen Tests, nämlich eines Tests mit allen möglichen Eingangsbelegungen in allen möglichen Kombinationen und Reihenfolgen.

3.8.2 Konzepte

Um die Tests effizient und flächendeckend innerhalb eines bestimmten Zeitraums durchführen zu können, müssen wir uns auf die wesentlichen Dinge konzentrieren. Einerseits kann die Richtigkeit der allgemeinen *get-* und *set-* Methoden vorausgesetzt werden, andererseits sollten die folgenden Bereiche intensiv mit jUnit-Tests [JUN] getestet werden:

- Fahralgorithmus,
- Routing,
- Datahandler,
- Klasse für die verschiedenen Vorgänge (z. B: Entladevorgang oder Beladevorgang).

Die Tests sollten nach dem Prinzip *Bottom-Up* und in zwei Phasen durchgeführt werden. Das heißt, dass die Methoden, auf deren Funktionalitäten die Simulation basiert, in der ersten Phase und zwar, wenn möglich, lokal getestet werden sollten. Wenn die Richtigkeit all dieser Methoden durch Tests bestätigt werden konnte, können wir davon ausgehen, dass die wichtigen grundlegenden Leistungen, wie z. B die Anpassung der Geschwindigkeit des Fahrzeuges, die Berechnung der Ticks des Fahrzeuges usw. von dem Programm gewährleistet werden können. Basierend auf diesen bestätigten Leistungen können dann in der zweiten Phase die Testmodelle gebaut und die möglichen Belegungen gezielt eingesetzt werden, um die Funktionen der komplexeren Vorgänge zu testen. Alle Tests sollten so gestaltet sein, dass sie sich jederzeit wieder automatisch durchführen lassen. Es sollten möglichst wenige und einfache Modelle gebaut und gespeichert werden. Im jUnit-Test sollte das entsprechende Modell über *jABC* aufgerufen werden und die Simulation läuft dann automatisch ab. Soll-Ergebnis und Ist-Ergebnis werden durch `assert` in Verbindung gesetzt. Nach der Durchführung des jUnit-Tests wird das Modell wieder entfernt und *jABC* wieder geschlossen. Zur Visualisierung kann man auch *jABC* nach der Durchführung des Tests aktiviert lassen und das Ergebnis direkt beurteilen.

3.8.3 Testmodelle

In Tests werden Modelle für unterschiedliche Testzwecke gebaut. Abbildung 3.8 zeigt eine Beispiel davon. In diesem Modell befinden sich 2 `CommissioningBay` und jeweils 1 `PaletteunloadingBay`, 1 `UnloadingBay`, 1 `LoadingBay` und 1 `HoldingTrack`.

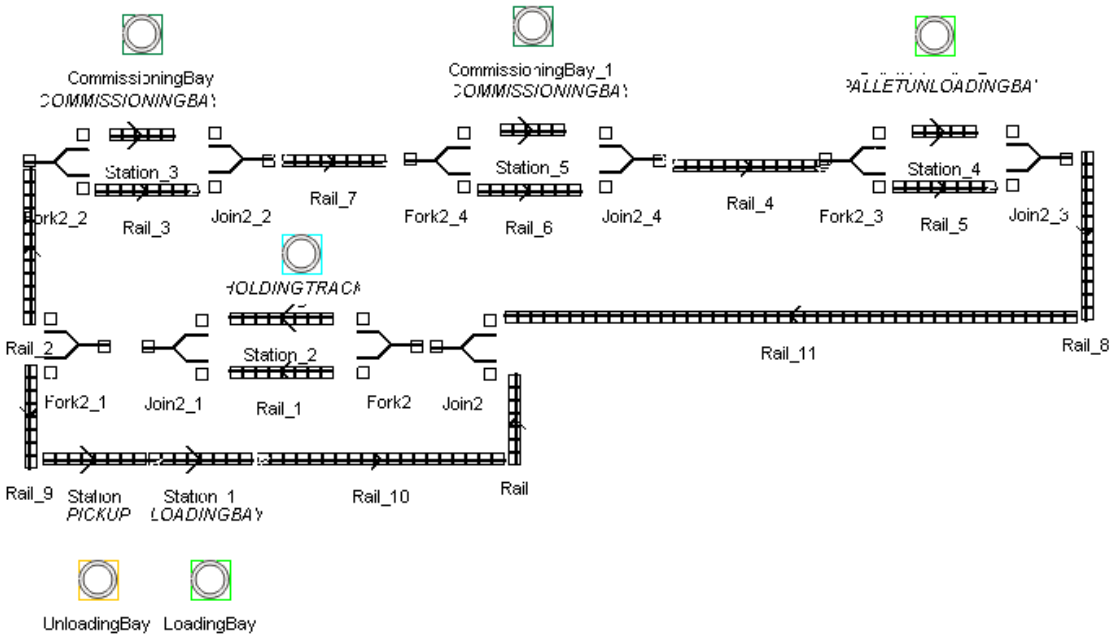


Abbildung 3.8: Beispiel für Testmodell.

Alle Modelle werden gespeichert und können bei der Durchführung der Tests entsprechend abgerufen und initialisiert werden.

3.8.4 Testen der grundlegenden Funktionalitäten

In der ersten Phase werden, wie schon erwähnt, die grundlegenden Funktionalitäten des Programms getestet. Diese müssen folgende Leistungen gewährleisten:

1. Erzeugen und Setzen des Fahrzeuges,
2. Aktualisieren des Zustands des Fahrzeuges bei der Bewegung,
3. Zurückliefern des Zustands des Fahrzeuges,
4. Anpassen der Geschwindigkeit des Fahrzeuges,
5. Berechnen der Ticks des Fahrzeuges,
6. Setzen der Ticks des Fahrzeuges,

7. Bewegen des Fahrzeuges,
8. Übergeben des Ziels des Fahrzeuges,
9. Erzeugen der Ladungen und
10. Aufteilen der Ladungen.

Es werden mehrere Testfälle benötigt, um die Korrektheit der grundlegenden Funktionalitäten zu überprüfen. Bei allen Testfällen werden die Soll-Ergebnisse vorberechnet. Einige Testfälle werden im Folgenden erläutert:

Test für die Geschwindigkeitsticks des Fahrzeuges: Testet, ob ein Fahrzeug während der Simulation auf einer Fahrstrecke die richtigen Ticks für seine Geschwindigkeit bekommen kann. Es wird im Test zuerst ein Fahrzeug erzeugt. Dieses Fahrzeug wird dann jeweils in die Zustände „keine Ladung“, „mit leerer Ladung“ und „mit geschlossener Ladung“ gesetzt. Es wird geprüft, ob es immer die richtigen Ticks bekommt.

Test für die Statusaktualisierung des Fahrzeuges: Testet, ob der Status eines Fahrzeuges nach einer Bewegung während der Simulation immer richtig aktualisiert wird. Es wird im Test ein Fahrzeug erzeugt. Die zu testende Funktion wird dann mit verschiedenen möglichen Parametern am Fahrzeug verwendet.

Test für die Zielübermittlung des Fahrzeuges: Testet die Richtigkeit der Übermittlung eines Ziels an ein Fahrzeug. Es wird im Test ein Fahrzeug erzeugt. Bei diesem Fahrzeug wird dann bei den Zuständen „keine Ladung“, „mit leerer Ladung“, „mit geschlossener Ladung“ und „mit angebrochener Ladung“ getestet, ob es jeweils immer das richtige Ziel bekommt.

Test für die Übermittlung des aktuellen SIB des Fahrzeuges: Testet die Funktion, welche das SIB, worauf gerade das zu testende Fahrzeug steht, zurückliefern muss. Im Test wird Modell 3 benutzt. Es werden drei Fahrzeuge auf bestimmte Positionen der unterschiedlichen SIBs gesetzt. Das Modell wird initialisiert und simuliert. Es wird geprüft, ob das richtige SIB von jedem Fahrzeug zurückgeliefert wird.

Test für das Hinzufügen einer Palette: Testet das Hinzufügen einer Palette zur Batch. Ein Palettenobjekt wird erzeugt und hinzugefügt. Soll-Ergebnis ist das anschließende Vorhandensein in der Batch-Liste. Anschließend wird ein zweites Testobjekt hinzugefügt. Dieses muss sich am Ende der Batch befinden.

Test für das Wiederauffüllen der Kommissionierstationen: Testet das Wiederauffüllen der Kommissionierstationen mit Paletten aus der Batch. Die Batch kann leer, mit einer Kommissionierpalette oder mit mehreren befüllt sein. Die Kommissionierstationen können leer, teilgefüllt oder voll sein. Alle essenziellen Kombinationen werden generiert. Ihre vorab berechnete Soll-Verteilung wird jeweils überprüft.

Test für das Erzeugen der Ladungsstücke: Testet das Erzeugen von Ladungsstücken aufgrund von Paletten der Batch. Paletten werden erzeugt und übergeben. Anschließend wird daraus Ladung erzeugt und verteilt. Die Soll-Verteilung ist vorgegeben.

Test für das Auslösen von Ein- und Auslagerungsoperationen: Testet das Auslösen von Ein- bzw. Auslagerungsoperationen im Hochregallager. Ladung kann einzeln oder im Doppelspiel ein- bzw. ausgelagert werden. Der zeitliche Rahmen für jede Operation ist festgelegt. Gleichzeitig sollte ein Doppelspiel stattfinden, falls möglich. Die Kombinationen werden mit entsprechender Ware nacheinander erzeugt und die Zeiten entsprechend vorgegeben. Sollwert ist eine Auslagerung im richtigen Verfahren sowie zum richtigen Zeitpunkt.

Test für die Anzahlübermittlung der zu kommissionierenden Artikel: Testet die Funktion, welche die richtige Anzahl der Artikel, die an einer Kommissionierzone benötigt werden, zurückliefern muss. Im Test wird eine neue CommissioningPallet erzeugt. Auf dieser CommissiningPallet werden verschiedene Artikel und deren verschiedene Anzahlen hinzugefügt. Eine get-Methode liefert die Anzahl eines Artikels wieder zurück. Dadurch kann geprüft werden, ob alle Artikel richtig auf diese neue CommissioningPallet hinzugefügt wurden.

Test für die Anzahlübermittlung der zu entladenden Artikel: Testet die Funktion, welche den Artikel entsprechend der gegebenen Anzahl entladen muss. Im Test wird eine CommissioningPallet mit verschiedenen Artikel erzeugt . Es wird dann eine bestimmte Anzahl von Artikeln per getesteter Funktion entladen.

Test für die Berechnung des ggT: Testet die Funktion, welche den ggT aller Zeitfaktoren und Ticks für die Fahrzeuge entsprechend ihren Geschwindigkeiten berechnet. Im Test wird durch diese Funktion der ggT aller Zeitfaktoren mit den Vorgabewerten berechnet. Anschließend werden mit diesem ggT die Ticks für einige Geschwindigkeiten umgerechnet.

Test für die Geschwindigkeitseinschränkung des Fahrzeuges: Testet die Funktion, welche garantieren muss, dass ein Fahrzeug immer die kleinste, erlaubte Geschwindigkeit wählt, wenn es sich auf einer Fahrstrecke mit Geschwindigkeitsbe-

schränkung bewegt. Durch diese Funktion werden im Test die Ticks der unterschiedlichen Geschwindigkeiten mit den Ticks, welche eine Fahrstrecke erlaubt, verglichen. Es wird geprüft, ob diese Funktion, entsprechend der Testsituation, die richtigen Ticks ausgibt.

3.8.5 Testen des Fahralgorithmus

Der Fahralgorithmus beschäftigt sich mit der Bewegung der Fahrzeuge auf der Fahrstrecke. Dabei werden sowohl die Geschwindigkeit der Fahrzeuge als auch die erlaubte Geschwindigkeit der Fahrstrecke berücksichtigt. Der Fahralgorithmus muss folgende Anforderungen erfüllen:

1. Auf gleicher Fahrstrecke darf das schneller fahrende Fahrzeug das langsamer fahrende Fahrzeug nicht überholen.
2. Normalerweise bewegt sich ein leeres Fahrzeug am schnellsten, ein Fahrzeug mit einer leeren Palette am zweitschnellsten und ein Fahrzeug mit einer angebrochenen Ladung am langsamsten.
3. Beim Stau muss das Fahrverhalten des Fahrzeuges dazu beitragen, dass der Stau so schnell wie möglich beseitigt werden kann.
4. Wenn ein Fahrzeug in eine Fahrstrecke eingefahren ist, dessen erlaubte Geschwindigkeit kleiner ist als die Geschwindigkeit des Fahrzeuges, muss das Fahrzeug seine Geschwindigkeit auf die Geschwindigkeit der Fahrstrecke anpassen.

Die Tests für den Fahralgorithmus umfassen drei Testfälle.

Test für die Überholverbotseinhaltung des Fahrzeuges: Dieser Test prüft, ob das Überholverbot auf der gleichen Fahrstrecke vom Fahrzeug eingehalten wird. Es werden zwei Fahrzeuge hintereinander auf eine gleiche Fahrstrecke gesetzt. Das erste Fahrzeug hat eine Ladung und das zweite Fahrzeug ist leer. Während des Tests werden die Pufferpositionen der beiden Fahrzeuge durchgehend verglichen. Es wird geprüft, ob die Pufferposition des leeren Fahrzeuges immer größer als die Pufferposition des Fahrzeuges mit der Ladung ist.

Test für die Geschwindigkeit des Fahrzeuges: Dieser Test prüft, ob sich Fahrzeuge ohne im Vergleich zu Fahrzeugen mit einer Ladung unterschiedlich schnell bewegen. Im Test werden vier Fahrzeuge auf der gleichen Pufferposition auf vier parallel laufende Fahrstrecken mit gleicher Länge gesetzt. Ein Fahrzeug hat keine Ladung, eines hat leere Palette und die anderen zwei haben die gleiche Ladung. Während des Tests werden die Pufferposition der vier Fahrzeuge durchgehend verglichen.

Test für die Geschwindigkeitsanpassung des Fahrzeuges: Dieser Test prüft, ob ein Fahrzeug, das in eine Fahrstrecke mit einer erlaubten Geschwindigkeit einfährt, welche kleiner ist als die des Fahrzeuges, seine Geschwindigkeit an die Geschwindigkeit der Fahrstrecke anpassen kann. Im Test wird ein Fahrzeug mit der Geschwindigkeit 1m/s auf eine Fahrstrecke mit einer erlaubten Geschwindigkeit von 1m/s gesetzt. An diese Fahrstrecke wird eine weitere Fahrstrecke mit einer erlaubten Geschwindigkeit von 0,8m/s angeschlossen. Während des Tests wird geprüft, ob sich die Geschwindigkeit des Fahrzeuges richtig anpasst.

3.8.6 Testen des Routings

Routing wird eigentlich nur dann benötigt, wenn ein Fahrzeug an einer verzweigenden Weiche steht, da es hier mehrere Möglichkeiten für das Weiterfahren gibt. Die Entscheidung muss dann gemäß des Routingergebnisses fallen. Das Fahrzeug muss den Ausgangsport vom Fork nehmen, der es zu seinem Ziel führen kann. Wenn mehr als nur eine Möglichkeit dafür vorhanden ist, muss das Fahrzeug den Ausgangsport bevorzugen, der ihn auf dem kürzesten Weg zu seinem Ziel führt. Es muss getestet werden, ob diese Anforderungen von der Implementierung des Routings erfüllt werden können.

Im Test werden mehrere Fahrzeuge mit unterschiedlichen Zielen auf die Fahrstrecke gesetzt. Das Modell wird initialisiert und simuliert. Während der Simulation wird geprüft, ob sich alle Fahrzeuge immer an ihren richtigen Positionen befinden und am Ende ihr Ziel erreichen. Für einige Fahrzeuge bestehen mehrere mögliche Wege, über welche diese ihr Ziel erreichen können. In diesem Fall wird noch geprüft, ob die Fahrzeuge ihr Ziel über den kürzesten Weg erreicht haben.

3.8.7 Testen der komplexeren Vorgänge

Die komplexeren Vorgänge simulieren das Entladen und Beladen eines Fahrzeuges.

Testen des CommissioningBayHandlings

CommissioningBayHandling ist eine Klasse, die für das Entladen der Ladungen in der Kommissionierzone zuständig ist. Für diese Klasse sind folgende Anforderungen gestellt:

1. Wenn ein Fahrzeug in eine `CommissioningBay` einfährt, dessen Artikel einem der in der Kommissionierzone erwarteten Artikel entspricht, muss das Fahrzeug dort entladen werden.

2. Wenn die Anzahl der Artikel auf dem Fahrzeug identisch ist mit der Anzahl der zu kommissionierenden Artkel in der Kommissionierzone, dann muss das

Fahrzeug soweit entladen werden, bis nur noch eine leere Palette übrigbleibt. Das Fahrzeug muss dann `PalletUnloadingBay` als weiteres Ziel bekommen.

3. Wenn die Anzahl der Artikel auf dem Fahrzeug größer ist als die Anzahl der zu kommissionierenden Artikel in der Kommissionierzone, muss das Fahrzeug die Anzahl an Artikeln entladen, die in dieser Kommissionierzone benötigt werden. Wenn die übriggebliebene Anzahl an Artikeln auf dem Fahrzeug von einer anderen Kommissionierzone noch benötigt werden, muss das Fahrzeug jene `CommissioningBay` als sein weiteres Ziel bekommen. Sonst muss das Fahrzeug eine `UnloadingBay` als Ziel bekommen.

4. Wenn die Anzahl der Artikel auf dem Fahrzeug niedriger ist als die Anzahl der zu kommissionierenden Artikel in der Kommissionierzone, dann muss das Fahrzeug soweit entladen werden, bis nur noch eine leere Palette übrigbleibt. Die Anzahl der noch zu kommissionierenden Artikel in dieser Kommissionierzone muss um die schon entladene Anzahl reduziert werden.

5. Die Ticks für das Entladen eines Artikels müssen richtig auf die Ticks des Fahrzeuges addiert werden.

Testfall 1: Im Test werden zwei Kommissionierpaletten in einer Kommissionierzone erzeugt. Jede Kommissionierpalette benötigt von einem Artikel 23 Stück. Es wird dann ein Fahrzeug mit 123 Stück des gleichen Artikels erzeugt und das Fahrzeug bekommt diese Kommissionierzone als Ziel. Das Modell wird initialisiert und simuliert. Im `CommissioningBay` wird geprüft, ob es keine Kommissionierpaletten mehr gibt. Wenn das Fahrzeug die Pufferposition 0 von `CommissioningBay` erreicht und dort entladen wird, wird geprüft, ob ihre Ticks um den Tickwert, der für das Entladen eines Artikels benötigt wird, erhöht wurden. Des weiteren wird geprüft, ob die Anzahl des Artikels des Fahrzeuges nach dem Entladen um 46 reduziert wurde und ob das Fahrzeug ein `UnloadingBay` als nächstes Ziel bekommt.

Testfall 2: Im Test werden zwei Kommissionierpaletten in einer Kommissionierzone erzeugt. Jede Kommissionierpalette benötigt von einem Artikel 100 Stück. Es wird dann ein Fahrzeug mit 123 Stück des gleichen Artikels erzeugt und das Fahrzeug bekommt diese Kommissionierzone als Ziel. Das Modell wird initialisiert und simuliert. Im `CommissioningBay` wird geprüft, ob es noch eine Kommissionierpalette gibt, die 77 Stück desselben Artikels benötigt. Des weiteren wird geprüft, ob das Fahrzeug nach dem Entladen nur noch eine leere Palette enthält und `PalletUnloadingBay` als nächstes Ziel bekommt.

Testfall 3: Im Test werden zwei Kommissionierpaletten in einer Kommissionierzone erzeugt. Jede Kommissionierpalette benötigt von einem Artikel 100 Stück. Es

wird dann ein Fahrzeug mit 123 Stück des gleichen Artikels erzeugt. Das Fahrzeug bekommt jedoch eine andere Kommissionierzone als Ziel. Das Modell wird initialisiert und simuliert. Im `CommissioningBay` wird geprüft, ob es immer noch zwei Kommissionierpaletten gibt, die 100 Stück vom Artikel benötigen. Wenn das Fahrzeug die Pufferposition 0 von `CommissioningBay` erreicht, wird sichergestellt, dass die Ticks des Fahrzeuges nicht um den Tickwert, der für das Entladen eines Artikels benötigt wird, erhöht wurden. Des weiteren wird geprüft, ob das Fahrzeug immer noch 123 Stück vom Artikel enthält.

Testen des LoadingBayHandlings

LoadingBayHandling ist die Klasse, die für das Beladen eines leeren Fahrzeuges zuständig ist. Für diese Klasse sind folgende Anforderungen gestellt:

1. Alle Ladungen, die in einem `LoadingBay` auf die Abholung warten, müssen richtig auf die Fahrzeuge beladen werden.
2. Die Warteliste muss richtig aktualisiert werden.
3. Die Ticks für das Beladen eines Fahrzeuges müssen richtig auf die Ticks des Fahrzeuges addiert werden.
4. Wenn ein Fahrzeug mit einer Ladung oder ein Fahrzeug mit einer leeren Pallette ins `LoadingBay` eingefährt, darf es nicht mehr mit einer neuen Ladung beladen werden.

Testfall 1: Dieser Test ist für die Situation vorgesehen, in der die Anzahl der Ladungen in `LoadingBay` niedriger als die Anzahl der Fahrzeuge ist, die zur `LoadingBay` unterwegs sind, um beladen zu werden. Im Test werden zuerst vier Fahrzeuge erzeugt und auf eine Fahrstrecke gesetzt. Dann werden drei Ladungen erzeugt und auf `LoadingBay` gelegt. Das Modell wird initialisiert und simuliert. Im `LoadingBay` wird geprüft, ob alle Ladungen aus der`LoadingBay` weggenommen werden. Wenn ein Fahrzeug die Pufferposition 0 der `LoadingBay` erreicht und dort beladen wird, wird geprüft, ob seine Ticks um den Tickwert, der für ein Beladen benötigt wird, erhöht werden. Außerdem wird geprüft, ob ein Fahrzeug leer aus der `LoadingBay` ausgefahren ist.

Testfall 2: Dieser Test ist für die Situation vorgesehen, in der die Anzahl der Ladungen in `LoadingBay` höher als die Anzahl der Fahrzeuge ist, die zur `LoadingBay` unterwegs sind, um beladen zu werden. Im Test werden zuerst vier Fahrzeuge erzeugt und auf eine Fahrstrecke gesetzt. Dann werden acht Ladungen erzeugt und sechs davon werden auf `LoadingBay` gelegt. Ein Fahrzeug wird mit einer Ladung geladen und ein anderes wird mit einer leeren Pallette geladen. Das Modell wird

initialisiert und simuliert. Im `LoadingBay` wird geprüft, ob nur zwei Ladungen aus `LoadingBay` weggenommen werden und die verbliebenen Ladungen den restlichen erzeugten Ladungen entsprechen. Wenn das Fahrzeug, das vorher schon mit einer Ladung oder einer leeren Palette beladen ist, die Pufferposition 0 der `LoadingBay` erreicht, wird geprüft, ob seine Ticks nicht um den Tickwert, der für ein Beladen benötigt wird, erhöht werden und wenn er aus der `LoadingBay` ausgefahren ist, wird geprüft, ob er immer noch die gleiche Ladung hat.

Testen des PalletUnloadingBayHandlings

PalletUnloadingBayHandling ist die Klasse, die für das Beladen einer leeren Palette zuständig ist. Für diese Klasse sind die folgende Anforderungen gestellt:

1. Alle leeren Paletten müssen richtig entladen werden.
2. Nach Entladen der leeren Palette muss das Fahrzeug das Ziel *Holding-Track* bekommen.
3. Die Ticks für Entladen einer leeren Palette müssen richtig auf die Ticks des Fahrzeuges addieret werden.
4. Wenn ein Fahrzeug mit einer Ladung ins PalletUnloadingBay eingefahren ist, darf es dann nicht entladen werden.

Im Test werden zuerst drei Fahrzeuge erzeugt und auf eine Fahrstrecke gesetzt. Die vorderen zwei Fahrzeuge werden dann mit einer leeren Palette mit dem Ziel `PalletUnloadingBay` beladen. Das dritte Fahrzeug wird mit einer Ladung mit dem Ziel `CommissioningBay` beladen. Das Modell wird initialisiert und simuliert. In der `PalletUnloadingBay` wird geprüft, ob die leeren Paletten von allen Fahrzeugen richtig entladen werden und das Fahrzeug nach dem Enladen der Palette das Ziel `HoldingTrack` hat. Wenn ein Fahrzeug die Pufferposition 0 der `PalletUnloadingBay` erreicht und dort seine leere Palette entladen wird, wird geprüft, ob seine Ticks um den Tickwert, die für das Enladen einer leeren Palette benötigt werden, erhöht werden. Wenn das Fahrzeug, das das Ziel `CommissioningBay` hat, die Pufferposition 0 der `PalletUnloadingBay` erreicht, wird geprüft, ob seine Ticks nicht um den Tickwert, die für das Entladen einer leeren Palette benötigt werden, erhöht werden und wenn es aus der `LoadingBay` ausgefahren ist, wird geprüft, ob es immer noch die gleiche Ladung und das gleiche Ziel hat.

Testen des UnloadingBayHandlings

UnloadingBayHandling ist die Klasse, die für das Entladen der angebrochenen Ladung zuständig ist. Für diese Klasse sind folgende Anforderungen gestellt:

1. Alle angebrochenen Ladungen müssen richtig entladen werden.

2. Nach dem Entladen der angebrochenen Ladung muss das Fahrzeug HoldingTrack als Ziel bekommen.

3. Die Ticks für das Entladen einer angebrochenen Ladung müssen richtig auf die Ticks des Fahrzeuges addiert werden.

4. Wenn ein Fahrzeug mit einer Ladung, die keine angebrochene Ladung ist, ins UnoadingBay einfährt, darf die Ladung von ihm nicht entladen werden.

5. Die Sammelliste der angebrochenen Ladungen muss richtig aktualisiert werden.

Im Test werden zuerst vier Fahrzeuge erzeugt und auf eine Fahrstrecke gesetzt. Die drei ersten Fahrzeuge werden dann mit einer angebrochenen Ladung mit dem Ziel `UnloadingBay` beladen. Das vierte Fahrzeug wird mit einer geschlossenen Ladung mit dem Ziel `CommissioningBay` beladen. Das Modell wird initialisiert und simuliert. Im `UnloadingBay` wird geprüft, ob die angebrochene Ladung von allen Fahrzeugen richtig entladen wird und das Fahrzeug nach dem Entladen der angebrochenen Ladung das Ziel `HoldingTrack` hat. Wenn ein Fahrzeug die Pufferposition 0 der `UnloadingBay` erreicht und dort seine angebrochene Ladung entladen wird, wird geprüft, ob seine Ticks um den Tickwert, der für das Entladen einer angebrochenen Ladung benötigt wird, erhöht werden. Wenn das Fahrzeug, das das Ziel `CommissioningBay` hat, die Pufferposition 0 der `UnloadingBay` erreicht, wird geprüft, ob die Ticks des Fahrzeuges nicht erhöht werden und wenn er aus der `LoadingBay` ausgefahren ist, wird geprüft, ob er immer noch die gleiche Ladung und das gleiche Ziel hat. Bei der Sammelliste der angebrochenen Ladungen wird geprüft, ob die Anzahl der angebrochenen Ladungen um drei erhöht wird.

Testen des HoldingTrackHandlings

HoldingTrackHandling ist die Klasse, die für das Fahrverhalten aller leeren Fahrzeuge im `HoldingTrack` zuständig ist. Für diese Klasse sind folgende Anforderungen gestellt:

1. Wenn keine Ladung im `LoadingBay` auf die Abholung wartet, dürfen keine Fahrzeuge im `HoldingTrack` bewegen.

2. Wenn es Ladungen gibt, die im `LoadingBay` auf die Abholung warten, müssen die Fahrzeuge aus dem `HoldingTrack` ausfahren, um die Ladungen zu holen.

3. Die Anzahl der ausgefahrenen Fahrzeuge muss mit der Anzahl der Ladungen übereinstimmen, die im `LoadingBay` auf die Abholung warten. Wenn die Anzahl der Fahrzeuge kleiner ist, müssen alle neu ins `HoldingTrack` eingefahrenen Fahrzeuge sofort wieder ausfahren, bis alle wartenden Ladungen geholt wurden.

Testfall 1: Dieser Test ist für die Situation vorgesehen, dass keine Ladungen im LoadingBay auf die Abholung warten. Im Test werden zuerst sechs Fahrzeuge erzeugt und auf das `HoldingTrack` gesetzt. Das Modell wird initialisiert und simuliert. Nach einer ersten Simulationsrunde wird geprüft, ob das „Stoppbit“ vom `HoldingTrack` auf „true“ gesetzt wurde. Nach mehren Simulationsrunden wird geprüft, ob alle acht Fahrzeuge immer noch im `HoldingTrack` stehengeblieben sind.

Testfall 2: Dieser Test ist für die Situation vorgesehen, in der die Anzahl der im `LoadingBay` auf die Abholung wartenden Ladungen kleiner ist als die Anzahl der Fahrzeuge, die im `HoldingTrack` stehen. Im Test werden zuerst sechs Fahrzeuge erzeugt und alle auf das `HoldingTrack` gesetzt. Dann werden vier Ladungen erzeugt und auf die `LoadingBay` gelegt. Das Modell wird initialisiert und simuliert. Nach einer Simulationsrunde wird geprüft, ob das „Stoppbit“ vom `HoldingTrack` auf „false“ gesetzt wird. Nach mehreren Simulationsrunden wird geprüft, ob noch vier Fahrzeuge im `HoldingTrack` stehengeblieben sind und ob alle vier Ladungen auf `LoadingBay` von den Fahrzeugen weggenommen werden.

Testfall 3: Dieser Test ist für die Situation vorgesehen, in der die Anzahl der im `LoadingBay` auf die Abholung wartenden Ladungen größer als die Anzahl der Fahrzeuge ist, die im `HoldingTrack` stehen. Im Test werden zuerst acht Fahrzeuge erzeugt und sechs davon auf das `HoldingTrack` gesetzt. Die zwei anderen werden auf eine Fahrstrecke gesetzt. Auf die `LoadingBay` werden dann acht Ladungen gelegt. Das Modell wird initialisiert und simuliert. Nach einer Simulationsrunde wird geprüft, ob das „Stoppbit“ vom `HoldingTrack` auf „false“ gesetzt wird. Nach mehreren Simulationsrunden wird geprüft, ob die Fahrzeuge, die vorher in `HoldingTrack` gesetzt wurden, alle aus `HoldingTrack` herausgefahren sind und ob immer noch zwei Ladungen im `LoadingBay` übriggeblieben sind. Nach weiteren Simulationsrunden wird geprüft, ob die zwei neu ins `HoldingTrack` eingefahrenen Fahrzeuge sofort wieder aus `HoldingTrack` herausgefahren sind, um die zwei übriggeblieben Ladungen im `LoadingBay` zu holen.

3.8.8 Testergebnisse

Alle Testergebnisse entsprechen den Soll-Ergebnissen.

3.9 Fazit

Nachdem im ersten Semester die Schwerpunkte auf Entwicklung und Visualisierung lagen, wurde das zweite Semester zum Abschluss der Entwicklung des Basissystems (einer lauffähigen Simulationsumgebung, welche alle geforderten Vorgänge simulie-

ren kann), zur Weiterentwicklung des Gesamtkonzeptes, zum Testen des Systems und zur Zusammenführung der beiden Teilgruppen genutzt.

Nachdem das Basissystem kurz nach Anfang des zweiten Semesters fertig war, ging die Entwicklung ins Detail. So wurde beispielsweise neben der Optimierung von Routing und Simulationszeit die Ladungsverteilung überarbeitet. Eine neue *jABC*-Version bedurfte zwischenzeitlich einiger Modifikationen des Codes. Gleichzeitig wurde an vielen kleinen Details gearbeitet und die einzelnen Teile zu einem einheitlichen Gesamtkonzept zusammengefügt.

Damit stand uns die Grundlage für eine komfortable, wirklichkeitsnahe Simulation zur Verfügung. Zu diesem Zeitpunkt wurde durch die Ergebnisse der UPPAAL-Teilgruppe deutlich, das es keine triviale Lösung für etwaige generelle Komissionierstrategien, Verteilungsalgorithmen, Routing etc. gibt. Nach eingehender Überlegung und Beratung mit Logistikern wurde der ursprüngliche Gedanke verschiedene Simulationsstrategien zu testen verworfen. Dies gab der *jABC*-Gruppe die Möglichkeit an Detailverbesserungen, besserer Konfigurierbarkeit und besserer Bedienbarkeit zu arbeiten. Die Möglichkeit eine Simulation beliebig unterbrechen zu können und zu wiederholen (Replay) wurde ebenfalls in diesem Zeitraum geschaffen. Gleichzeitig galt es nun die bereits erstellten Teile eingehend auf ihre Korrektheit zu testen und die zwei Teilgruppen mit Hilfe eines gemeinsamen Modells zusammen zu führen. Nachdem die Zusammenführung abgeschlossen war, wurde sich ausschließlich dem Testen gewidmet. Der erfolgreiche Abschluss der Tests markiert damit auch das Ende der eigentlichen Arbeiten.

Insgesamt ist die *jABC*-Teilgruppe gut voran geschritten. Es konnten nicht nur etwaige Defizite des vergangenen Semesters aufgeholt werden, sondern auch das PG-Ziel erreicht werden.

4 Uppaal-Modell

4.1 Begriffserklärungen

EHB-Fahrzeug, Fahrzeug, Gehänge bezeichnet ein selbständig in einem Schienensystem fahrendes Gerät, das in der Lage ist, eine befüllte Palette aufzunehmen und wieder abzugeben. Es kann in Bahnhöfe einfahren und dort festgestellt werden, damit ein Kommissionierer Artikel von der getragenen Palette herunternehmen kann.

EHB-Palette ist eine Palette, die von einem EHB-Fahrzeug transportiert wird oder werden soll. Die auf einer solchen Palette liegenden Artikel sind von genau einer Sorte. EHB-Paletten lagern in einem Hochregallager, während sie nicht auf einem Fahrzeug unterwegs sind. Sie werden nur aufgrund eines EHB-Auftrags ausgelagert.

EHB-Auftrag ist eine Anweisung für ein EHB-Fahrzeug, welche Bahnhöfe und anderen Ziele es mit einer bestimmten EHB-Palette anfahren soll. Zu jeder ausgelagerten EHB-Palette gehört genau ein EHB-Auftrag.

Ziel bedeutet im Zusammenhang mit einem EHB-Auftrag eine durch das bearbeitende EHB-Fahrzeug anzufahrende Stelle (i. d. R. ein Bahnhof).

Unterziel ist Teil eines Ziels und identifiziert Palettenaufträge, die im zum Ziel gehörenden Bahnhof abgearbeitet werden und Bedarf an den auf der EHB-Palette mitgebrachten Artikeln haben.

Anbruchpalette bezeichnet eine EHB-Palette, die nicht mehr ihre ursprüngliche Artikelmenge enthält. Solche Paletten entstehen, wenn nicht alle Artikeleinheiten der Palette durch einen EHB-Auftrag verplant (und durch einen Kommissionierer heruntergenommen) wurden. Sie sind von Interesse, da ihre Wiedereinlagerung in das Hochregallager viel Zeit kostet.

Schienenstück, -element bezeichnet einen Teil eines Schienensystems, auf dem EHB-Fahrzeuge fahren können.

Übergabepunkt ist die eindeutig numerierte Übergabeschnittstelle zwischen zwei Schienenstücken.

Wagenpuffer ist ein Leergleis, auf dem unbenötigte EHB-Fahrzeuge abgestellt werden.

Bahnhof ist einen Teil des Schienensystems, in dem EHB-Fahrzeuge abgebremst und festgestellt werden können, damit der für den Bahnhof zuständige Kommissionierer Artikel entnehmen und diese auf die zu den aktuell bearbeiteten Palettenaufträgen gehörenden Paletten packen kann.

Belade-, Entladezone ist der Bereich wo Ladung zwischen HRL und EHB-Fahrzeug verschoben wird.

Palettenauftrag ist eine Anweisung für den Kommissionierer, mit welchen Artikeln in welcher Menge er eine leere Palette im Bahnhof zu befüllen hat. Die anhand von Palettenaufträgen gepackten Paletten werden durch Lastwagen zu den bestellenden IKEA-Läden befördert. Die Palettenaufträge, deren resultierende Paletten gemeinsam in einen LKW gelangen sollen, gehören zum selben LKW-Auftrag.

Bedarf ist eine zu einem Palettenauftrag gehörende virtuelle Artikelmenge bestimmter Artikelart, die durch eine entsprechende reale Artikelmenge auszugleichen, d. h. „zu erfüllen“ ist, indem letztere auf die zugehörige Palette gepackt wird.

LKW-Auftrag ist eine Menge von Palettenaufträgen, deren resultierende Paletten in denselben LKW gepackt werden sollen.

Batch ist eine Menge von LKW-Aufträgen die im selben Arbeitsdurchgang (Arbeitsschicht) bearbeitet werden sollen.

4.2 Anforderungen

Mit Hilfe von Uppaal soll ein abstraktes Modell des zu simulierenden Lagers (siehe Kapitel 1.1) erstellt werden, das nicht der Visualisierung, sondern der Analyse der Abläufe dient. Die Analyse wird anhand von später zu entwickelnden Anfragen an den in Uppaal integrierten Model-Checker durchgeführt. Hierbei sind vor allem Anfragen von Interesse, die die Durchführbarkeit von Aufträgen innerhalb der so genannten Batch unter Berücksichtigung von Zeitschranken überprüfen.

Um ein korrektes und aussagekräftiges Ergebnis zu erhalten sind bei der Entwicklung verschiedene Aspekte zu berücksichtigen. So muss bezüglich der EHB-Strecke eine richtige Fortbewegung der auf ihr fahrenden Fahrzeuge gewährleistet werden. Dieses beinhaltet unter anderem, dass sich Fahrzeuge auf einfachen Streckenabschnitten nicht überholen und nicht kollidieren dürfen. Des weiteren müssen die Fahrzeuge die Schienen in vorgegebener Richtung und in korrekter Reihenfolge befahren, da es sich bei der Strecke um eine „zyklische Einbahnstraße“ handelt.

Neben diesen eher technischen Aspekten müssen aber auch Aspekte der Lagerhaltung berücksichtigt werden. Zu diesen gehören zum Teil selbstverständliche Dinge, wie beispielsweise die gleichbleibende Menge der Waren innerhalb des Modells, aber auch der Algorithmus zur Verwaltung der *Anbruchpalette* beim Ein- und Auslagern aus dem bzw. in das Hochregallager. Nach Möglichkeit soll die Verwendung des Algorithmus so flexibel sein, dass er zum Testen von Alternativen einfach ausgetauscht werden kann.

Gleiches gilt für den zur Verteilung der zu kommissionierenden Aufträge verwendeten Algorithmus. Dieser sollte nach Möglichkeit ebenfalls leicht und schnell austauschbar sein, sodass sich zum einen komplett unterschiedliche, aber auch Weiterentwicklungen von Ansätzen leicht implementieren lassen. Die verschiedenen Varianten zur Verteilung sollen unterschiedliche Restriktionen berücksichtigen, wie zum Beispiel die Zuordnung zwischen Auftragszielen und bestimmten Kommissionierstationen.

Um die Entwicklung des Modells zu erleichtern, soll zunächst ein Teil des Modells nachgebildet werden, an dem das korrekte Zusammenspiel der Prozesse und Optimierungen leichter nachvollzogen werden kann. Durch die Wiederverwendung dieser Komponenten soll dann das gesamte Modell entstehen.

4.3 Vereinfachende Annahmen

Bei der Modellierung, auf die in Kapitel 4.5 eingegangen wird, wurden einige Annahmen gemacht, um die Komplexität des UPPAAL-Modells zu reduzieren. Allgemein erleichtern diese die Implementierung der Vorlagen und verkleinern damit auch deren Fehleranfälligkeit, bei gleichzeitigem Gewinn an Lesbarkeit und Performanz bei der Simulation. Bei jeder einzelnen Vereinfachung gilt es allerdings zu unterscheiden, aus welchem Grunde sie eingeführt wurde. Manche sind nur vorläufig und nur deshalb eingeführt worden, weil zum Zeitpunkt der Modellierung nichts Genaueres über das Verhalten des Lagers an den betreffenden Stellen bekannt war. Die Stellen, an denen bei der Implementierung solche Annahmen eingebracht wurden, müssen bei Bekanntwerden der genauen Gegebenheiten daraufhin überprüft werden, ob sie in der jetzigen Form erhalten bleiben können. Andere Annahmen sind getroffen worden, um eine Rahmenimplementierung machen zu können, die später evtl. noch zu verfeinern ist. Bei diesen wurde darauf geachtet, dass sie dennoch korrekte Schranken liefern, die im weiteren Modellierungsprozess z. T. schärfer werden sollen.

So wurde anfangs eine Durchschnittsgeschwindigkeit für fahrende Fahrzeuge angenommen. Sowohl die Beschleunigung von Fahrzeugen als auch das Abbremsen wurden weggelassen, da Beschleunigungs- bzw. Verzögerungswege bei der jetzigen bekannten Geschwindigkeit von 1 m/s zu vernachlässigen sind. Allerdings berücksichtigt das Modell z. Zt. noch keine unterschiedlichen Geschwindigkeitsstufen für z. B. beladene und unbeladene Fahrzeuge. Hierbei handelt es sich um einen Aspekt,

der weiter auszumodellieren ist.

Im Rahmen vernünftiger Grenzen ist im Moment unbekannt, welchen Sicherheitsabstand Fahrzeuge zueinander haben sollen und ob es Abschnitte von Schienen gibt, auf denen sich zu einer Zeit nur ein Fahrzeug befindet. Daher enthält das Uppaal-Modell Blockstellen, die nur von einem Fahrzeug besetzt werden können. Erst nach Verlassen einer solchen kann ein weiteres Fahrzeug darauf einfahren. Damit wird automatisch auch ein Mindestsicherheitsabstand gewahrt. Bei Wartegleisen – das sind Schienenstücke, auf denen auftragslose Fahrzeuge auf ihren nächsten Auftrag warten – wird angenommen, dass Fahrzeuge nach ihrer Einfahrt sofort wieder bereit sind, einen neuen Auftrag zu bekommen und direkt wieder Ausfahrt erhalten können. Diese Annahme ist korrekt, solange es tatsächlich ein direkt am Ausgang wartendes Fahrzeug gibt bzw. ein neuer Auftrag an ein Fahrzeug erst dann vergeben wird, wenn dieses sich wieder unmittelbar am Ausgang befindet. An dieser Stelle sollte für die Fahrzeuge noch unbedingt eine hinreichend große Wartezeit berücksichtigt werden.

In besagtes Wartegleis fahren zum jetzigen Zeitpunkt alle Fahrzeuge ein, die keinen Auftrag mehr haben, da zum einen deren exaktes Verhalten noch unbekannt ist, zum anderen eine Simulation oder ein Model-Checking nicht enden würde, wenn Leerfahrzeuge nach Belieben umherfahren könnten.

Ebenfalls unbekannt ist die Pick-Zeit, die ein Kommissionierer für einen bestimmten Artikel benötigt, und ob ein solches Detail in aller Ausführlichkeit berücksichtigt werden soll. Daher wird z. Zt. eine konstante Kommissionierzeit pro abgeladenem Artikel angesetzt, die für eine Berechnung des Modells groß genug zu wählen ist.

Bahnhöfe können auch dann mit Artikeln für Palettenaufträge beliefert werden, wenn alle Kommissionierplätze zum Zeitpunkt der Auftragserstellung belegt sind und der für die Lieferung vorgesehene Auftrag sich noch nicht in der Kommissionierphase befindet (zum genauen Ablauf der Kommissionierung an einem Bahnhof, siehe Vorlage `Bahnhof` in Kapitel 4.5.7). Dies hat zur Konsequenz, dass evtl. nicht erfüllbare EHB-Aufträge generiert werden, welches der Fall ist, wenn ein Fahrzeug mit einer Lieferung für einen sich nicht in der Kommissionierung befindenden Auftrag in einem Bahnhof an der Abladestelle steht. Dieses blockiert dann den Bahnhof. Daher macht es wenig Sinn, schon zu Anfang EHB-Aufträge zu generieren, die alle einem Bahnhof zugeordneten Palettenaufträge berücksichtigen. Aus diesem Grund wurde eine Konstante c angenommen, sodass nur Fahrzeuge für die nächsten c zu kommissionierenden Aufträge ausgesendet werden können. Hier könnte neben einer Überlegung, wie die Konstante günstig zu wählen ist, gleiches auch empirisch am konkreten Fall überprüft werden. Auch wären andere Verfahren zur Generierung von Fahrzeugaufträgen denkbar, die sich des Problems auf andere Weise annehmen.

Im Moment wird keine geeignete Startaufstellung der Fahrzeuge festgelegt. Im tatsächlichen Lager können vor der ersten Tagesschicht EHB-Fahrzeuge schon mit Paletten beladen werden und an die entsprechenden Bahnhöfe fahren, sodass diese direkt bei Schichtbeginn ausgelastet sind. Im Modell fahren jedoch alle Fahrzeuge erst zu Beginn der Simulation an einer bestimmten Stelle los. Hier ist noch zu über-

legen, wie eine günstigere Aufstellung aussehen könnte, sodass es weniger Vorlaufzeit gibt und damit ein optimaleres Ergebnis erzielt wird.

4.4 Konzepte bei der Modellierung

4.4.1 Schienenstücke

Als *Schienenstücke* oder alternativ *Schienenelemente* bezeichnen wir alle Automatenvorlagen, die einen durch Elektrohängebahnen befahrbaren Abschnitt des IKEA-Lagers modellieren. Charakteristikum von Schienenstücken im UPPAAL-Modell ist daher, dass diese mit `Fahrzeug`-Prozessen und mit anderen Schienenstück-Prozessen interagieren. Im gegenwärtigen Modell existieren Vorlagen für eine einfache Schiene, Weichen, Wagenpuffer, Bahnhöfe sowie Entlade- und Beladezonen. In Kapitel 4.5 werden die spezielle Funktion und die Funktionsweise jeder dieser Vorlagen detailliert erläutert.

An einem einfachen Beispiel soll zunächst beschrieben werden, wie die Interaktion zwischen Fahrzeugen und Schienenstücken prinzipiell stattfindet. Einzelne Schienenstücke haben im UPPAAL-Modell potentiell mehrere Einfahr- und Ausfahrpunkte, von denen aus Fahrzeuge ein- bzw. ausfahren können. So haben einfache Schienen (Vorlage `Schiene`) jeweils genau einen Einfahr- und Ausfahrpunkt, dreiteilende Weichen (Vorlage `DreierWeicheTeilung`) aber z. B. einen Einfahrpunkt und drei Ausfahrpunkte. Jedem Einfahrpunkt eines Schienenstücks wird eine eindeutige Nummer, die mit `vonID` bezeichnet wird, zugeordnet. Ebenso wird jedem Ausfahrpunkt eine Nummer zugeordnet, die `zuID` genannt wird. Wir sagen, zwei Schienenstücke haben einen *Übergabepunkt* gemeinsam, gdw. die `zuID` eines Schienenstückes mit der `vonID` eines anderen übereinstimmt. Wenn zwei Schienenstücke `a` und `b` einen Übergabepunkt gemeinsam haben, dann bedeutet das, dass der mit der `vonID` ausgezeichnete Einfahrpunkt des Schienenstücks `b` auf den mit der `zuID` von `a` ausgezeichneten Ausfahrpunkt folgt. Mit anderen Worten, das Schienenstück `b` liegt an dem Schienenstück `a` an. Aus dieser Interpretation folgt, dass bei einer Menge von Schienen jede `vonID` nur genau einem Einfahrtspunkt genau eines Schienenstückes zugeordnet sein darf und dass `zuID`s nur mit Werten belegt sein dürfen, die in der Menge der `vonID`s vorkommen. Eine Menge von verbundenen Schienenstücken wird als *Schienensystem* bezeichnet.

Jede Interaktion zwischen zwei Schienenstücken und einem Fahrzeug modelliert eine Fahrzeugbewegung. Ein Beispiel findet sich in Abbildung 4.1. Nehmen wir an, dass zwei Schienen `a` und `b` (jeweils Vorlage `Schiene`) den Übergabepunkt `n` gemeinsam haben, d. h. $a.zuID = b.vonID = n$.

Ein Schienenstück, auf dem sich kein Fahrzeug befindet, ist in einem mit „Frei“ bezeichneten Initialzustand. Nur in diesem Zustand kann ein Fahrzeug aufgenommen werden, denn dann lauscht das Schienenstück `b` auf dem `schieneZuSchiene-`

`Kanal[vonID]`. Die Nummer des Übergabepunktes wird also zur Referenzierung des Kanals, über den sich Vorgänger- und Nachfolgeschiene synchronisieren, benutzt. Die Vorgängerschiene `a` kann sich nun auf diesem Kanal mittels des Synchronisationsausdrucks `schieneZuSchieneKanal[zuID]!` melden. In den `schieneZuSchieneParameter` trägt die Vorgängerschiene die Identifikationsnummer des zu übergebenen Fahrzeugs (`fahrzeugID`, in der Abbildung mit `fhzID` abgekürzt) ein. Aktion 1 in der Abbildung symbolisiert diese Synchronisation. Schienenstück `b` übernimmt die `fahrzeugID`, während `a` nach der Synchronisation in den Zustand „Frei“ zurück geht und wieder aufnahmebereit ist. Das Schienenstück `b` synchronisiert sich jetzt mit dem Fahrzeug über den `schieneZuFahrzeugKanal[fahrzeugID]`.

Der eben erhaltene Parameter wird also wiederum zur Identifikation des passenden Kanals genutzt. Schienenstück `b` teilt nun über den Parameter `schieneZuFahrzeugParameter` dem Fahrzeug seine `vonID` mit, das ist die Nummer des Einfahrtspunkts, den das Fahrzeug gerade passiert hat. So erhält das Fahrzeug immer die Information, auf welchem Schienenstück es sich gerade befindet. Gleichzeitig werden auch die Länge des Schienenstücks und die maximal darauf erlaubte Geschwindigkeit übergeben. Dieser Schritt ist der Aktion 2 der Abbildung zu entnehmen.

Im jetzt erreichten Zustand können beliebige Aktionen mit dem Fahrzeug geschehen, die natürlich abhängig vom konkreten Schienenstück sind. In einem Bahnhof z. B. könnte das Fahrzeug entladen werden. Auf einfachen Schienen, wie in der Zeichnung angedeutet ist, vergeht eine Wartezeit, deren Dauer abhängig von der Länge des Schienenstücks (s) geteilt durch das Minimum der Fahrzeuggeschwindigkeit ($V_{\max,\,\mathrm{Fhz}}$) und der auf dem Schienenstück maximal erlaubten Geschwindigkeit ($V_{\max,\,\mathrm{Schiene}}$) ist, als Formel $t = \frac{s}{min\{V_{\max,\,\mathrm{Fhz}}, V_{\max,\,\mathrm{Schiene}}\}}$. Nach Ablauf der Wartezeit meldet sich das Fahrzeug über den `fahrzeugZuSchieneKanal` bei der befahrenen Schiene (Aktion 3). Letztere synchronisiert sich nun mit ihrem Nachfolger auf gleiche Weise wie gerade beschrieben.

4.4.2 Routing

Damit die Fahrzeuge den Weg zu ihren Zielbahnhöfen im Elektrohängebahnnetz finden, bedarf es eines Routings. Wir haben uns für ein einfaches, statisches Routing entschieden: Jede der verzweigenden Weichen erhält eine feste Tabelle, in der für jedes mögliche Ziel die Richtungen stehen, in die ein erwünschter Weg zu diesem Ziel verläuft. Diese Tabelle wird bisher von Hand gepflegt. Es ist jedoch eine automatische Generierung im Programm UppaalVis (siehe Kapitel 5) angedacht.

Weil Uppaal-CORA in den Versionen vor dem 06.02.2006 noch Probleme mit der Parameterübergabe von Arrays bei der Prozessinstanziierung hatte, haben wir die Routing-Informationen als Behelf in einem globalen Array zusammengefasst und eine Routing-Identifikationsnummer eingeführt. Das Routing-Array eines vorübergehend benutzten Entwicklungsmodells war folgendermaßen definiert:

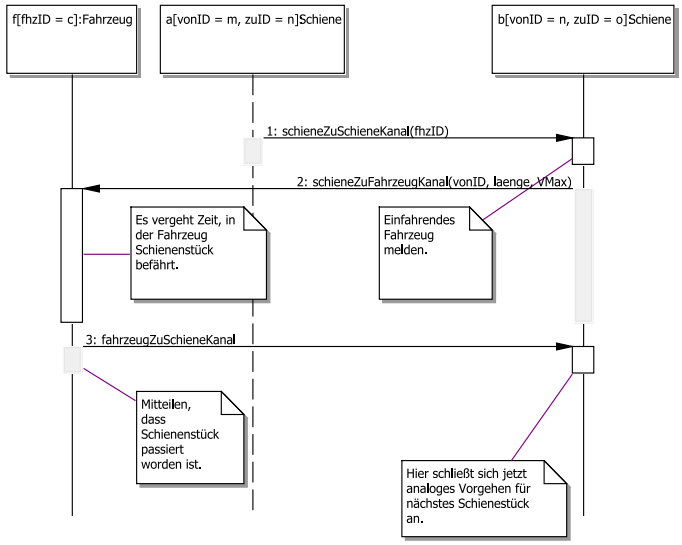


Abbildung 4.1: Eine Fahrzeugbewegung am Beispiel zweier Schienenprozesse.

```
const int routing[ANZAHL_VERZWEIGENDE_WEICHEN][BAHNHOEFE] =
                                        {{MITTE, LINKS, RECHTS, MITTE},
                                         {LINKS, RECHTS, RECHTS, LINKS},
                                         {LINKS, RECHTS, RECHTS, RECHTS}};
```

Die Ausdrücke `LINKS`, `MITTE` und `RECHTS` sind als Konstanten (Zweierpotenzen) definiert und dürfen in der Tabelle auch addiert werden, um für ein Ziel mehrere gleichwertige, alternative Richtungen anzugeben. Die Informationen der ersten Datenzeile (Index 0) sind für die verzweigende Weiche mit der Routing-Identifikationsnummer 0 gedacht und bedeuten, dass ein Fahrzeug mit Ziel-Identifikationsnummer 0 an das mittlere Nachfolgeschienenstück zu übergeben ist, für Ziel 1 an das linke, für Ziel 2 an das rechte und für Ziel 3 ebenso an das mittlere Schienenstück. In den Automaten der verzweigenden Weichen ist dies durch alternative Transitionen sichergestellt, deren Wächterausdrücke nur wahr werden, wenn das passende Richtungsbit in der Routing-Information vorkommt, wie in Abbildung 4.2 zu sehen ist.

4.4.3 Aufträge

Das neu zu bauende Ikea-Lager wird Lieferaufträge über so genannte Low-Flow- und Low-Volume-Artikel von Ikea-Läden in ganz Europa entgegennehmen. Der aktuelle Kenntnisstand ist, dass diese Aufträge gesammelt und zu *Palettenaufträgen* zusammengefasst werden. Die zugehörigen Paletten müssen kommissioniert und in Lastkraftwagen verpackt werden, um von diesen zu ihren Zielläden gebracht zu werden. Palettenaufträge werden in Stapeln zusammengefasst, den *Batch*es.

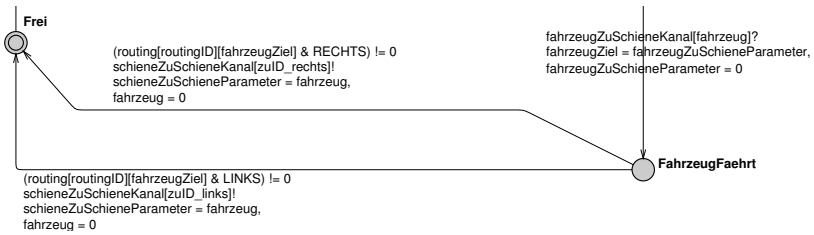


Abbildung 4.2: Alternativtransitionen im Automaten einer zu zwei Schienenstücken verzweigenden Weiche.

Wir gehen davon aus, dass wir eine fertig aufbereitete Batch erhalten, in deren Palettenaufträgen steht, in welchen LKW die jeweils zugehörige Palette einmal verladen werden soll und welche Artikel in welcher Menge auf die Palette gepackt werden sollen. Ebenso nehmen wir an, dass diese Batch für genau eine Arbeitsschicht gedacht ist. Aus diesen Palettenaufträgen erzeugen wir *EHB-Aufträge*, die einem EHB-Fahrzeug für eine Palette mit einer einzigen Artikelart sagen, in welche Bahnhöfe es diese fahren soll.

Die naheliegendste Möglichkeit, die Palettenaufträge in Uppaal zu realisieren, ist es nun, die Batch, jeden LKW-Auftrag und jeden Palettenauftrag zu einem Prozess zu machen. EHB-Aufträge lassen sich so jedoch nicht realisieren, da ihre Anzahl nicht von vorn herein fest steht und es keine Möglichkeit zur Prozesserzeugung während der Systemlaufzeit gibt. Vorab müssen die Palettenauftragsprozesse ihre Bedarfe an Artikeln und den LKW-Auftrags-Identifikator kennen.

Zuordnung zu Bahnhöfen

Die Verteilung von Palettenaufträgen zu Bahnhöfen ist eine der wesentlichen Entscheidungen, die wir zu treffen haben. Damit die Verteilungsstrategie leicht beeinflusst werden kann, wurde ein `Zuordner`-Prozess erdacht, der zwischen Palettenaufträgen und Bahnhöfen vermittelt.

Die Zuordnung von Palettenaufträgen zu Bahnhöfen findet zu Beginn der Systemlaufzeit *committed* statt, also mit Priorität und ohne Zeitfortschritt. Genau ein zufällig ausgewählter Bahnhof meldet sich beim Zuordner mit seiner Identifikationsnummer. Danach weist der Zuordner dem ersten Palettenauftrag diese Bahnhofsnummer zu. Der Zuordner führt dies der Reihe nach für alle Palettenaufträge aus. Durch dieses Verfahren erreichen wir eine zufällige Verteilung der Palettenaufträge auf die Bahnhöfe, bei dem es keine unnötigen Alternativtransitionen gibt.

Melden von Bedarfen

Sobald der Kommissionierbahnhof eines Palettenauftrags fest steht, können die Bedarfe des Auftrags vom Steuersystem berücksichtigt werden. Es ist vernünftig, EHB-

Fahrzeuge mit Artikeln für einen Palettenauftrag bereits loszuschicken, wenn für den Auftrag noch keine Kommissionierpalette im entsprechenden Bahnhof bereitsteht, denn geschähe dies erst mit Bereitstellen der Leerpalette im Bahnhof, so stünde diese wahrscheinlich mehrere Minuten leer herum. Andererseits kann es leicht passieren, dass EHB-Aufträge erzeugt werden, die Artikel für Palettenaufträge zu einem Bahnhof fahren, an dem für einen der Palettenaufträge noch keine Kommissionierpalette steht.

Wir haben uns für einen Mittelweg entschieden, der hinreichende Entscheidungsmöglichkeiten bietet. Nach der Zuordnung gelangen die Palettenaufträge im Bahnhof in zufälliger Reihenfolge in eine Warteschlange. Beim Einschlangen teilt ein Palettenauftrag seine Artikelbedarfe mit und wird in der einmal erzeugten Reihenfolge vom Bahnhof behandelt. Die Bedarfe werden durch `Sammler`-Prozesse erfasst, die sich die eingegangenen Bedarfe in einer Liste merken und abarbeiten, indem sie daraus EHB-Aufträge erzeugen.

Generierung der EHB-Aufträge

Für jede Artikelart gibt es genau einen Sammler. Die Reihenfolge der gesammelten Bedarfe kommt von der Reihenfolge, in der sich Palettenaufträge in Warteschlangen eingereiht haben, und ist damit zufällig. Wenn die Gesamtartikelmenge der erfassten und noch nicht bedienten Bedarfe größer als 0 ist, die Anzahl der Anbruchpaletten für die Artikelart weniger als die maximal zulässige Zahl beträgt und eine Beladezone im Zustand Frei ist, kann der Sammler einen EHB-Auftrag generieren und diesen zusammen mit einer passenden Palette an genau eine der freien Beladezonen geben. Diese wird ihn wiederum an das nächste EHB-Fahrzeug weiterreichen. Für jede Zusammenstellung eines EHB-Auftrags wird Algorithmus 1 aufgerufen.

Dieser initialisiert zunächst die benutzten Datenstrukturen und Variablen (Zeilen 2f.), um diese dann mit Daten zu füllen (Zeilen 4-24). Schließlich werden noch globale Variablen aktualisiert (Zeile 25ff.). In Uppaal ist dies als Funktion realisiert, die die EHB-Paletten-Datenstruktur zurückgibt.

Die erfassten Bedarfe eines Sammlers sind Datensätze, die Bahnhofsnummer, Artikelmenge und Palettenauftragsnummer enthalten. In der globalen Variable *gesammeltebedarfe* werden die Artikelmengen der erfassten Bedarfe kumuliert. Wird ein Bedarf ganz oder teilweise einer EHB-Palette (mit EHB-Auftrag, hier repräsentiert durch *palette*) zugeordnet, so wird die zugeordnete Artikelmenge von der des erfassten Datensatzes und der eben genannten globalen Variable entfernt. Nach Erfassung und Zuordnung aller Bedarfe haben die Artikelmengen der Bedarfsdatensätze und die in der Variable *gesammeltebedarfe* also den Wert 0.

Die Variable *verplant* speichert die Artikelmenge, die während des aktuellen Aufrufs schon der in der Erzeugung befindlichen EHB-Palette zugewiesen wurden.

Das zu benutzende Element z des Ziel-Arrays der EHB-Palette *palette* muss bestimmt werden, da Bedarfe von Palettenaufträgen, die dem selben Bahnhof zuge-

Algorithmus 1 Generierung einer EHB-Palette mit EHB-Auftrag.

Require: Erfasste Bedarfe nicht leer und $gesammeltebedarfe > 0$

Ensure: Die EHB-Palette *palette* ist mit Daten und einem passenden EHB-Auftrag gefüllt

```
    procedure BERECHNEZIELE
        Initialisiere palette;
        verplant ← 0;
        if Anbruchpalette vorhanden then
 5:         palette.artikelmenge ← Artikelmenge der Anbruchpalette;
        else
            palette.artikelmenge ← Palettenkapazität;
        end if
        palette.artikelart ← Artikelart dieses Sammlers;
10:     for b ∈ Erfasste Bedarfe do
            if b.artikelmenge > 0 then
                Bestimme zu benutzendes Element z des Ziel-Arrays von palette;
                Bestimme zu benutzendes Element u des Unterziel-Arrays von z;
                z.bahnhofID ← b.bahnhofID;
15:             u.palettenauftragID ← b.palettenauftragID;
                x ← Min(palette.artikelmenge − verplant, b.artikelmenge);
                u.artikelmenge ← x
                verplant ← verplant + x
                b.artikelmenge ← b.artikelmenge − x
20:             if alle Unterziele von z belegt || verplant ≥ palette.artikelmenge then
                    Ende der For-Schleife
                end if
            end if
        end for
25:     if verplant < palette.artikelmenge then
            Anzahl der Anbruchpaletten um 1 erhöhen
        end if
        gesammeltebedarfe ← gesammeltebedarfe − verplant
    end procedure
```

ordnet sind, unter dem selben Ziel zusammengefasst werden sollen. In der wirklichen Funktion speichert eine Hilfsdatenstruktur für jeden Bahnhof, ob er schon zu einem Ziel des EHB-Auftrags gehört. Jedes Unterziel ist für den Bedarf eines Palettenauftrags gedacht. Die Unterziele eines Zieles werden sequentiell belegt.

Zum Schluss wird überprüft, ob die Gesamtmenge der verplanten Artikel kleiner als die Artikelmenge auf der Palette ist (Zeile 25). Wenn dies zutrifft, wird die Palette nach ihrer Rückkehr noch die Differenz der Artikelmenge tragen und als Anbruchspalette gespeichert werden müssen. Die Anzahl dieser Paletten wird gespeichert, um sie beschränken zu können.

Der vorliegende Algorithmus stellt zwar ein Kernstück unseres UPPAAL-Modells dar, doch er hat einige Schwächen und kann daher weder als fertig noch als vollständig angesehen werden. Es wurde überlegt, sinnvolle Alternativen zu schaffen, die durch Ausführung paralleler Transitionen unter gleichen oder ähnlichen Bedingungen benutzt werden können.

Kommissionierung

Eine freie Beladezone erhält von einem zufällig ausgewählten Sammler eine EHB-Palette mit EHB-Auftrag, eine Datenstruktur, in der steht, wieviele Artikel sich auf der Palette befinden, welche Artikelart in welchen Mengen zu welchen Bahnhöfen gefahren werden soll und welche Palettenauftragsbedarfe damit erfüllt werden sollen. Daraufhin wird ein Fahrzeug angefordert und die EHB-Palette dem nächsten ankommenden leeren Fahrzeug übergeben. Dieses arbeitet seine Ziele nun sequentiell ab. Beim Durchfahren jeder verzweigenden Weiche teilt das Fahrzeug derselben mit, welcher Bahnhof oder welche Be- und Entladestation sein nächstes Ziel ist. Die Weiche kennt den kürzesten Weg dorthin und schaltet entsprechend. Sobald das Fahrzeug im Zielbahnhof auf dem Entladeplatz festgestellt wurde, teilt es unter Abarbeitung der Unterziele den Palettenaufträgen die benötigte Menge der geladenen Artikelart als gedeckt mit. Die Palettenauftragsprozesse wissen auf diese Weise, wann sie abgearbeitet sind und teilen dies anschließend auch dem zugehörigen LKW-Auftragsprozess mit. Ist dieser erfüllt, kann er es schließlich der Batch mitteilen, die nach Fertigstellung des letzten LKW-Auftrags ebenfalls in den Zustand Fertig wechselt.

4.5 Vorlagen

4.5.1 Vorlage Schiene

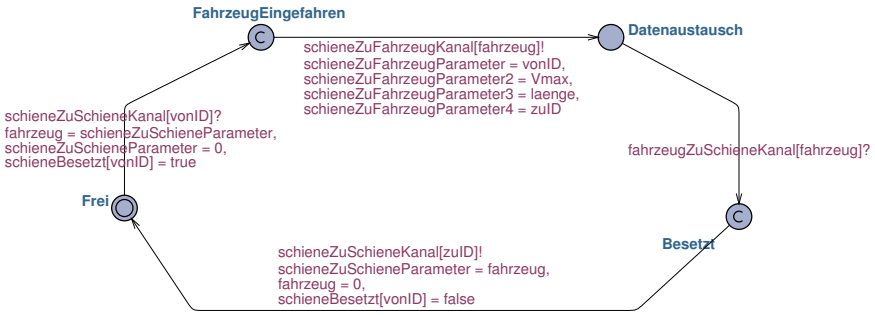


Abbildung 4.3: Vorlage `Schiene`.

Die Vorlage `Schiene` modelliert ein normales Schienenstück, das ein Fahrzeug aufnehmen kann und eine individuelle Länge sowie eine maximale Geschwindigkeit besitzt (vgl. Abbildung 4.3). Über die Parameter `vonID` und `zuID` wird die Schiene in das Modell eingebunden. In der lokalen Variable `fahrzeug` wird die aktuelle FahrzeugID gespeichert.

Befindet sich die Schiene im Zustand `Frei`, kann durch eine vom Vorgängerelement ausgehende Synchronisation ein Zustandswechsel ausgelöst werden. Das aktuelle Schienenstück erhält von seinem Vorgänger die ID des einfahrenden Fahrzeugs. Gleichzeitig wird die boolesche Variable `schieneBesetzt[vonID]` auf `true` gesetzt. Anschließend wird dem eingefahrenen Wagen der gerade überfahrende und der nächste Übergabepunkt sowie die Länge und die maximale Geschwindigkeit des aktuellen Schienenstücks mitgeteilt. Nach dem Speichern dieser Informationen berechnet das Fahrzeug mit Hilfe einer lokalen Uhr die Zeit, die es benötigt, um dieses Schienenstück entlangzufahren.

Zuletzt kommuniziert das Fahrzeug mit der aktuellen Strecke, damit diese sich mit ihrem Nachfolger synchronisieren und das Fahrzeug weiterleiten kann. Nach der Übergabe dieses Fahrzeugs wird die boolesche Variable `schieneBesetzt[vonID]` wieder auf `false` gesetzt, damit andere Fahrzeuge in dieses Schienenstück einfahren können.

4.5.2 Vorlage ZweierWeicheTeilung

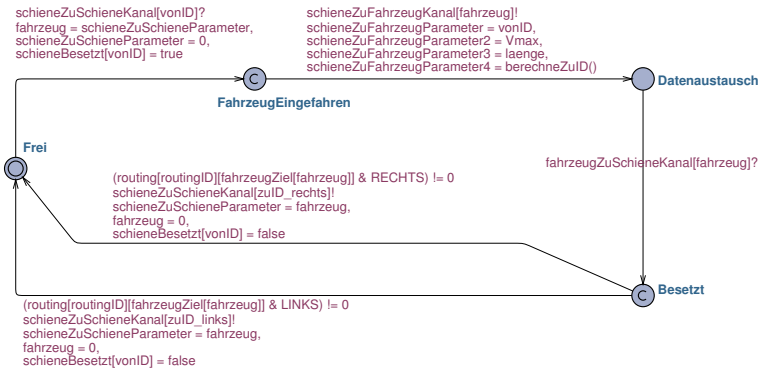


Abbildung 4.4: Vorlage `ZweierWeicheTeilung`.

Die Vorlage `ZweierWeicheTeilung` ist ein spezielles Schienenstück (vgl. Abbildung 4.4). Es erbt einerseits die Eigenschaften einer normalen Schiene, anderseits besitzt es eine spezielle Funktion `berechneZuID()`, d. h. für diese Vorlage wird festgehalten, in welche Richtung ein eingefahrenes Fahrzeug weiter fahren muss (nach links oder nach rechts), um zu einem Bahnhof zu gelangen.

4.5.3 Vorlage ZweierWeicheVereinigung

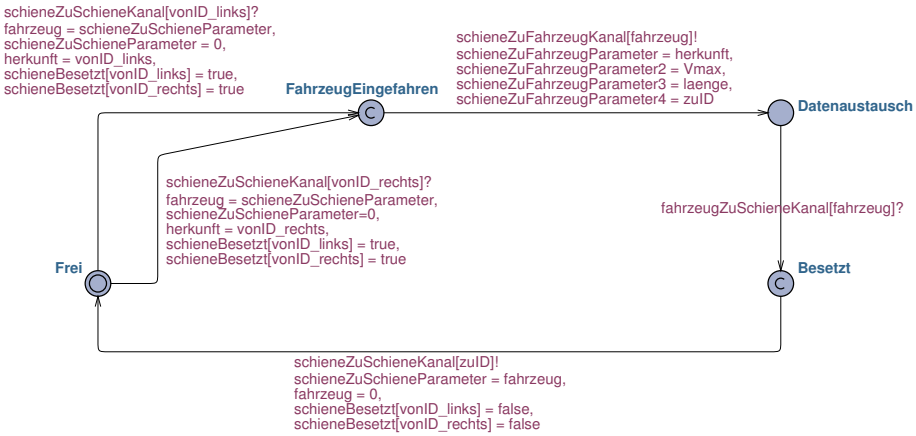


Abbildung 4.5: Vorlage `ZweierWeicheVereinigung`.

Auch diese Vorlage `ZweierWeicheVereinigung` ist ein spezielles Schienenstück (vgl. Abbildung 4.5). Bei der vereinigenden Weiche werden ein linker Vorgänger und ein rechter Vorgänger festgelegt. Im Zustand `Frei` wartet die Schiene dann auf Synchronisationsaufrufe beider Vorgänger. Um eine mögliche Kollision zweier

Fahrzeuge von den beiden Richtungen zu vermeiden, werden die booleschen Variablen `schieneBesetzt[vonIDlinks]` und `schieneBesetzt[vonIDrechts]` nach Synchronisationsaufruf von einer der beiden Vorgänger auf `true` gesetzt. Diese zwei booleschen Variablen werden erst nach der Weiterleitung des eingefahrenen Fahrzeugs zu dem Nachfolger dieses speziellen Schienenstücks wieder auf `false` gesetzt. Der restliche Ablauf entspricht dem der Schienen-Vorlage.

4.5.4 Vorlage DreierWeicheTeilung

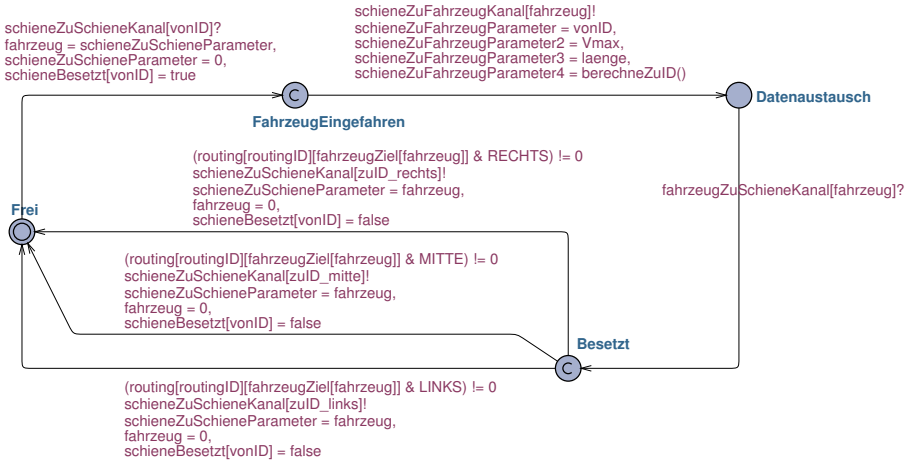


Abbildung 4.6: Vorlage `DreierWeicheTeilung`.

Diese Vorlage stellt eine sich in drei Richtungen teilende Weiche dar (vgl. Abbildung 4.6). Sie entspricht der in zwei Richtungen verteilenden Weiche (vgl. 4.5.2) mit der Ausnahme, dass sie für die dritte Zielrichtung (d. h. zur Mitte) einen weiteren Instanziierungsparameter erhält und vom Zustand `Besetzt` eine dritte Transition nach `Frei` definiert.

4.5.5 Vorlage DreierWeicheVereinigung

Diese Vorlage stellt eine Weiche dar, die drei Streckenteile zu einem vereint (vgl. Abbildung 4.7). Auch hier gibt es zusätzlich zur Vorlage `ZweierWeicheVereinigung` (vgl. 4.5.3) einen Instanziierungsparameter und eine dritte Transition für die Aufnahme eines Fahrzeugs.

4.5.6 Vorlage Fahrzeug

Wenn ein Fahrzeug ein Schienenstück erreicht, wird diesem zunächst über den `schieneZuFahrzeugKanal` die Nummer des gerade überfahrenen Übergabepunktes mitge-

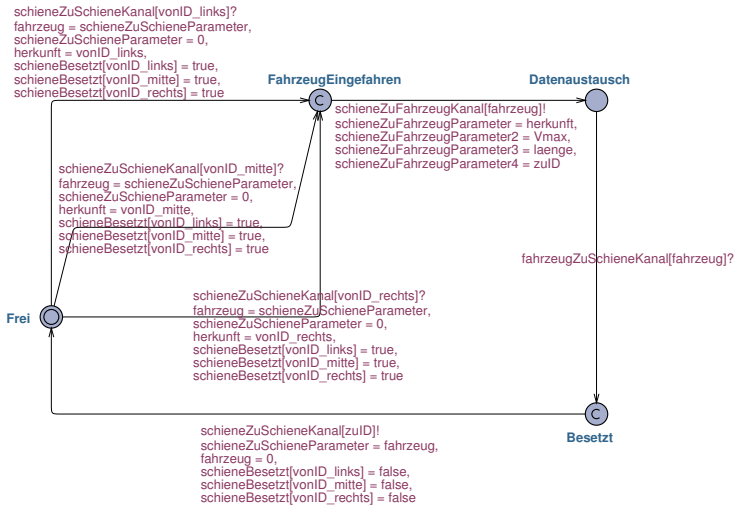


Abbildung 4.7: Vorlage `DreierWeicheVereinigung`.

teilt, die in der lokalen Variable `position` zwischengespeichert wird (vgl. Abbildung 4.8). Diese hat den Wert der `vonID` des Schienenstücks. Gleichzeitig erhält das Fahrzeug auch die Länge, die Höchstgeschwindigkeit sowie den nächsten Übergabepunkt der aktuellen Schiene. Jedes Fahrzeug besitzt eine lokale Uhr, mit der es die Zeit berechnen kann, die es zum Entlangfahren der Schiene benötigt. Sobald die berechnete Fahrzeit überschritten wird, wechselt das Fahrzeug in den Zustand `warteAufWeiterfahrt`. In diesem Zustand verbleibt das Fahrzeug, bis der nächste Übergabepunkt des aktuellen Schienenstücks frei wird.

Die speziellen Schienenstücke `Bahnhof`, `Wartegleis` und `EBZKasten` können gleichzeitig mehrere Fahrzeuge aufnehmen. Über den `fahrzeugUeberspringFahrsequenzKanal` wird dem Fahrzeug von einer dieser drei Vorlagen mitgeteilt, dass es direkt in dieses spezielle Schienenstück einfahren kann.

Um mit einem Bahnhof, einer Entlade- oder Beladezone im `EBZKasten` kommunizieren zu können, muss ein Fahrzeug zunächst über den `fahrzeugBremsKanal` festgestellt werden.

Zum Beladen mit einer neuen Palette erhält das Fahrzeug unter Benutzung des `EBZZuFahrzeugKanal`s Artikelart und -menge sowie die anzufahrenden Ziele durch den Parameter `aktArtikel`. Zur Simulation der realen Zeit, die für einen Beladevorgang benötigt wird, erfährt das Fahrzeug über den `EBZZuBeladeFahrzeugZeitstrafeKanal`, dass es mit Hilfe der lokalen Uhr die Zeit berechnen soll, die es zum Beladen der vom EBZKasten übergegebenen Palette benötigt. Sobald die vorgesehene `BELADEZEIT` überschritten wird und der nächste Übergabepunkt des EBZKastens frei ist, teilt das Fahrzeug dem EBZKasten über den `fahrzeugZuEBZKastenEntsendeKanal` mit, dass es weiter zu den entsprechenden Zielbahnhöfen fahren muss. Gleichzeitig bekommt der EBZKasten die Identifikationsnummer dieses beladenen

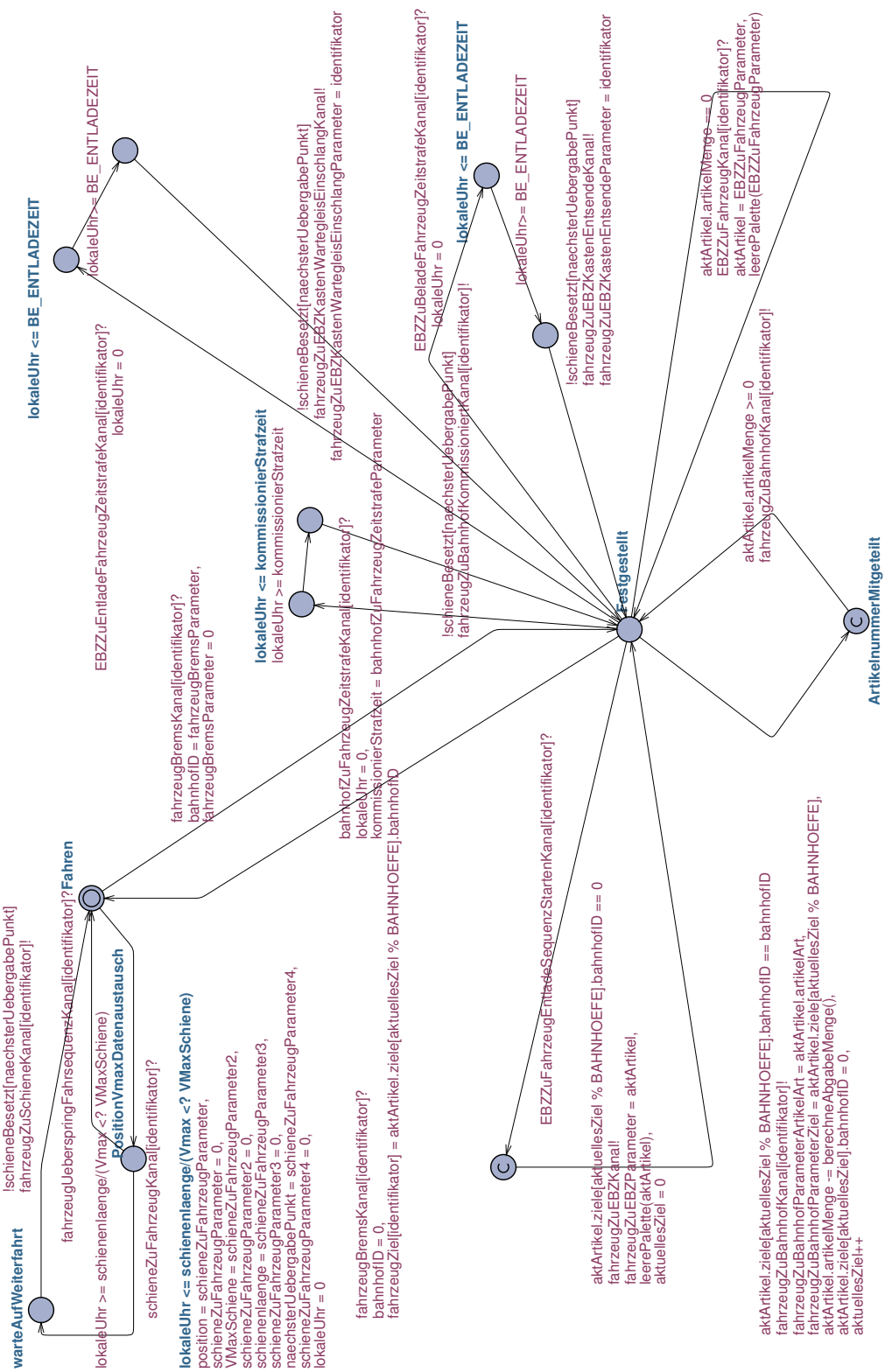


Abbildung 4.8: Vorlage `Fahrzeug`.

Fahrzeugs.

Ist das Fahrzeug in einen seiner Zielbahnhöfe eingefahren, kann die von ihm transportierte EHBPalette dort entladen werden. Über den `fahrzeugZuBahnhofKanal` teilt das Fahrzeug dem Bahnhof mit, welche Artikelart es geladen hat und welche Menge für welchen Palettenauftrag kommissioniert werden muss. Nach dem Kommissionieren berechnet das Fahrzeug mit der Funktion `berechneAbgabeMenge` die Anzahl der insgesamt an diesem Bahnhof abgegebenen Artikelmenge und fährt das nächste Ziel an.

Zur Simulation der realen Zeit, die für das Kommissionieren der vom Fahrzeug transportierten Artikelbenötigt wird, wird das Fahrzeug über den `bahnhofZuFahrzeugZeitstrafeKanal` mitgeteilt, dass es mit Hilfe der lokalen Uhr die Zeit berechnen soll, die zum Kommissionieren dieser Palette benötigt wird. Sobald die berechnete `kommissionierStrafezeit` abgelaufen und der nächste Übergabepunkt des Bahnhofs frei ist, teilt das Fahrzeug dem Bahnhof über den `fahrzeugZuBahnhofkommissionierKanal` mit, dass es weiter zu seinen nächsten Zielbahnhöfen fahren muss.

Nach der Abarbeitung aller Ziele fährt ein Fahrzeug in eine Entladezone im EBZKasten ein. Über den `EBZZuFahrzeugEntladeSequenzStartenKanal` wird dem Fahrzeug mitgeteilt, dass es vollständig entladen werden kann. Zu diesem Zweck teilt das Fahrzeug der Entladezone Artikelart und die verbliebene Artikelmenge mit. Durch die Funktion `leerePalette` wird die übergebene Palette geleert. Um die reale Zeit des Entladens simulieren zu können, erfährt das Fahrzeug über den `EBZZuEntladeFahrzeugZeitstrafeKanal`, dass es durch die lokale Uhr die Zeit berechnen soll, die zum Entladen der mitgebrachten Palette benötigt wird. Sobald die berechnete `ENTLADEZEIT` überschritten wird und der nächste Übergabepunkt des EBZKastens frei ist, teilt das Fahrzeug dem EBZKasten über den `fahrzeugZuEBZKastenWartegleisEinschlangKanal` mit, dass es weiter zum `Wartegleis` fahren muss. Gleichzeitig speichert der EBZKasten die Identifikationsnummer dieses gerade entladenen Fahrzeugs.

Wenn eine Beladezone wieder ein Fahrzeug zur Beladung benötigt, teilt sie dies dem Wartegleis über den `fahrzeugAnforderungsKanal` mit. Anschließend teilt das Wartegleis dem ersten Fahrzeug in der Warteschlange mit, dass es losfahren soll.

4.5.7 Vorlage Bahnhof

Diese Vorlage modelliert einen `Bahnhof`, der auch zu einem Schienenmodell gehört. Ein Bahnhof besitzt eine individuelle BahnhofsID und implementiert zusätzlich einige spezielle Funktionen (vgl. Abbildung 4.9).

Im Vergleich zu normalen Schienenstücken muss ein Bahnhof zunächst über den `bahnhofZuZuordnerKanal` dem Zuordner seine Identifikationsnummer mitteilen, um bei der Zuordnung der Palettenaufträge berücksichtigt zu werden. Außerdem kann ein Bahnhof maximal fünf Fahrzeuge gleichzeitig aufnehmen. Mit Hilfe der Varia-

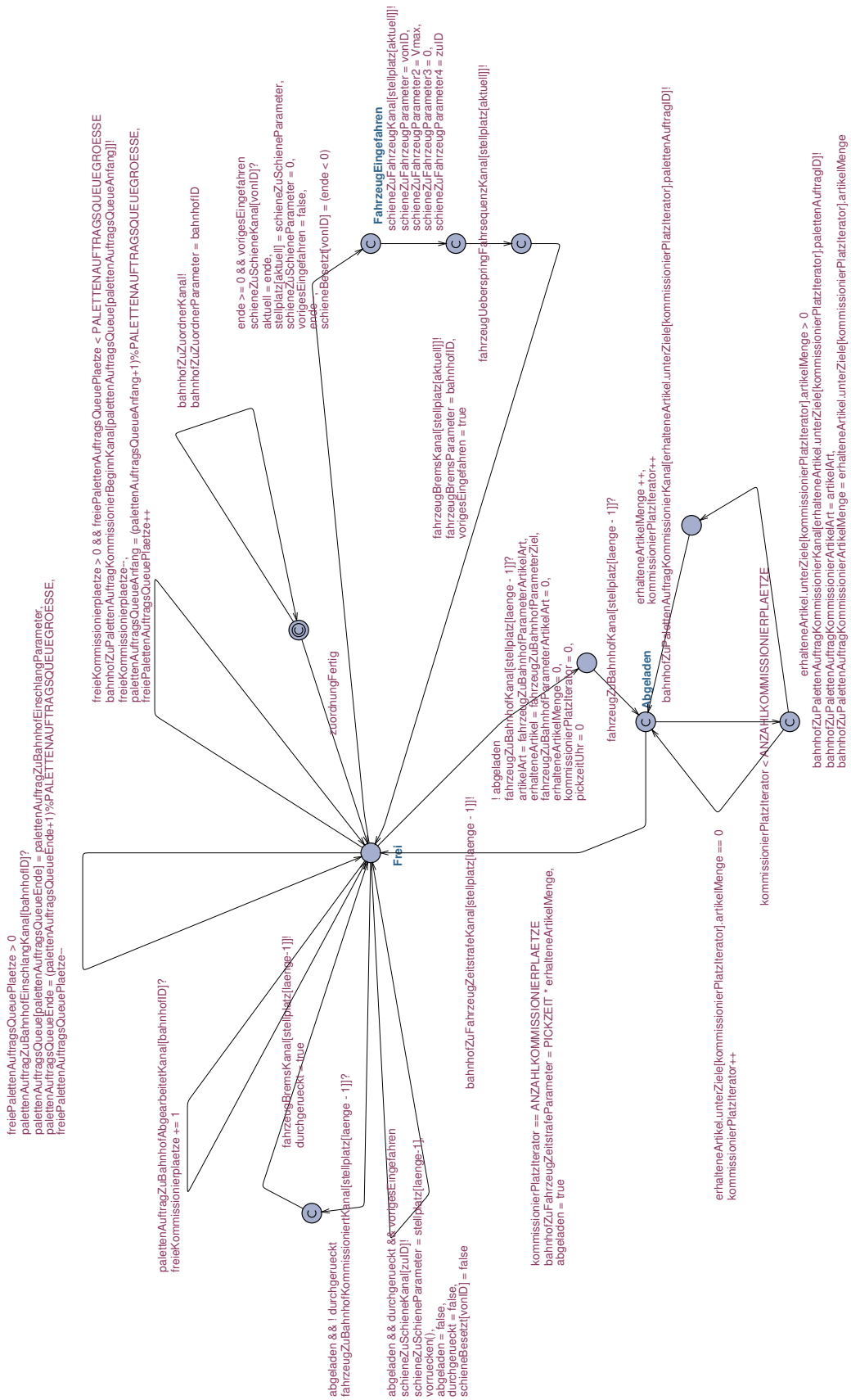


Abbildung 4.9: Vorlage `Bahnhof`.

ble `ende` prüft der Bahnhof, ob er noch einen freien Platz hat, um ein Fahrzeug einfahren zu lassen. Die Variable `ende` speichert die Position, an der das nächste einfahrende Fahrzeug gestellt wird. Ein Fahrzeug kann nur dann einfahren, wenn die Variable `ende` nicht kleiner als Null ist. Über den `schieneZuSchieneKanal` wird der aktuelle Stellplatz im Bahnhof dem eingefahrenen Fahrzeug zugeteilt und die boolesche Variable `vorigesEingefahren` wird am Anfang `false` gesetzt. Zusätzlich bekommt das Fahrzeug den Übergabepunkt, die maximale Geschwindigkeit und die Länge des Bahnhofs übergeben. Danach teilt der Bahnhof über den `fahrzeugBremsKanal[stellplatz[aktuell]]` dem Fahrzeug seine BahnhofsID mit und das Fahrzeug wird an seinem zugewiesenen Stellplatz warten, bis es seine Palette abladen kann. Gleichzeitig wird die boolesche Variable `vorigesEingefahren` gleich `true` gesetzt, damit ein anderes Fahrzeug, vorausgesetzt es existiert ein freier Stellplatz, in den Bahnhof einfahren kann.

Zunächst teilt ein zugewiesener Palettenauftrag, der bereits alle Bedarfe den entsprechenden Sammlern mitgeteilt hat, über den `palettenauftragZuBahnhofEinschlangKanal` dem Bahnhof seine PalettenauftragsID mit. Diese PalettenauftragsID wird in einer Palettenauftragswarteschlange vom Bahnhof gespeichert. Die Anzahl der freien Plätze in dieser Palettenauftragswarteschlange verringert sich um 1.

Ein Bahnhof hat insgesamt sechs Kommissionierplätze. Solange der Bahnhof einen freien Kommissionierplatz hat, teilt er dem ersten Palettenauftrag in Palettenauftragswarteschlange über den `bahnhofZuPalettenAuftragKommissionierBeginnKanal` mit, dass er jetzt kommissioniert wird. Während der Kommissionierung synchronisiert sich der Bahnhof über den `bahnhofZuPalettenAuftragKommissionierKanal` mit diesem Palettenauftrag um zu erfahren, welche von ihm benötigten Artikelart und wie viel davon bereits an seinem Kommissionierplatz im Bahnhof kommissioniert geworden sind. Nachdem ein Palettenauftrag fertig kommissioniert ist, synchronisiert sich dieser über den `palettenauftragZuBahnhofAbgearbeitetKanal` mit dem Bahnhof und gibt einen Kommissionierplatz frei.

Im Prozess des Abladens teilt das am ersten Stellplatz stehende Fahrzeug über den `fahrzeugZuBahnhofKanal[stellplatz[laenge-1]]` mit, welche Artikelart und Artikelmenge von welchem Palettenauftrag an welchem Kommissionierplatz benötigt wird. Wenn dieses Fahrzeug seine Artikel für alle Kommissionierplätze abgeladen hat, wird die boolesche Variable `abgeladen` gleich `true` gesetzt. Um die reale Kommissionierzeit simulieren zu können, teilt der Bahnhof über den `bahnhofZuFahrzeugZeitstrafeKanal` dem Fahrzeug mit, wie viel Zeit benötigt wird, um die entsprechende Artikelmenge von ihm zu kommissionieren. Nach Ablauf dieser Zeit teilt das Fahrzeug über den `fahrzeugZuBahnhofKommissionierKanal` dem Bahnhof mit, dass es den Bahnhof verlassen muss. Die übrigen Fahrzeuge werden durch die Funktion `vorruecken` um einen Stellplatz vorgerückt, damit der Abladeprozess erneut ausgeführt werden kann. Gleichzeitig wird die boolesche Variable `durchgerueckt` gleich `true` gesetzt.

4.5.8 Vorlage EBZKasten

Wenn ein Fahrzeug im EBZKasten entladen werden soll, teilt der Schienenvorgänger des EBZKastens dem EBZKasten über den `schieneZuSchieneKanal` die Identifikationsnummer des entsprechenden Fahrzeugs mit und die Anzahl der zu entladenden Fahrzeuge im EBZKasten erhöht sich um 1. Solange die Anzahl die Kapazität der Entladezone (d. h. die Anzahl der Entladestationen im EBZKasten) nicht überschreitet, können weitere Fahrzeuge in den EBZKasten einfahren.

Nach dem Datenaustausch zwischen dem EBZKasten und dem zu entladenden Fahrzeug (aktueller Übergabepunkt, der Übergabepunkt zum Wartegleis sowie die maximale Geschwindigkeit des EBZKastens) wird der EBZKasten über den `fahrzeugBremsKanal` das Fahrzeug bremsen lassen. Durch den `EBZZuFahrzeugEntladeSequenzStartenKanal` erfährt das Fahrzeug, dass es vollständig entladen werden kann. Danach teilt das Fahrzeug dem EBZKasten seine transportierte Artikelart und -menge mit. Anschließend leitet der EBZKasten diese Informationen über den `EZoneZuSammlerKanal` dem entsprechenden Sammler weiter, damit der Sammler unter Berücksichtigung der auf dem Fahrzeug verbliebenen Artikelmenge eine neue EHB-Palette generieren kann. Durch die Funktion `leerePalette` wird die transportierte Palette aus dem Modell entfernt (auf diese Weise wird das Einlagern einer Anbruchpalette in das Hochregallager simuliert).

Die Vorlage `EBZKasten` modelliert ein spezielles Schienenstück, an dem mehrere Fahrzeuge EHB-Palette abladen sowie aufladen können (vgl. Abbildung 4.10).

Nach der Übermittlung der Informationen über die zu entladenen Artikel an den Sammler synchronisiert der EBZKasten sich mit dem Fahrzeug über den `EBZZuEntladeFahrzeugZeitstrafeKanal` und beginnt das Fahrzeug zu entladen. Sobald die vorgesehene Entladungzeit abgelaufen und der Übergabepunkt zu dem Wartegleis frei ist, teilt das entladene Fahrzeug über den `fahrzeugZuEBZKastenWartegleisEinschlangKanal` dem Wartegleis seine ID mit, damit der EBZKasten nach dem Lösen der Bremse des Fahrzeugs dieses Fahrzeug an das Wartegleis weiterleiten kann. Die Anzahl der noch zu entladenden Fahrzeuge verringert sich um 1.

Der EBZKasten erhält auch die Informationen über die zu ladenen EHB-Paletten von den Sammlern über den `sammlerZuBZoneKanal` (d. h. welche Artikelart und -mengen an welchen Bahnhöfen kommissioniert werden). Dieser synchronisiert sich mit dem Wartegleis über den `fahrzeugAnforderungsKanal` und fordert ein Fahrzeug zur Beladung an. Solange es im EBZKasten noch freie Beladestation gibt und das Wartegleis ein leeres Fahrzeug enthält, bekommt der EBZKasten die FahrzeugID vom Wartegleis mit Hilfe des `schieneZuSchieneKanal`s übergeben und die Anzahl der zubeladenen Fahrzeuge erhöht sich um 1.

Nach dem Datenaustausch zwischen dem EBZKasten und dem zu beladenden Fahrzeug wird das Fahrzeug gebremst. Über den `EBZZuFahrzeugKanal` übergibt der EBZKasten dem Fahrzeug die Informationen über die Beladung und die boolesche

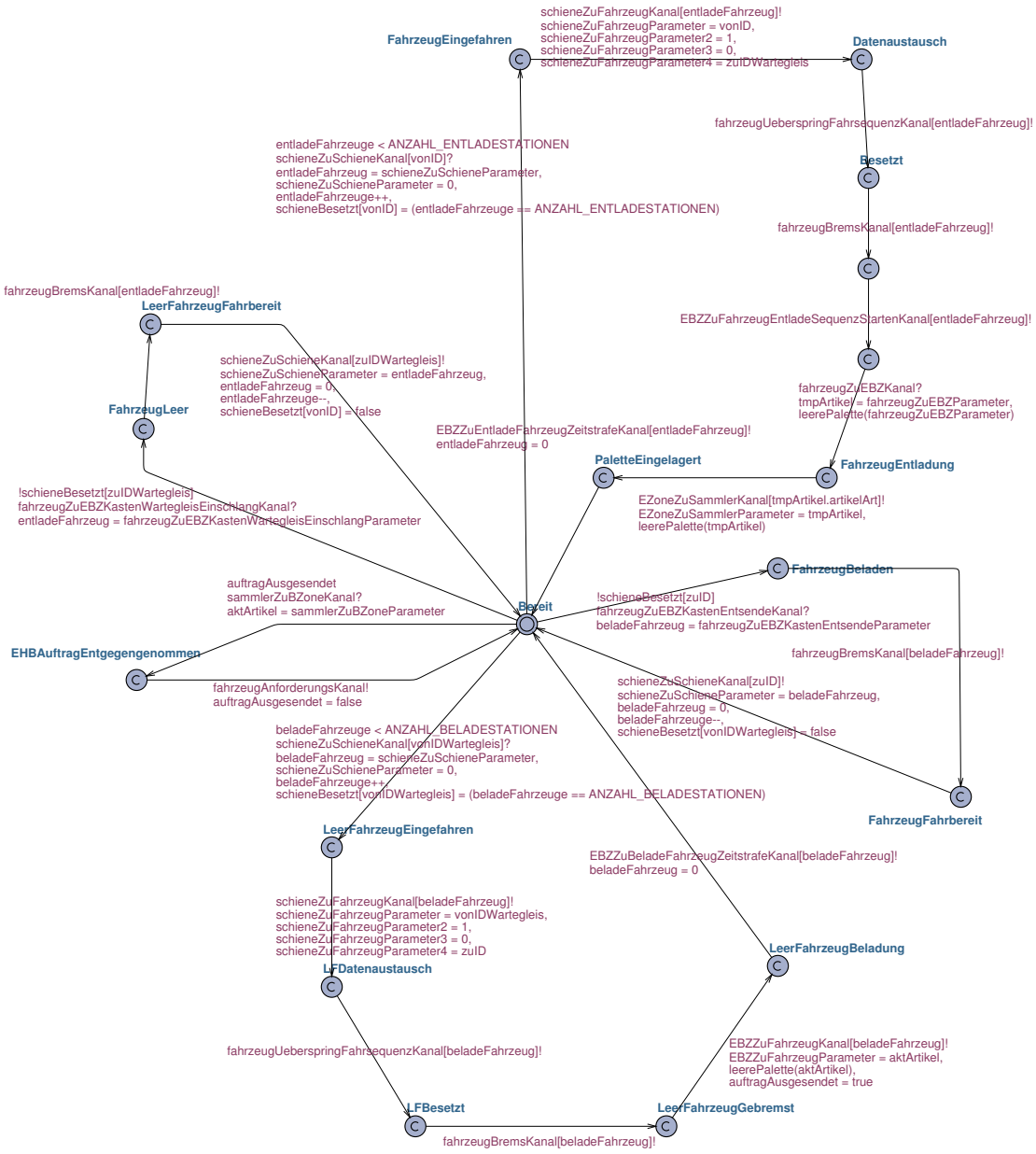


Abbildung 4.10: Vorlage `EBZKasten`.

Variable `auftragAusgesendet` wird auf `true` gesetzt. Das bedeutet, dass der vom Sammler generierte EHB-Auftrag über den EBZKasten dem Fahrzeug mitgeteilt wurde.

Im Anschluss synchronisiert sich der EBZKasten über den `EBZZuBeladeFahrzeug-ZeitstrafenKanal` mit dem Fahrzeug, um die reale Zeit für diesen Beladevorgang zu simulieren. Sobald das Fahrzeug beladen ist und der Schienennachfolger frei ist, teilt das Fahrzeug über den `fahrzeugZuEBZKastenEntsendeKanal` dem EBZKasten seine Identifikationsnummer mit. Nach der Übergabe dieser FahrzeugID an den Nachfolger des EBZKastens fährt das beladene Fahrzeug zu den entsprechenden Zielbahnhöfen. Die Anzahl der noch zubeladenen Fahrzeuge verringert sich um 1.

4.5.9 Vorlage PolyQuelle

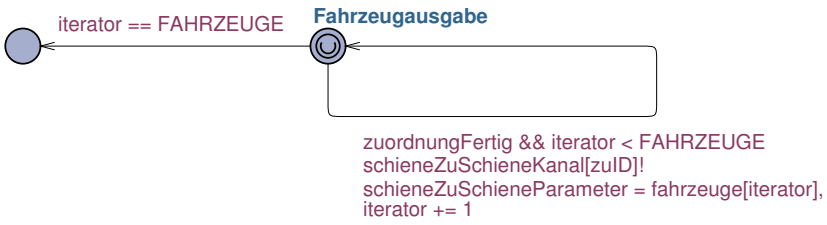


Abbildung 4.11: Vorlage `PolyQuelle`.

Die `PolyQuelle` besteht aus einem einfachen Array, dass die IDs der Wagen enthält, die die Strecke befahren sollen (vgl. Abbildung 4.11). Bevor die Wagen in das Schienenmodell einfahren, müssen alle Palettenaufträge zugeordnet werden. Mit Hilfe der Variable `iterator` wird das Array durchlaufen, um so die Wagen nacheinander auf die Strecke zu setzen.

Sobald alle Fahrzeuge die Polyquelle verlassen haben, wechselt diese in den Endzustand.

4.5.10 Vorlage Batch

Diese Vorlage stellt eine `Batch` dar, die aus einer Menge von LKWAufträgen besteht (vgl. Abbildung 4.12). Eine Batch befindet sich zuerst im initialen Zustand `Bearbeitung`. Wenn einer der zugehörigen LKWAufträge abgearbeitet wurde, teilt er dies seiner Batch über den `LKWAuftragZuBatchKanal` mit und die Anzahl der LKWAufträge verringert sich um 1. Sobald alle zugehörigen LKWAufträge fertig kommissioniert sind, wechselt die Batch in den Zustand `Fertig`.

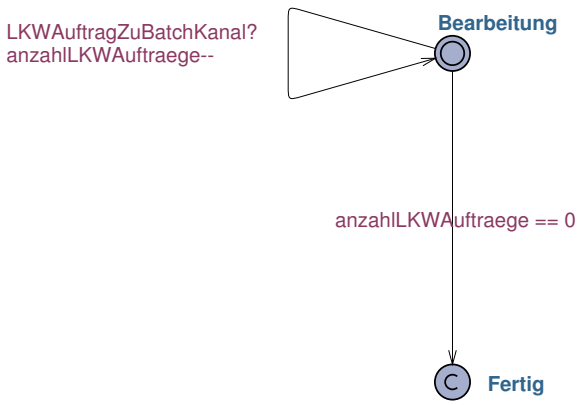


Abbildung 4.12: Vorlage `Batch`.

4.5.11 Vorlage LKWAuftrag

Diese Vorlage modelliert einen `LKWAuftrag`, der aus einer Menge von Palettenaufträgen besteht (vgl. Abbildung 4.13). Ein LKWAuftrag kommuniziert zuerst mit einem der zugehörigen Palettenaufträge über `palettenAuftragZuLKWAuftragKanal` und zählt mit, wie viele Palettenaufträge von ihm schon abgearbeitet wurden. Wenn alle zugehörigen Palettenaufträge fertig sind, synchronisiert er sich mit der Batch und teilt dieser mit, dass er abgearbeitet geworden ist.

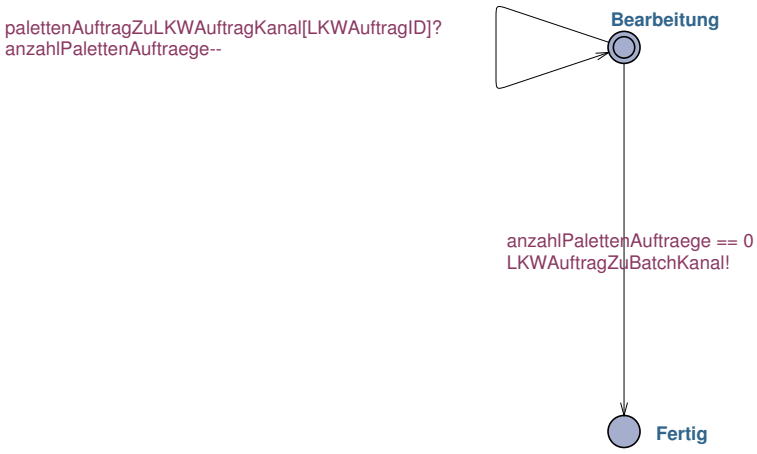


Abbildung 4.13: Vorlage `LKWAuftrag`.

4.5.12 Vorlage Palettenauftrag

Diese Vorlage modelliert einen `Palettenauftrag`, der eine individuelle Palettenauftragsnummer und eine zugehörige LKW-Auftragsnummer enthält (vgl. Abbil-

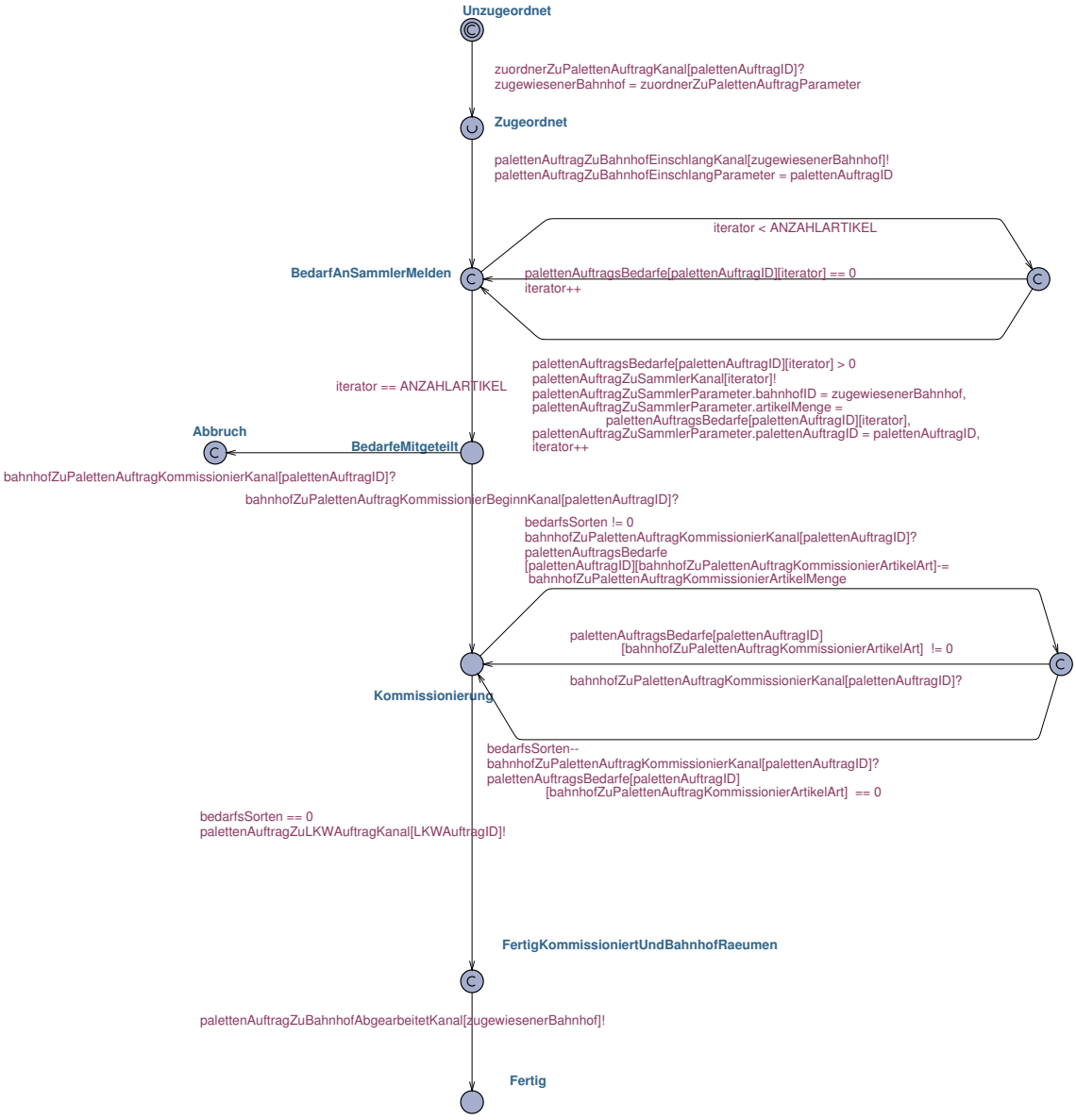


Abbildung 4.14: Vorlage `Palettenauftrag`.

dung 4.14). Über den Parameter `bedarfSorten` werden die Artikelarten, die von diesem Pallettenauftrag benötigt werden, gespeichert.

Einem Palettenauftrag wird zunächst über den `zuordnerZuPalettenAuftrag-Kanal` vom Zuordner mitgeteilt, an welchem Bahnhof er kommissioniert wird. Dann wechselt der Palettenauftrag in den Zustand `zugeordnet` und synchronisiert sich mit seinem zugeteilten Bahnhof über den `palettenAuftragZuBahnhofEinschlang-Kanal`, so dass seine PalettenauftragsID in der `palettenAuftragsQueue` dieses Bahnhofs gespeichert wird (vgl. 4.5.7).

Danach teilt der Palettenauftrag den entsprechenden Sammlern mit, welche Artikelarten und -menge von ihm gebraucht werden. Gleichzeitig bekommt der Sammler (vgl. 4.5.14) die Identifikationsnummer des Palettenauftrags. Sobald der Palettenauftrag kommissioniert werden kann, synchronisiert sich der entsprechende Bahnhof mit dem Palettenauftrag über den `bahnhofZuPalettenAuftragKommissionierBeginn-Kanal` und der Palettenauftrag wechselt in den Zustand `Kommissionierung`. Über den `bahnhofZuPalettenAuftragKommissionierKanal` erfährt der Palettenauftrag, welche von ihm benötigte Artikelart und wie viel davon an seinem Kommissionierplatz im Bahnhof schon kommissioniert worden sind.

Sobald alle benötigten Artikelmengen der entsprechenden Artikelart für diesen Palettenauftrag fertig kommissioniert wurden, teilt der Palettenauftrag seinem zugehörigen LKWAuftrag mit, dass der Kommissionierprozess dieses Palettenauftrags abgeschlossen ist. Der Palettenauftrag wechselt in den Zustand `FertigKommissioniert-UndBahnhofRaeumen`.

Am Ende synchronisiert sich der Palettenauftrag mit dem zugewiesenen Bahnhof über den `palettenAuftragZuBahnhofAbgearbeitetKanal`, damit der Bahnhof wieder einen freien Kommissionierplatz bekommt, an dem weitere Palettenaufträge kommissioniert werden können.

Wenn es vorkommt, dass ein Palettenauftrag schon alle Bedarfe an die entsprechenden Sammlern gemeldet hat und sich im Zustand `BedarfMitgeteilt` befindet, aber noch keinen Kommissionierplatz zugeteilt bekommen hat, kann das vordere Fahrzeug, dessen Artikel für diesen Auftrag benötigt werden, nicht kommissioniert werden. Da sich die anderen Fahrzeuge aufstauen, wechselt der Palettenauftrag direkt von dem Zustand `BedarfMitgeteilt` in den Zustand `Abbruch`. Die Simulation wird an dieser Stelle abgebrochen (Deadlock).

4.5.13 Vorlage Zuordner

Diese Vorlage modelliert einen `Zuordner`, der die vorhandenen Palettenaufträge den Bahnhöfen zuordnet (vgl. Abbildung 4.15). Jeder Bahnhof synchronisiert sich über den `bahnhofZuZuordnerKanal` mit dem Zuordner und teilt diesem seine BahnhofsID mit. Danach befindet sich der Zuordner im Zustand `BahnhofWillig` und übergibt einem Palettenauftrag eine BahnhofsID über den `zuordnerZuPalettenAuftragKanal`. Der lokale Parameter `iterator` berechnet, wie viele Palettenaufträge schon zugeord-

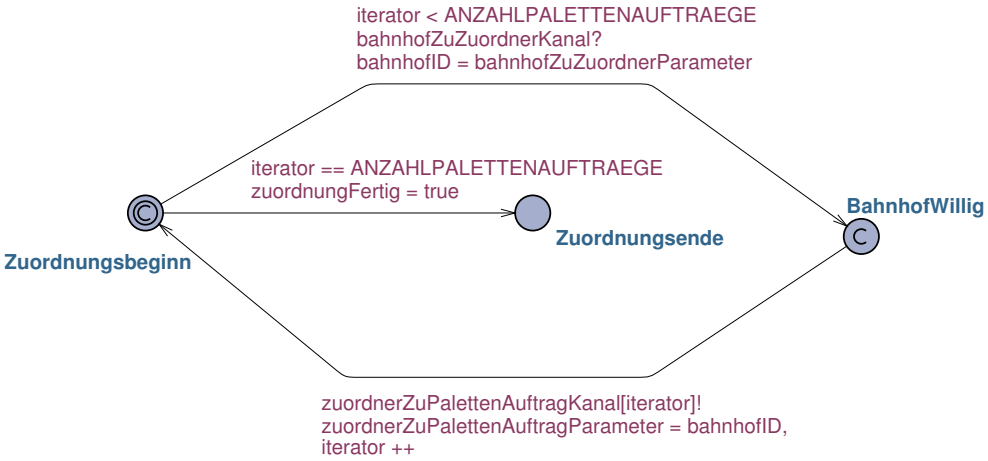


Abbildung 4.15: Vorlage `Zuordner`.

net geworden sind. Sobald alle Palettenaufträge eine BahnhofsID bekommen haben, wechselt der Zuordner in den Zustand `Fertig`.

4.5.14 Vorlage Sammler

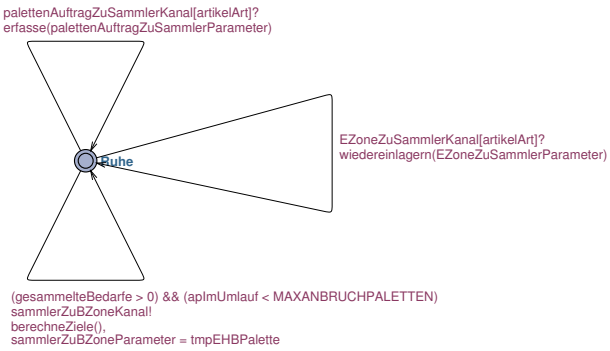


Abbildung 4.16: Vorlage `Sammler`.

Diese Vorlage modelliert einen `Sammler`, der die Bedarfe einer einzigen Artikelart sammelt (vgl. Abbildung 4.16). Jeder zugeordnete Palettenauftrag wird zunächst über den `palettenauftragZuSammlerKanal` dem entsprechenden Sammler mitteilen, welche Menge an welchem Bahnhof von seiner Artikelart benötigt wird. Danach addiert der Sammler durch die Funktion `erfasse` den gesamten Bedarf dieser Artikelart über alle Palettenaufträge. Anschließend teilt er durch die Funktion `berechneZiele` der Beladezone im EBZKasten die Ziele der nächsten auszusendenden Palette sowie weitere Informationen (die entsprechenden Artikelarten und -menge) über die Ziele mit.

Wenn ein Fahrzeug an einer Entladezone im EBZKasten die von ihm mitgebrachte Palette entladen hat, wird die auf dieser Palette noch vorhandene Artikelmenge über den `EZoneZuSammlerKanal` zum entsprechenden Sammler übermittelt, damit der Sammler durch die Funktion `wiedereinlagern` die entsprechende Anbruchpalette wieder einlagern kann.

4.5.15 Vorlage Wartegleis

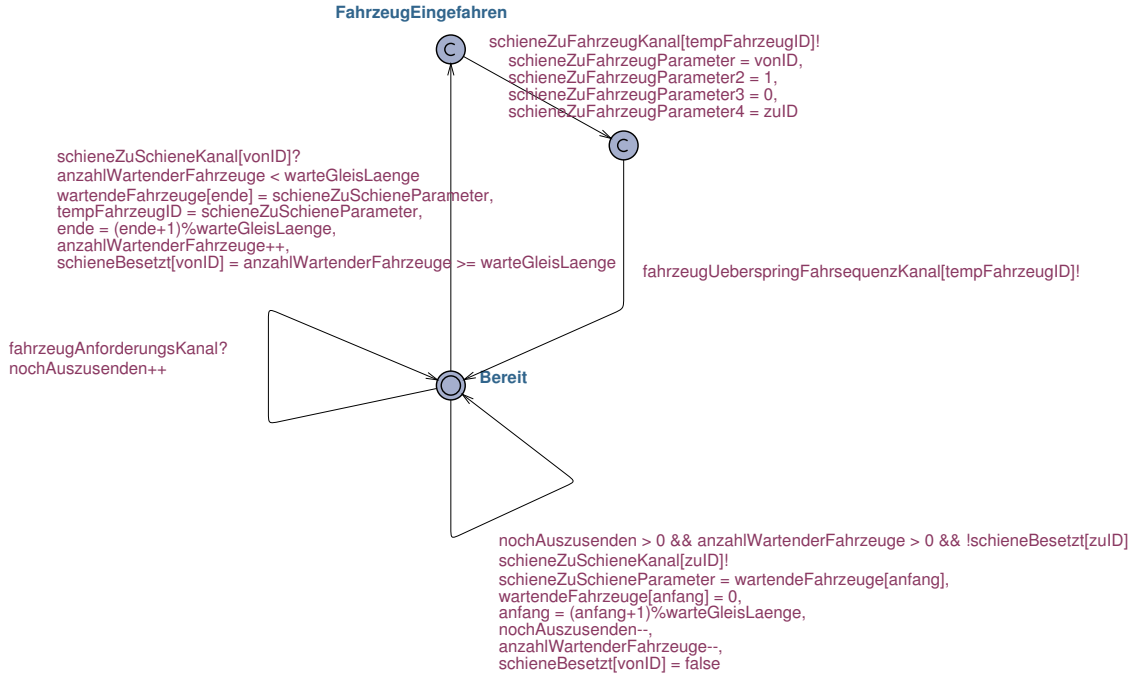


Abbildung 4.17: Vorlage `Wartegleis`.

Diese Vorlage modelliert das spezielle Schienenstück `Wartegleis`, das beliebig viele Fahrzeuge aufnehmen kann. Zusätzlich besitzt es die Eigenschaften eines normalen Schienenstücks (vgl. Abbildung 4.17). Die leeren Fahrzeuge, die ihre Ziele abgearbeitet haben, fahren hier ein. Die Variable `ende` speichert die Position des Stellplatzes, an dem das zuletzt eingefahrene Fahrzeug wartet. Die Variable `tempFahrzeugID` speichert die Identifikationsnummer des zuletzt eingefahrenen Fahrzeugs. Das erste Fahrzeug im Wartegleis wird vom Wartegleis entsendet, wenn dem Wartegleis vorher über den `fahrzeugAnforderungsKanal` von der Beladezone im EBZKasten mitgeteilt wurde, dass diese wieder ein Fahrzeug zur Beladung benötigt. Die Variable `nochAuszusenden` speichert die Anzahl der noch auszusendenen Fahrzeuge. Nach der Ausfahrt eines Fahrzeugs verringert sie sich diese um 1 und das nächste Fahrzeug im Wartegleis kommt an den ersten Stellplatz.

4.6 Anpassung und Benutzung des Modells

Mit den erarbeiteten Automatenvorlagen lassen sich nach Belieben Schienenmodelle zusammenbauen. Dazu müssen im Wesentlichen eine Reihe von ganzzahligen Konstanten gesetzt und passende Prozesse instanziiert werden. Für komplexere Modelle empfehlen wir den Modelleditor von UppaalVis.

4.6.1 Erstellen der EHB-Strecke

Für jedes geplante Schienenelement, wie etwa Geraden, Weichen oder Bahnhöfe muss ein entsprechender Prozess instanziiert werden. Je nach Prozessvorlage gibt es verschiedene Parameter, die anzugeben sind. Um zwei Schienenstücke im Modell „zusammenzustecken“, so dass ein Fahrzeug von einem zu nächsten fahren könnte, werden dem `zuID`-Parameter des einen und dem `vonID`-Parameter des anderen Stückes die selben Zahlen zugewiesen. Diese Nahtstellen zwischen den Schienen sind die Übergabepunkte. Ihre Numerierung beginnt bei 0. Natürlich haben Weichen je nach Art mehr als eine `zuID` oder `vonID`. Weitere bei den meisten Schienenautomatenvorlagen nötige Parameter sind die Streckenlänge (`laenge`) und die Maximalgeschwindigkeit (`Vmax`), mit der das Schienenstück zu befahren ist.

Verzweigende Weichen benötigen eine `routingID`, um die für sie relevante Zeile der Routingtabelle zu identifizieren. Natürlich muss die Routingtabelle (`routing[][]`) entsprechend gesetzt werden.

Bahnhöfe benötigen zusätzlich eine `bahnhofID`.

Der Fahrzeugquelle muss ein Array übergeben werden, in dem für jedes Fahrzeug die Identifikationsnummer eingetragen ist. Dies ist nötig, damit der Fahrzeugquellprozess die Fahrzeuge zu Beginn korrekt ins Schienenmodell einfahren lassen kann. Zusätzlich benötigt jedes Fahrzeug aber auch einen eigenen Prozess, dem bei Instanziierung Fahrzeugnummer (`identifikator`) und Maximalgeschwindigkeit (`Vmax`) mitgegeben werden müssen.

Der an das Hochregallager angrenzende Bereich von Be- und Entladezonen für EHB-Fahrzeuge wird durch einen Prozess namens `EBZKasten` realisiert. Seine Instanziierung erfordert ein weiteres Paar Übergabepunkte für den Anschluss eines Wartegleises, auf dem nicht benutzte EHB-Fahrzeuge geparkt werden.

Weitere eventuell anzupassende Konstanten sind:

- `BAHNHOEFE` für die Anzahl der Bahnhöfe,
- `PALETTENAUFTRAGSQUEUEGROESSE` (siehe Abschnitt 4.5.12),
- `FAHRZEUGE` für die Anzahl der Fahrzeuge,
- `VMAX` als globale Maximalgeschwindigkeit,
- `UEBERGABEPUNKTE` für die Anzahl derselben,

- ANZAHL_VERZWEIGENDE_WEICHEN,
- ANZAHLKOMMISSIONIERPLAETZE für die Anzahl der Kommissionierplätze in einem Bahnhof,
- PICKZEIT als Zeit, die für das Bewegen einer Artikeleinheit durch den Kommissionierer benötigt wird,
- ANZAHL_ENTLADESTATIONEN und ANZAHL_BELADESTATIONEN, die die Anzahlen der gleichzeitig im EBZ-Kasten ent- und beladbaren Fahrzeuge angeben.

4.6.2 Erstellen des Hochregallagers

Das eigentliche Hochregallager mit den Informationen, welche Paletten welcher Artikelart voll oder angebraucht in ihm enthalten sind, wird durch Sammlerprozesse simuliert. Für jede Artikelart gibt es genau einen Sammler, woraus sich der (einzige) Parameter artikelArt ergibt.

Anpassbare Konstanten sind hier:

- PALETTENKAPAZITAET als Artikeleinheitenmenge, mit der volle Paletten das Hochregallager verlassen,
- MAXANBRUCHPALETTEN für die höchstens erlaubte Anzahl angebrochener Paletten einer Artikelart,
- BE_ENTLADEZEIT für die Zeit, die benötigt wird, eine Palette auf ein EHB-Fahrzeug zu laden oder herunterzunehmen.

4.6.3 Erstellen der Batch

Die Palettenaufträge werden in einem großen globalen Bedarfsarray (palettenAuftragsBedarfe[][]) kodiert. Für jeden Palettenauftrag gibt es eine Zeile, für jede Artikelart eine Spalte. Ein Eintrag selbst ist eine Ganzzahl und stellt den Bedarf des entsprechenden Palettenauftrags an der entsprechenden Artikelart dar.

Für jeden Palettenauftrag muss es einen Prozess geben, der über seine palettenAuftragID die zutreffende Zeile im Bedarfsarray kennt. Der Parameter bedarfsSorten muss der Anzahl der verschiedenen Artikelarten entsprechen, an denen der Palettenauftrag einen Anfangsbedarf größer als 0 hat. Durch den Parameter LWKAuftragID kennt der Prozess seinen LKW-Auftrag.

Für jeden LKW-Auftrag hat einen Prozess zu existieren, dem Parameter LKWAuftragID und anzahlPalettenAuftraege mitgegeben werden müssen. Schließlich muss es noch einen parameterlos erzeugten Prozess für die Auftragsbatch geben.

Alle erzeugten Prozesse werden nun in der Systemdeklaration aufgeführt und damit als dem System zugehörig markiert. Hierbei ist eine Priorisierung der Fahrzeuge eventuell sinnvoll (vgl. Abschnitt 4.7.2).

4.7 Optimierungen

4.7.1 Speicherverbrauch senken

Der Zustand eines Uppaal-Systems ist festgelegt durch die Orte, in denen sich seine Prozesse befinden sowie durch die Wertebelegung der Uhren und Variablen. Der Speicherverbrauch eines Trace, also einer Folge von Systemzuständen, ist davon abhängig, wie viele Prozesse, Uhren und Variablen es im System gibt und wie viele Zustände der Trace enthält.

Um Speicher zu sparen, wurden daher sooft wie möglich Konstanten anstatt Variablen benutzt. Uhren in den Schienenstücken für die Bestimmung der Fahrzeuggeschwindigkeit wurden in die Fahrzeuge verlagert, weil es üblicherweise weniger Fahrzeuge als Schienenstücke gibt. Die zunächst modellierten Be- und Entladezonenvorlagen, von denen es im großen Modell jeweils 14 Stück hätte geben sollen, wurden gegen eine konfigurierbare Blackbox (*EBZKasten*) ausgetauscht.

Während in älteren Versionen als Uppaal 4 je ein Sammler für eine Artikelart nötig war, um zufällige Reihenfolgen bei der Artikelbehandlung zu ermöglichen, wäre dies nun nicht mehr nötig. Die neue Programmversion ermöglicht die Generierung von Zufallszahlen („nichtdeterministische Wertzuweisung für Ganzzahlvariablen“), mit der man zur Systemlaufzeit zufällige Artikelarten auswählen könnte. Die Aufteilung in je einen Sammler pro Artikelart könnte dadurch einer monolithischen Waren-/Hochregallagerverwaltung weichen.

4.7.2 Entscheidungsmöglichkeiten dezimieren

In Systemen mit mehreren Prozessen, die mehr oder weniger unabhängig voneinander und an beliebigen Zeitpunkten Zustandsübergänge vollziehen können, muss der Verifizierer auch alle möglichen Reihenfolgen der Zustandübergänge berücksichtigen. Dies vergrößert den Zustandsraum des Systems natürlich immens.

Committed-Abläufe

Die Markierung des Ortes einer Automatenvorlage als *committed* erlaubt die bevorzugte Behandlung der ausgehenden Transitionen dieses Zustands (vgl. auch Abschnitt 2.2.3). Die möglichen Transitionen sind auf diejenigen beschränkt, die Prozesse in einem als *committed* markierten Ort betreffen. Ist eine Folge von Orten einer Automatenvorlage markiert, kann man es erreichen, dass die zugehörigen Transitionen zusammenhängend ausgeführt werden, so als wären sie atomar gewesen.

Sequenzialisierungen von committed-Folgen

Wenn sich jedoch bei einer Synchronisation mehrere Prozesse eines System gleichzeitig in committed-Zustände begeben, danach aber unabhängig weiterlaufen, so

verschwindet der Vorteil der Priorisierung wieder weitgehend. Hier hilft eine Sequentialisierung zu bevorzugender Abläufe. Dies soll an einem Beispiel deutlich gemacht werden, dessen Verwirklichung mit 20% weniger Verifikationsrechenzeit einhergingen (bei der Berechnung eines schnellsten, die Batch erfüllenden Systemablaufs):

Falls beispielsweise in einem Bahnhof einige Einheiten eines bestimmten Artikels angenommen wurden, werden sie im Anschluss auf die im Bahnhof in Kommission befindlichen Paletten je nach Bedarf verteilt. Der Bahnhofsprozess geht der Reihe nach die Paletten durch und schaut, ob Bedarf an dem Artikel besteht. Wenn Bedarf besteht, wird dem Palettenauftrag die passende Menge übergeben. Dieser muss natürlich seinen Restbedarf umgehend anpassen. Hier wäre es praktisch, wenn der gesamte Ablauf der Verteilung in Nullzeit und bevorzugt stattfände. Die zunächst implementierte triviale Lösung, alle beteiligten Transitionen auf *committed* zu setzen, führte leicht zu dem beschriebenen Szenario, dass sich ein Bahnhof und ein Palettenauftrag beide in einer zu bevorzugenden Folge von Zuständen befanden, und deshalb wieder alle möglichen Reihenfolgen berücksichtigt werden mussten. Eine viel bessere Lösung ist es, wenn der Bahnhof bei der Übergabe der Ware an den Palettenauftrag (gemeinsame Synchronisation mit Datenübergabe) seine committed-Folge durch nicht als *committed* markierte Zustände unterbricht. Der Palettenauftrag arbeitet nun seinerseits eine committed-Folge ab und führt nach Aktualisierung der Daten wieder eine gemeinsame Synchronisation mit dem Bahnhof durch. Nun kann dieser eine neue Folge zu bevorzugender Zustandsübergänge beginnen und damit die Verteilung in Nullzeit weiterführen.

Scharfe Zeit

Weil es unserer Erfahrung nach beim Fahren im Schienensystem noch sehr häufig auftrat, dass der Verifizierer entscheiden konnte, welches Fahrzeug zuerst fahren durfte, haben wir dieses Phänomen genauer untersucht. Üblicherweise lag der Wert der Systemuhr dabei in einem Intervall, das so groß war, dass sowohl das eine als auch das andere Fahrzeug zuerst fahren konnte, während die inneren Uhren beider Fahrzeuge gewöhnlich nicht in den gleichen Intervallgrenzen lagen.

Um die Zeitpunkte des Fahrens weiter einzuschränken, wurde eine Invariante auf den Wartezustand gelegt als Ergänzung zum entgegengesetzten Wächter auf der ausgehenden Transition. Ebenso wurde an anderen Stellen im Modell, an denen uhrenabhängige Prozesse ablaufen („Zeitstrafen“ für Be-/Entladen, Kommissionieren) der Fortschritt über neu eingesetzte Invarianten erzwungen. Dies führte zum derzeitigen Systemverhalten, dass die Uhren des Modells stets zu präzisen Zeitwerten zurückkehren. Durch dieses Erzwingen des Weiterfahrens eines Fahrzeug, sobald es möglich wird, kommt es nur noch bei absolut synchron fahrenden Fahrzeugen vor, dass mehrere dieser gleichzeitig fahren können.

Priorisierungen

Mit der neuen Version 4 von Uppaal gibt es die Möglichkeit, Kanäle und Prozesse zu priorisieren, so dass im Falle eigentlich zur selben Zeit möglicher Zustandsübergänge der Verifizierer anhand der Prioritäten der beteiligten Prozesse oder der benutzten Synchronisationskanäle entscheidet, welcher Zustandsübergang ausgeführt wird.

Die Fahrzeugprozesse werden seitdem unterschiedlich priorisiert, so dass in keinem Systemzustand zwei Fahrzeuge gleichzeitig fahren können.

4.8 Korrektheitstests des Modells

Um ein korrektes Funktionieren des Uppaal-Modells mit hoher Wahrscheinlichkeit zusichern zu können, wurden einige Tests durchgeführt. Im Gegensatz zum Testen von Methoden in Programmiersprachen, für die sich Quasistandards etabliert haben – als Beispiel sei das JUnit-Framework [JUN] angeführt – musste für das Uppaal-Modell ein eigenes Konzept entwickelt werden.

Zunächst wird eine Strukturierung in *Kategorien*, *Teilaspekte* und *Testfälle* vorgenommen, die zur Übersichtlichkeit beitragen und damit eine vollständige Identifizierung von Testfällen ermöglichen soll. Die genannten Begriffe werden im nächsten Kapitel eingeführt.

Kapitel 4.8.2 vermittelt anschließend allgemeine Strategien bei der Entwicklung von Testfällen.

In dem darauf folgenden Kaptiel 4.8.3 werden durchgeführte Tests beschrieben und in Kapitel 4.8.4 finden sich weitere identifizierte Testfälle.

4.8.1 Strukturierung

Das Uppaal-Modell beinhaltet einige wichtige Kernfunktionalitäten. Diese sind

- Auftragszuordnung,
- EHB-Auftragsgenerierung,
- Belade- und Entladevorgänge,
- Fahrvorgänge und
- Auftragserfüllung.

Im Zusammenhang mit den Tests sprechen wir auch von *Kategorien*.

Die Auftragszuordnung ist die Funktionalität der Zuordnung von Palettenaufträgen zu Bahnhöfen. Jeder Palettenauftrag muss vor Beginn einer Simulation einem Bahnhof zugeordnet werden, an dem er kommissioniert werden soll.

Die EHB-Auftragsgenerierung übernimmt im Modell die Funktion eines Warehousemangementsystems. Sie sorgt dafür, dass EHB-Fahrzeuge mit (Liefer-)Aufträgen versehen werden. Sie weist also einzelne EHB-Fahrzeuge an, eine Palette mit einer gewissen Artikelart vom HRL zu einem oder mehreren Bahnhöfen zu fahren und dort bestimmte Palettenaufträge zu beliefern.

Unter Belade- und Entladevorgängen werden alle Beladevorgänge am HRL und Entladevorgänge an Bahnhöfen, das Kommissionieren, und die Wiedereinlagerung von Anbruchpaletten in das HRL zusammengefasst.

Mit Fahrvorgängen wird jegliche Art von EHB-Fahrzeugbewegung, also auf einfachen Schienen, aber auch in Bahnhöfen und im Wartegleis bezeichnet.

Die Auftragserfüllung fasst zusammen, inwieweit Palettenaufträge, LKW-Aufträge und letztendlich die Batch abgearbeitet sind.

Für jede Funktionalität lassen sich mehrere, im Folgenden *Teilaspekte* genannte, Vorgänge oder Teilfunktionalitäten identifizieren, die in ihrer Gesamtheit die Kernfunktionalität realisieren. Beispielsweise sei hier die Auftragserfüllung genannt, die aus den Teilaspekten „Erfüllung eines Palettenauftrages“, „Erfüllung eines LKW-Auftrages“ und „Erfüllung der Batch“ besteht.

Um die Korrektheit eines Teilaspektes zu überprüfen, sind potentiell mehrere Testfälle notwendig. Am deutlichsten wird das, wenn man sich überlegt, dass in der Kategorie „Fahrvorgänge“ der Teilaspekt „Einfahrt in ein Schienenstück“ in verschiedenen Testfällen für einfache Schienen, Bahnhöfe, Weichen und andere Schienenelemente getestet werden muss.

4.8.2 Vorgehensweise bei der Durchführung der Tests

Tests von Vorlagen

Die allgemeine Vorgehensweise stellt sich grob wie folgt dar: Es werden Testautomaten entwickelt, die mit einigen der Vorlagen des eigentlichen Uppaal-Modells interagieren. So wird ein lauffähiges Testsystem konstruiert, das durch die Testautomaten, auch mithilfe zusätzlich eingeführter Synchronisationen und Parameterübergaben, in bestimmte (Initial-)Zustände gesetzt werden kann. Auf diesen Testsystemen können dann Modelcheckinganfragen gemacht werden, um Eigenschaften zu verifizieren.

Bei der im vorherigen Kapitel gemachten Einteilung deutete sich schon an, dass es häufig günstig ist, Vorlagen im Zusammenspiel zu testen, so wird z. B. das Einfahren in ein Schienenstück mit einem Fahrzeug und einer Weiche überprüft, und dass ein isoliertes Testen einer einzigen Vorlage eher eine Ausnahme darstellen wird.

Der Vorteil dieser Vorgehensweise liegt darin, dass Funktionalitäten, die erst durch ein komplexes Zusammenspiel von mehreren Prozessen ermöglicht werden, als Ganzes überprüfbar sind. Es würde zudem in einem erheblichen Mehraufwand resultieren, würden die Tests nur an den Schnittstellen von einzelnen Vorlagen ansetzen. Auch würde dadurch die Komplexität und die Anzahl der Testvorlagen steigen,

wodurch sich potentiell mehr Fehler in das Testsystem einschleichen könnten, als aufgedeckt würden. Daher könnte dieses Vorgehen evtl. mehr Schaden als Nutzen bringen.

Grenzen des Modelcheckings

Uppaal bietet leider nur eine eingeschränkte Variante der CTL (s. Kapitel 2.2.4). So ist es u. a. nicht möglich, einen (Weak-)Until-Operator zu nutzen. Dieser wäre in einigen Situationen sehr hilfreich, beispielsweise um zu überprüfen, ob eine Eigenschaft vor einer Aktion vorhanden oder nicht vorhanden ist.

Stattdessen muss man sich mit einigen Tricks behelfen, die immer abhängig vom konkreten Testfall sind. Es ist denkbar, Flags einzusetzen, die während oder nach Ausführung festgelegter Aktionen gesetzt werden und bei Modelcheckinganfragen benutzt werden können, um entsprechende Schlüsse zu ziehen. Beispiele für dieses Vorgehen gibt es in den durchgeführten Testfällen (Kapitel 4.8.3) zu genüge. Dort wird explizit auf eine Einführung solcher Flags hingeweisen.

Weiterhin sind manche Vorgänge im Modell zu kompliziert, als dass sie durch eine Formel, dazu noch in der eingeschränkten CTL-Variante von Uppaal, genau erfasst werden können. In solchen Fällen sollte man sich damit zufrieden geben, Eigenschaften vor und nach einem Vorgang zu überprüfen, quasi nach dem Black-Box-Modell nur das „Ein-/Ausgabeverhalten einer Menge von Vorlagen“ zu beobachten. Auch hier bringt eine Einführung von Hilfsvariablen oft großen Nutzen.

Tests von Funktionen

Die Uppaal-Versionen ab 3.6 (vgl. Kapitel 2.2.5) unterstützen programmiersprachliche Konstrukte wie Funktionen, die auch Einzug in das Modell erhalten haben. Im Gegensatz zu der sonst in der Projektgruppe eingesetzen Sprache Java sind diese in einer Syntax zu implementieren, die der von C recht nahe kommt.

Daher ist es kein großer Aufwand, einzelne Funktionen nach Java zu portieren, um sie anschließend analog zum sonst gewählten Vorgehen mit JUnit[JUN] auf ihre Korrektheit zu überprüfen.

Bisher wurden allerdings keine Funktionen des Uppaal-Modells getestet, da sie oft sehr einfach sind und nur für die Programmierung von Schleifen genutzt wurden.

Vorlage für Testfälle

Für die Protokollierung eines absolvierten Tests verwenden wir eine Vorlage, die unten aufgeführt ist. Wir erklären unter jeder Überschrift, welche Angaben die Abschnitte bei einem durchgeführten Test enthalten.

Name des Testmodells: Zu jedem Teilaspekt gibt es ein Uppaal-Testmodell, das sich im `CVS` im Modul `Uppaal` unter `Modell/Tests` findet.

Motivation und Ziel des Tests: Hier findet man eine Erklärung, wodurch dieser Teilaspekt motiviert ist, d. h., an welcher Stelle im gewöhnlichen Modellablauf die hier zu testende Situation eintritt und, falls das einer Erläuterung bedarf, warum deren Korrektheit überprüft werden muss.

Des Weiteren wird angegeben, welche Anforderungen an das Testmodell gestellt werden. I. d. R. ergibt sich aus jeder Anforderung ein Testfall, der ihre Erfüllung verifizieren soll.

Annahmen: Hier werden Annahmen aufgeführt, insbesondere solche, die gewisse Eingaben ausschließen. Im UPPAAL-Werkzeug ist es oft nicht oder nur mit nicht vertretbarem Aufwand möglich, gewisse unerwünschte Eingaben auszuschließen. Es können aber sinnvolle Vorbedingungen gefordert werden, die insofern gerechtfertigt sind, als dass UPPAAL-Schienensysteme i. A. durch das Visualisierungstool (s. Kapitel 5) erstellt werden und durch dieses garantiert wird, dass nicht gegen die Annahmen verstoßen wird.

Vorgehensweise: In diesem Teil wird eine allgemeine Vorgehensweise angegeben, die sich auf *alle* Testfälle eines Teilaspektes bezieht. Zum Beispiel könnte ein Testsystem beschrieben werden, welches für alle Testfälle gleich oder zumindest ähnlich ist.

Testfall: Für jeden Testfall findet sich hier eine Kurzbeschreibung der zu verifizierenden Anforderung.

Testfallspezifische Vorgehensweise: Hier werden Besonderheiten bei der Durchführung des Testfalls angegeben, z. B. welche Vorlage(n) genutzt werden.

Spezifische Annahmen: Gibt eine Auflistung testfallspezifischer Annahmen.

Äquivalenzklassen regulärer Eingaben: Aufführung von Klassen von regulären Eingaben, die zu einem gleichwertigen Ergebnis führen (sollten).

Äquivalenzklassen fehlerhafter Eingaben: Aufführung von Klassen von fehlerhaften Eingaben, die nicht durch Annahmen ausgeschlossen wurden. Für fehlerhafte Eingaben darf die Berechnung des Modells nicht wie im regulären Fall fortgeführt werden und zu einem Ergebnis kommen, sondern das Modell muss in irgendeiner Weise auf sie reagieren, z. B. indem ein Ablauf in einen Deadlockzustand gezwungen wird und die Berechnung dieses Ablaufes damit abbricht.

Fehlerhafte Eingaben sollten an kritischen Stellen deshalb berücksichtigt werden, um Fehler, die evtl. bei der Arbeit mit dem Modell trotz Testens auftreten, abfangen zu können und damit für eine höhere Zuverlässigkeit des Modells zu sorgen.

Angenommene Werte: Es werden willkürlich konkrete Werte angenommen, sodass für jede Äquivalenzklasse eine repräsentative Wertemenge als Eingabe getestet wird.

Abbildungen und Quellcode: Falls notwendig, werden Abbildungen oder Quellcode an dieser Stelle eingefügt. Dieser Abschnitt kann entfallen.

Modelcheckinganfragen und erwartete Ergebnisse: Normalerweise wird die Korrektheit durch Modelcheckinganfragen überprüft. Die Anfragen, eine natürlichsprachliche Formulierung und Erklärung sowie die erwarteten Ergebnisse auf diese Anfragen finden sich hier.

Modelcheckinganfragen, die auf Eigenschaften einzelner Prozesse zugreifen, werden in Uppaal immer in der Form *prozessname.eigenschaft* gestellt. Wir verwenden bei der Systemdeklaration und für Variablen immer intuitive und selbsterklärende Namen, die bei Prozessen Abkürzungen der Vorlagennamen sind (z. B. Bhf für einen Prozess der Vorlage Bahnhof), sodass auch ohne explizite Erklärung bei Kenntnis des Testsystems klar ist, welche Eigenschaften welches Prozesses überprüft wird.

Tatsächliche Ergebnisse des Tests: Eine Auflistung der Ergebnisse des Testlaufs bzw. der Modelcheckinganfragen wird hier angegeben.

Bewertung der Ergebnisse: Die obigen Ergebnisse werden, falls notwendig, in diesem Abschnitt bewertet. Das ist insbesondere dann interessant, wenn sich die tatsächlichen Ergebnisse von den erwarteten unterscheiden.

4.8.3 Durchgeführte Tests

Dieses Kapitel umfasst eine Auswahl der Testfälle, die praktisch durchgeführt wurden. Bei der Auswahl wurde darauf geachtet, dass die Testfälle die Korrektheit nicht trivialer Kernfunktionalitäten überprüfen. Weitere identifizierte Testfälle finden sich in Kapitel 4.8.4.

Kategorie Auftragszuordnung - Teilaspekt: Anmelden von Palettenaufträgen in die Warteschlange eines Bahnhofs

Name des Testmodells: einschlangen.xml

Motivation und Ziel des Tests: Nachdem die Zuordnung der Palettenaufträge zu den Bahnhöfen abgeschlossen ist, können sich die Palettenaufträge in jeder möglichen Reihenfolge für eine Aufnahme in die *Palettenauftragsqueue* des Bahnhofs bewerben, dem sie zugeordnet wurden. Diese Queue gibt eine Reihenfolge vor, welcher Palettenauftrag als nächstes einem freien Kommissionierplatz zugeordnet wird.

Es ist zu überprüfen, dass

- ein Palettenauftrag korrekt in die Queue aufgenommen werden kann, solange sich weniger als für einen Parameter k-viele Palettenaufträge bereits in der Queue befinden.

Weitere Anforderungen finden sich unter 4.8.4, Kategorie „Auftragszuordnung“, Teilaspekt „Anmelden von Palettenaufträgen in die Warteschlange eines Bahnhofs“ .

Annahmen:

- Alle im Testmodell verwendeten Palettenaufträge und Bahnhöfe haben eine eindeutige, natürliche Zahl als Identifikationsnummer.

Vorgehensweise: Es wird ein spezielles UPPAAL-System konstruiert, auf dem die notwendigen Eigenschaften per Modelchecking verifiziert werden.

Testfall : Ein Palettenauftrag kann in die Queue des Bahnhofes aufgenommen werden, solange sich weniger als k Palettenaufträge in der Queue befinden.

Testfallspezifische Vorgehensweise: Im Testmodell befinden sich drei Palettenaufträge mit Identifikationsnummern (sog. `palettenAuftragIDs`) 0, 1 und 2 sowie ein Bahnhof (`BahnhofID` 1). Eine Initialisierungsvorlage sorgt dafür, dass die Palettenaufträge zu diesem Bahnhof für die Kommissionierung zugeordnet werden und sich daher anschließend im Zustand `Zugeordnet` befinden. Zusätzlich wird der Bahnhof in den Zustand `Frei` gesetzt.

Bei der Vorlage `PalettenAuftrag` wird eine Modifizierung der Auszeichnung der Zustände `Zugeordnet` als committed und `BedarfAnSammlerMelden` als gewöhnlich vorgenommmen, damit erzwungen wird, dass das System nach dem Aufnehmen von Palettenaufträgen in die Queue in einen Deadlock läuft und nicht wie im tatsächlichen Modell eine Bedarfsmeldung stattfindet. Diese Modifikation verändert das Verhalten des Testmodells im Vergleich zum Originalmodell ansonsten nicht.

Spezifische Annahmen:

- Der Parameter, der die Größe der Queue für Palettenaufträge im Bahnhof bestimmt wird als $k = 2$ angenommen.

Äquivalenzklassen regulärer Eingaben:

A) In der Queue sind noch Plätze frei und es kann noch ein Palettenauftrag aufgenommen werden.

B) Die Queue ist bereits gefüllt mit Palettenaufträgen, kein neuer Auftrag kann mehr aufgenommen werden.

Äquivalenzklassen fehlerhafter Eingaben: entfällt.

Angenommene Werte: Die Queue sei zu Beginn leer.

Modelcheckinganfragen und erwartete Ergebnisse: Wie angedeutet, ist das Testmodell so beschaffen, dass es in einen Deadlock läuft, sobald die Queue die maximale Anzahl an Palettenaufträgen aufgenommen hat. Allgemein ist ein Deadlock immer äquivalent dazu, dass die Berechnung eines Ablaufs beendet ist.

Wenn sich das System noch in der Berechnung findet, müssen daher noch Palettenaufträge aufgenommen werden können. Diese Eigenschaft wird durch die Anfrage

```
A[] (not deadlock --> Bhf.freiePalettenAuftragsQueuePlaetze > 0)
```

überprüft.

Die Palettenauftragsqueue ist ein Array, das an den belegten Positionen die Identifikationsnummer eines Palettenauftrages hält.

Sollte ein Deadlock auftreten, muss die Palettenauftragsqueue eine der möglichen Zweierkombinationen $k \in \{(i,j)|i,j = 1,\ 2,\ 3\ und\ i \neq j\}$ beherbergen. Der dritte Palettenauftrag muss dann jeweils noch im Zustand `Zugeordnet` sein, welches bedeutet, dass er, sobald es wieder einen freien Platz gibt, noch in die Queue aufgenommen werden kann. Beide Eigenschaften drückt die Anfrage[1]

```
A[] (deadlock -->
     (Bhf.pAQ[0] = 0 && Bhf.pAQ[1] = 1 && pa2.Zugeordnet) ||
     (Bhf.pAQ[0] = 1 && Bhf.pAQ[1] = 0 && pa2.Zugeordnet) ||
     (Bhf.pAQ[0] = 0 && Bhf.pAQ[1] = 2 && pa1.Zugeordnet) ||
     (Bhf.pAQ[0] = 2 && Bhf.pAQ[1] = 0 && pa1.Zugeordnet) ||
     (Bhf.pAQ[0] = 1 && Bhf.pAQ[1] = 2 && pa0.Zugeordnet) ||
     (Bhf.pAQ[0] = 2 && Bhf.pAQ[1] = 1 && pa0.Zugeordnet)
     )
```

aus.

Beide Anfragen zusammengenommen decken also die beiden Äquivalenzklassen ab. Sie sollen deshalb *erfüllt* sein.

Tatsächliche Ergebnisse des Tests: Alle Anfragen sind *erfüllt*.

Bewertung der Ergebnisse: Entfällt.

Kategorie Belade- und Entladevorgänge - Teilaspekt: Kommissionierung am Bahnhof

Name des Testmodells: kommissionierung.xml

[1] Wir kürzen aus Gründen der Textsetzung in der Anfrage `palettenAuftragQueue` mit `pAQ` ab.

Motivation und Ziel des Tests: Im Modell geschieht der Vorgang des Kommissionierens in einem Bahnhof nach folgendem Muster. Ein in den Bahnhof eingefahrenes Fahrzeug lädt nach Vorgabe durch den EHB-Auftrag, den es erfüllt, einen Teil seiner Ladung ab. Hierbei wird dem Bahnhof auch die Information, welcher der im Bahnhof zu kommissionierenden Palettenaufträge wie viel von der Ladung erhalten soll, übergeben. Der Bahnhof kümmert sich dann um die Zuteilung der Waren an die Palettenaufträge.

Es ist zu gewährleisten, dass

- die im EHB-Auftrag verzeichneten Palettenaufträge genau ihre zugeteilten Artikelmengen bekommen.

Weitere Testfälle finden sich unter Kapitel 4.8.4, Kategorie „Belade- und Entladevorgänge“, Teilaspekt „Kommissionierung am Bahnhof“.

Annahmen:

- Neben der Artikelart 0 gibt es eine Artikelart 1 im System.
- Es wird davon ausgegangen, dass die Palettenaufträge aus mindestens einer Position bestehen.
- Jede Position eines Palettenauftrags besteht aus einer gültigen Artikelnummer und einer Artikelmenge, die anfangs echt größer null ist.
- Die Anzahl der zu liefernden Waren für jeden Palettenauftrag ist nicht negativ.

Vorgehensweise: Es werden für die Äquivalenzklassen regulärer Eingaben und die fehlerhafter Eingaben jeweils spezielle Ausgangssituationen auf einem speziellen Uppaalsystem konstruiert. Unter „testfallspezifische Vorgehensweise“ wird der Initialzustand des Testsystems detailliert erläutert.

Anschließend werden geeignete Modelcheckinganfragen an das System gestellt.

Testfall: Es bekommen nur die vorgesehenen Palettenaufträge die für sie bestimmte Artikelmenge.

Testfallspezifische Vorgehensweise: Es wird ein Testmodell verwendet, das von folgender Anfangssituation ausgeht. Ein Fahrzeug (Vorlage `Fahrzeug`) mit einer Palette von 50 Artikeln ist in einen Bahnhof eingefahren und schon im Zustand `festgestellt`. Das Fahrzeug bearbeitet einen EHB-Auftrag, den wir später abhängig von unseren Äquivalenzklassen festlegen. Es gibt weiterhin fünf Palettenaufträge (Vorlage `PalettenAuftrag`), die sich im Zustand `Kommissionierung` befinden und

einen, der im Zustand `BedarfeMitgeteilt` ist. Da die Auftragszuordnung zu Beginn geschieht, ist die Annahme, dass sich Palettenaufträge „mindestens“ im Zustand `BedarfeMitgeteilt` befinden, gerechtfertigt.

Um die dargestellte Situation leicht herstellen zu können, werden die Originalvorlagen des Uppaalsystems um Transitionen mit Synchronisationsanweisungen ergänzt. Es sind zwei Testläufe notwendig, einer, der alle Äquivalenzklassen regulären Eingaben überprüft und einer, der die Äquivalenzklassen fehlerhafter Eingaben überprüft. Es existieren zwei Initialisierungsvorlagen, für jeden Testlauf eine, die die passenden Kanäle nutzen und damit die Vorlagen in die Zustände versetzen, die für die Überprüfung der Äquivalenzklassen gefordert werden.

Um die Modelcheckinganfragen einfacher gestalten zu können, wird weiterhin eine boolesche Variable `entladeSequenzBeendet` eingeführt, die initial den Wert `false` hat und, sobald der Bahnhof nach dem Zuteilen der vom Fahrzeug enthaltenen Ladung wieder in seinen Zustand `Frei` zurückkehrt, `true` gesetzt wird.

Wir verwenden bei den Äquivalenzklassen die Notation $\mathrm{P}A_i^Z(j) \rightarrow k$, um festzusetzen, dass dem Palettenauftrag i von der Artikelart j eine Artikelmenge von k Artikeln bei der Belieferung zugeteilt werden soll und analog die Notation $\mathrm{P}A_i^B(j) \rightarrow k$, um auszudrücken, dass der Palettenauftrag i einen Bedarf von k Artikeln von der Artikelart j hat.

Spezifische Annahmen: Keine.

Äquivalenzklassen regulärer Eingaben:

A) Ein Palettenauftrag hat Bedarf an der gelieferten Artikelart, für ihn ist aber keine Ladung vorgesehen.

B1) Ein Palettenauftrag hat Bedarf an der gelieferten Artikelart, für ihn ist (ein Teil) der Ladung des EHBs vorgesehen.

B2) Sei der Spezialfall von B1, dass der Bedarf des Palettenauftrages komplett befriedigt wird.

C) Ein Palettenauftrag hat keinen Bedarf an der gelieferten Artikelart und für ihn sind keine Artikel vorgesehen.

Äquivalenzklassen fehlerhafter Eingaben: Es gibt drei Fälle, in denen das System zu keinem gültigen Ergebnis kommen darf.

F1) Ein Palettenauftrag hat Bedarf an der gelieferten Artikelart, für ihn werden jedoch mehr Artikel geliefert, als er (noch) daran Bedarf hat.

F2) Ein Palettenauftrag bekommt Artikel von einer Artikelart, obwohl er keinen Bedarf (mehr) von dieser Artikelart hat. Dieser Fall ist ein Spezialfall des obigen.

F3) Für ein Palettenauftrag werden Artikel geliefert, obwohl sich dieser noch nicht in der Kommissionierung befindet.

Die ersten beiden Fälle könnten eintreten, falls durch einen Sammler fehlerhafte EHB-Aufträge generiert werden.

Der dritte Fall ist ein Fall, der auch in regulär funktionierenden Abläufen vorkommen kann. Für einen beliebig einstellbaren, aber festen Parameter $k > 0$ kann ein Sammler kann bei der Generierung eines EHB-Auftrages bereits für k-viele Palettenaufträge, die zwar einem Bahnhof zugeordnet sind, aber sich noch nicht in der Kommissionierung befinden, Artikel zur Auslieferung vorsehen. Damit sollen Abläufe möglich werden, bei denen für eine Palette, die erst nach dem Zuendekommissionieren einer anderen Palette in einen Bahnhof gelegt wird, schon vorher ein EHB-Fahrzeug ausgesendet werden darf, um potentiell schnellere Ergebnisse zu erzielen. Durch zu frühzeitiges Aussenden kann es jetzt aber passieren, dass ein Fahrzeug schon im Bahnhof eintrifft und Waren für einen Palettenauftrag hat, der noch nicht in der Kommissionierung ist, welches diesen Fehlerfall motiviert.

Angenommene Werte:
Wir nehmen an, dass das eingefahrene EHB-Fahrzeug die Artikelart 1 transportiert.

Für die regulären Fälle werden folgende Werte angenommen.

A) $\mathrm{PA}_0^B(1) \to 4$ und $\mathrm{PA}_0^Z(1) \to 0$
B1) $\mathrm{PA}_1^B(1) \to 4$ und $\mathrm{PA}_1^Z(1) \to 3$
B2) $\mathrm{PA}_2^B(1) \to 4$ und $\mathrm{PA}_2^Z(1) \to 4$
C) $\mathrm{PA}_3^B(1) \to 0$ und $\mathrm{PA}_3^Z(1) \to 0$

Für die Fehlerfälle werden folgende Werte festgesetzt.

F1) $\mathrm{PA}_4^B(1) \to 3$ und $\mathrm{PA}_4^Z(1) \to 4$
F2) $\mathrm{PA}_5^B(1) \to 0$ und $\mathrm{PA}_5^Z(1) \to 3$
F3) $\mathrm{PA}_6^B(1) \to 3$ und $\mathrm{PA}_6^Z(1) \to 2$ und PA_6 ist im Zustand `BedarfeMitgeteilt`

Modelcheckinganfragen und erwartete Ergebnisse: Sobald der Bahnhof nach dem Zuteilen der Ladung an die Palettenaufträge in den Zustand `Frei` gesprungen ist, hat die Variable `entladesequenzBeendet`den Wert *true*. Es wird daher überprüft, ob alle Palettenaufträge nun die für sie vorgesehene Menge erhalten haben. Die Bedarfe müssen nach dem Entladen daher folgende Werte aufweisen.

A) $\mathrm{PA}_0^{B'}(1) \to 4$
B1) $\mathrm{PA}_1^{B'}(1) \to 1$
B2) $\mathrm{PA}_2^{B'}(1) \to 0$
C) $\mathrm{PA}_3^{B'}(1) \to 0$

Überprüft wird dieses für *alle* regulären Äquivalenzklassen durch die gegebene Modelcheckinganfrage:

```
A[] ( (palettenAuftragsBedarfe[0][1] == 4 &&
       palettenAuftragsBedarfe[3][1] == 0 )
                        &&
       (bhf.entladesequenzBeendet  -->
        (palettenAuftragsBedarfe[1][1] == 1 &&
         palettenAuftragsBedarfe[2][1] == 0)
       )
    )
```

Diese soll *erfüllt* sein.

Für die Fehlerfälle F1 und F2 dürfen die betroffenen Palettenaufträge nicht den Zustand `Kommissionierung` verlassen. Da gilt $\mathrm{PA}_4^B(i) \to 0$ und $\mathrm{PA}_5^B(i) \to 0$ für $i \neq 1$, genügt es sicherzustellen, dass die beiden Palettenaufträge trotz Belieferung mit der Artikelart 1 niemals in den Zustand `FertigKommissioniertUndBahnhof-Raeumen` gelangen. Dann ist garantiert, dass dieses Modell zu keinem Ergebnis kommt.

Damit ist als Modelcheckinganfrage

```
A[] (not pa4.FertigKommissioniertUndBahnhofRaeumen &&
     not pa5.FertigKommissioniertUndBahnhofRaeumen)
```

gerechtfertigt, die *erfüllt* sein soll.

Im Fehlerfall F3 muss der betroffene Palettenauftrag nach der Belieferung in den als committed ausgezeichneten Zustand `Abbruch` gelangen, welcher sicherstellt, dass die Auswertung des Modells unmöglich wird, da er einen Deadlock forciert.

Damit ist die Anfrage

```
A[] (bhf.entladesequenzBeendet  -->
      pa6.Abbruch)
```

ausreichend.

Tatsächliche Ergebnisse des Tests: Alle Anfragen sind *erfüllt*.

Bewertung der Ergebnisse: Alle gewünschten Eigenschaften wurden verifiziert. Für die Fehlerfälle F1 und F2 bleibt zu bemerken, dass das Modell zwar zu keinem falschen Ergebnis kommt, jedoch unnötig lang, je nach konkretem Aufbau eines Schienensystems, evtl. auch unendlich lang, berechnet werden könnte. Daher sollte unbedingt sichergestellt werden, dass nur korrekte EHB-Aufträge generiert werden. Zusätzlich könnte man einen weiteren Zustand (ähnlich wie bei F3) einfügen, der einen Deadlock des Systems bei inkorrekten EHB-Aufträgen verursacht.

Kategorie Belade- und Entladevorgänge - Teilaspekt: Wiedereinlagern von Anbruchpaletten

Name des Testmodells: wiedereinlagern.xml

Motivation und Ziel des Tests: Nachdem ein Fahrzeug alle seine Ziele angefahren hat, kehrt es zum EBZKasten zurück, um dort entweder seine Leerpalette abzugeben oder um seine Anbruchpalette zu überstellen, die anschließend wieder in das HRL eingelagert werden muss. Aus lagerverwaltungstechnischen Gründen kann davon ausgegangen werden, dass für jede Artikelart maximal k Anbruchpaletten zu einer Zeit wieder eingelagert werden können. Nach Abgabe der Palette fährt ein Fahrzeug in das Wartegleis ein.

Es muss sichergestellt werden, dass

- die Anbruchpalette korrekt ins HRL zurückgestellt wird und eine Leerpalette nicht weiter berücksichtigt wird.

Weitere zu überprüfende Eigenschaften sind dem Kapitel 4.8.4, Kategorie „Belade- und Entladevorgänge“, Teilaspekt „Wiedereinlagern von Anbruchpaletten“, zu entnehmen.

Annahmen:

- Die Anzahl zurückgegebener Artikel ist größer oder gleich null[2].
- Jede Artikelart hat als Identifikationsnummer eine eindeutige natürliche Zahl.
- Es werden keine Paletten mit ungültigen, d. h. im System nicht vorhandenen Artikelnummern in das HRL eingelagert.

Vorgehensweise: Auch hier wird ein Testsystem benutzt, auf dem die zu verifizierenden Eigenschaften per Modelchecking überprüft werden. Der genaue Aufbau des Testsystems ist der testfallspezifischen Vorgehensweise zu entnehmen.

[2]Dieses erfordert, dass besonders auf die Korrektheit generierter EHB-Aufträge und die Beladung entsendeter EHB-Fahrzeuge wertgelegt werden muss, sodass nicht irrtümlicherweise zu viel Ware abgeladen werden kann und die Anzahl beförderter Artikel negativ wird.

Testfall: Anbruchpaletten werden korrekt ins HRL zurückgestellt.

Testfallspezifische Vorgehensweise: Es wird ein spezielles Testmodell verwendet, das von folgender Anfangssituation ausgeht. Nachdem ein Fahrzeug alle seine Ziele angefahren hat, steht es schon bereit im EBZ-Kasten, um dort seine Palette zurückzugeben. Sollte die Palette noch Ladung tragen, dann wird sie vom EBZ-Kasten an den Sammler, der für die entsprechende Artikelart zuständig ist, weitergereicht. Dieser hält sie als Anbruchpalette vor. Ansonten wird sie verworfen.

Der beschriebene Vorgang modelliert die Anbruchpalettenrückgabe an das HRL bzw. die Leerpalettenrückgabe im tatsächlichen IKEA-Lager.

Soweit erforderlich, werden die beteiligten Vorlagen `Fahrzeug`, `Sammler` und `EBZ-Kasten` um Transitionsanweisungen ergänzt, damit die beschriebene Situation leichter hergestellt werden kann.

Des weiteren wird eine boolesche Variable `entladenBeendet` eingeführt, die initial den Wert `false` hat und, sobald eine Einlagerung stattgefunden hat, auf `true` gesetzt wird.

Wir verwenden bei den Äquivalenzklassen die Notation $\mathrm{A}P \rightarrow (i, j)$, um festzusetzen, dass die vom Fahrzeug abzuladende Anbruchpalette noch j Artikel der Artikelart i trägt.

Ferner benutzen wir die Notation $\mathrm{S}AM_i(j) \rightarrow l$, wobei $j = 0, \ldots, k-1$, um auszudrücken, dass die j. eingelagerte Palette des Sammlers, der für die Artikelart i zuständig ist, einen Restbestand von l Artikeln *vor* der Entladung des EHB-Fahrzeugs aufweist und analog $\mathrm{S}AM'_i(j) \rightarrow l$, um den Bestand *nach* der Entladung des EHB-Fahrzeugs wiederzugeben.

Es gilt $\mathrm{S}AM_i(j) \rightarrow 0$ genau dann, wenn das Einlagern einer Anbruchpalette mit Artikelart i an Position j noch möglich ist.

Spezifische Annahmen:

- Die Anzahl der vorgehaltenen Anbruchpaletten wählen wir mit $k = 2$.

Äquivalenzklassen regulärer Eingaben:

A) Das Fahrzeug bringt eine Leerpalette zurück und das Einlagern einer weiteren Anbruchpalette mit der gleichen Artikelart ist noch möglich.
B) Das Fahrzeug bringt eine Leerpalette zurück und das Einlagern einer weiteren Anbruchpalette mit der gleichen Artikelart ist nicht mehr möglich.
C) Das Fahrzeug bringt eine Palette mit, die noch Ladung trägt und das Einlagern einer weiteren
Anbruchpalette mit der gleichen Artikelart ist möglich.

Äquivalenzklassen fehlerhafter Eingaben:

F1) Das Fahrzeug bringt eine Palette mit, die noch Ladung trägt, das Einlagern einer weiteren Anbruchpalette mit der gleichen Artikelart ist jedoch nicht mehr möglich.

Angenommene Werte: Für die regulären Fälle nehmen wir folgende Werte an:

A) $AP \rightarrow (0,0)$, $SAM_0(0) \rightarrow 7$ und $SAM_0(1) \rightarrow 0$
B) $AP \rightarrow (0,0)$, $SAM_0(0) \rightarrow 7$ und $SAM_0(1) \rightarrow 2$
C) $AP \rightarrow (0,5)$, $SAM_0(0) \rightarrow 7$ und $SAM_0(1) \rightarrow 0$

Für den Fehlerfall setzen wir folgende Werte fest:

F1) C) $AP \rightarrow (0,5)$, $SAM_0(0) \rightarrow 7$ und $SAM_0(1) \rightarrow 8$

Modelcheckinganfragen und erwartete Ergebnisse: Wir erwarten in den regulären Fällen, dass vor dem Einlagern immer die angenommenen Werte gültig sind und nach dem Einlagern an dem freien HRL-Platz der Bestand der Anbruchpalette verzeichnet ist bzw. dass der Bestand unverändert ist, sofern das Fahrzeug eine Leerpalette trug. Weiterhin soll nach der Entladung die Palette, die das Fahrzeug trägt, leer sein.

A) $AP' \rightarrow (0,0)$, $SAM'_0(0) \rightarrow 7$ und $SAM'_0(1) \rightarrow 0$
B) $AP' \rightarrow (0,0)$, $SAM'_0(0) \rightarrow 7$ und $SAM'_0(1) \rightarrow 2$
C) $AP' \rightarrow (0,0)$, $SAM'_0(0) \rightarrow 7$ und $SAM'_0(1) \rightarrow 5$

Bei den Modelcheckinganfragen muss berücksichtigt werden, dass die Anbruchpaletten, die die Sammler vorhalten, zunächst durch die Initialisierungsvorlage gesetzt werden müssen. Daher werden die Anfragen, die sich auf Zeitpunkte beziehen, die vor dem Beenden der Entladesequenz liegen, durch eine Implikation eingeleitet, die garantiert, dass der Anbruchpalettenbestand bereits gesetzt wurde.

Für den Fall C) ergibt sich damit die Anfrage

```
A[] (
   (
    (Init.IntialisierungFertig && not Sammler0.entladenBeendet) -->
     (Sammler0.anbruchPaletten[0] == 7 &&
      Sammler0.anbruchPaletten[1] == 0 &&
      Fhz.aktArtikel.artikelArt   == 0 &&
```

```
      Fhz.aktArtikel.artikelMenge == 5 )
   )
    &&
  (  Sammler0.entladenBeendet -->
     (Sammler0.anbruchPaletten[0] == 7 &&
      Sammler0.anbruchPaletten[1] == 5 &&
      Fhz.aktArtikel.artikelArt   == 0 &&
      Fhz.aktArtikel.artikelMenge == 0 )
  )
   )
```

Für die Fälle A) und B) kann, da sich der Bestand, den der Sammler vorhält, nicht ändert, eine vereinfachte Anfrage gestellt werden.

Für die Äquivalenzklasse A) ist damit

```
A[] (
     Init.IntialisierungFertig -->
      (Sammler0.anbruchPaletten[0] == 7 &&
       Sammler0.anbruchPaletten[1] == 2 &&
       Fhz.aktArtikel.artikelArt   == 0 &&
       Fhz.aktArtikel.artikelMenge == 0 )
    )
```

und für die Äquivalenzklasse B)

```
A[] (
     Init.InitialisierungFertig -->
      (Sammler0.anbruchPaletten[0] == 7 &&
       Sammler0.anbruchPaletten[1] == 0 &&
       Fhz.aktArtikel.artikelArt   == 0 &&
       Fhz.aktArtikel.artikelMenge == 0 )
    )
```

hinreichend.

Alle bisher aufgeführten Anfragen sollen *erfüllt* sein.

Für den Fehlerfall erwarten wird, dass der Bestand der Anbruchpaletten unangetastet bleibt. Daher wird die Anfrage

```
Init.IntialisierungFertig -->
      (Sammler0.anbruchPaletten[0] == 7 &&
       Sammler0.anbruchPaletten[1] == 8)
```

gestellt. Sie soll ebenfalls *erfüllt* sein.

Außerdem muss in irgendeiner Weise verhindert werden, dass das Fahrzeug die Entladezone verlässt, solang keine der beiden Anbruchpaletten ausgelagert wurde.

Tatsächliche Ergebnisse des Tests: Die Anfragen für die Fälle A), B) und C) sind *erfüllt*.

Die Anfrage für F1) ist *nicht erfüllt*. Nach einer Auswertung ergibt sich, dass $\mathrm{A}P' \rightarrow (0,0)$, $\mathrm{S}AM'_0(0) \rightarrow 7$ und $\mathrm{S}AM'_0(1) \rightarrow 5$ gilt.

Bewertung der Ergebnisse: In den regulären Fällen arbeitet das Modell korrekt. Im Fehlerfall F1) zeigt sich jedoch, dass eine Variable, die den Bestand einer Anbruchpalette vorhält, überschrieben wurde. Als Konsequenz ergibt sich, dass die Anzahl der ausgesendeten Paletten, die später wieder eingelagert werden, in jedem Falle korrekt von den Sammlern, die die EHB-Aufträge generieren, berücksichtigt werden muss. Der Teilaspekt „Aussendung von EHB-Fahrzeugen mit EHB-Aufträgen“ der Kategorie „EHB-Auftragsgenerierung“ soll die Forderung garantieren.

4.8.4 Weitere identifizierte Testfälle

Kategorie Auftragszuordnung

Teilaspekt: Zuordnungskombinationen Ein Zuordner weist einen Palettenauftrag einem Bahnhof zu, an dem dieser zu kommissionieren ist. Palettenaufträge und Bahnhöfe werden dabei über Identifikationsnummern referenziert.

Es muss gewährleistet werden, dass

- jede Zuordnungskombination zwischen Palettenauftrag und Bahnhof möglich ist.

Teilaspekt: Anmelden von Palettenaufträgen in die Warteschlange eines Bahnhofs Es ist neben der unter Kapitel 4.8.3, Teilaspekt „Anmelden von Palettenaufträgen in die Warteschlange eines Bahnhofes“, aufgezeigten Bedingung zu überprüfen, dass

- alle einem Bahnhof zugeordneten Palettenaufträge in jeder möglichen Reihenfolge in die Queue aufgenommen werden können, die Queue also „fair“ ist,
- die Queue nach der Aufnahme dann sequentiell abgearbeitet wird und ein aus der Queue genommener Palettenauftrag in die Kommissionierung kommt und
- nach der Entnahme eines Palettenauftrages aus der Queue ein zugeordneter, aber noch nicht in die Queue gekommenen Palettenauftrag, nun aufgenommen werden kann.

Teilaspekt: Eingeschlangte Palettenaufträge melden Bedarfe an Sammler

Motivation und Ziel des Tests: Sobald ein Palettenauftrag in die Auftragsqueue seines Bahnhofes aufgenommen wurde, muss er alle seine Positionen an sogenannte Sammler (Vorlage `Sammler`) melden. Für jede Artikelart existiert ein spezieller Sammler, der alle gemeldete Positionen dieser Artikelart vorhält.

Deswegen muss sichergestellt werden, dass

- alle Palettenaufträge ihre Bedarfe den korrekten Sammlern vollständig mitteilen und jeder Sammler alle Bedarfe korrekt vorhält.

Teilaspekt: Beendigung des Zuordnungsvorgangs vor Beginn der Simulation

Um den Suchraum des Modells nicht unnötig aufzublähen, darf o. B. d. A. die Zuordnung von Palettenaufträgen zu Bahnhöfen vor dem Beginn der eigentlichen Simulation (Fahr- und Kommissionierungsvorgänge usw.) stattfinden. Daher wird verlangt, dass

- die Zuordnung in Nullzeit geschieht und erst nach deren Beendigung Fahrvorgänge etc. beginnen.

Kategorie EHB-Auftragsgenerierung

Teilaspekt: Aussendung von EHB-Fahrzeugen mit EHB-Aufträgen Nachdem alle Bedarfe von den Palettenaufträgen an die Sammler gemeldet wurden, können die Sammler zu beliebigen Zeitpunkten EHB-Aufträge generieren, um die Bahnhöfe bzw. die Palettenaufträge mit Kommissionierware beliefern zu lassen.

Aufgrund der Platzbeschränkung in einem HRL kann man davon ausgehen, dass für einen Artikel zu jedem Zeitpunkt die Anzahl der Anbruchpaletten im Lager maximal k ist. Daher muss bei der Generierung von EHB-Aufträgen die Anzahl von Anbruchpaletten, die später wieder eingelagert werden müssen, berücksichtigt werden.

Um eine korrekte Funktionalität zu gewährleisten, muss überprüft werden, dass

- angeforderte Ware (mit Bahnhofs- und Palettenauftragsidentifikationsnummer) vorgehalten wird und

- für alle angeforderten Artikel (irgendwann einmal) auch EHB-Aufträge generiert werden, wobei die Anzahl der Anbruchpaletten berücksichtigt wird, aber niemals unangefordert Ware ausgesendet wird.

Kategorie Fahrvorgänge

Teilaspekt: Einfahrt in Schienenstücke Wenn Fahrzeuge von einem Schienenstück in ein nachfolgendes Schienenstück einfahren, geschieht das nach folgendem Protokoll: Die Vorgängerschiene teilt der nachfolgenden Schiene die Identifikationsnummer (`FahrzeugID`) des einfahrenden Fahrzeuges mit. Die Nachfolgerschiene synchronisiert sich mittels dieser Nummer mit dem einfahrenden Fahrzeug und teilt die Nummer des passierten Übergabepunktes (die sog. `vonID` des Schienenstückes), die Länge des Schienenstückes sowie die auf diesem Schienenstück maximal erlaubte Geschwindigkeit (`Vmax`) mit. Schienenstücke aller Art halten vor, ob sie noch Fahrzeuge aufnehmen können.

Es muss für alle Arten von Schienenstücken sichergestellt werden, dass

- ein Schienenstück seinen Status (besetzt/noch k Plätze frei) korrekt protokolliert,
- das einfahrende Fahrzeug angesprochen wird und diesem die Eigenschaften (`vonID, laenge, Vmax`) dieses Schienenstücks übermittelt werden.

Teilaspekt: Verweilzeit auf Schienenstücken Fahrzeuge können potentiell unterschiedliche Maximalgeschwindigkeiten fahren. Genauso können Schienen für unterschiedliche Geschwindigkeiten vorgesehen sein.

Daher ist für Schienenstücke, die im Schienensystem ausschließlich die Überfahrt eines Fahrzeuges über einen Gleisabschnitt modellieren (Weichen, Schienen usw.), zu überprüfen, dass

- ein Fahrzeug eine gewisse Verweildauer verbringt, die sich über das Minimum der Maximalgeschwindigkeit des Fahrzeuges (v_{fhz}) und der Maximalgeschwindigkeit der Schiene (v_{elem}) und die Länge des Schienenstücks (l_{elem}) berechnet[3].

In Bahnhöfen, im EBZKasten u. ä. ergibt sich die Verweildauer aus anderen Faktoren wie Belade- und Entladezeiten und wird daher gesondert betrachtet, sodass sie hier nicht berücksichtigt werden muss.

Teilaspekt: Mehrfachbesetzung von Schienen Manche Schienenstücke dürfen zu einer Zeit nur von einem Fahrzeug befahren werden. Andere, wie z. B. Bahnhöfe, dürfen nur durch eine feste Anzahl von Fahrzeugen gleichzeitig besetzt werden. Im jetzigen Modell gibt es darüberhinaus keine Überholmöglichkeit für Fahrzeuge auf solchen mehrfach besetzten Stücken. Daher muss sichergestellt werden, dass

- einfache Schienenstücke, Weichen u. ä. nur durch ein Fahrzeug besetzt sein dürfen,

[3] Als Formel: $t = \frac{l_{\text{elem}}}{\min\{v_{\text{elem}}, v_{\text{fhz}}\}}$

- auf Schienenstücken, die eine Mehrfachbesetzung zulassen, für je zwei nacheinander eingefahrene Fahrzeuge gilt, dass diese in der gleichen Reihenfolge wieder ausfahren.

Teilaspekt: Routing auf verzweigenden Weichen Verzweigende Weichen (Vorlagen `ZweierWeicheTeilung` und `DreierWeicheTeilung`) haben zwei bzw. drei Ausfahrten zu nachfolgenden Schienenstücken. Zielen, die durch Fahrzeuge explizit angefahren werden können, ist eine Identifikationsnummer (sog. `BahnhofsID`) zugeordnet. Jedes Fahrzeug, das sich an einer teilenden Weiche befindet, übermittelt diese Identifikationsnummer des nächsten anzufahrenden Ziels an die Weiche. Anhand der Nummer findet ein Routing statt, d. h. die Weiche wählt eine bzw. mehrere durch eine Routingtabelle festgelegte Ausfahrt(en) für das Fahrzeug aus. Es ist zu beachten, dass

- Fahrzeuge verzweigende Weichen nur durch die durch das Routing festgelegten Ausgänge verlassen dürfen.

Teilaspekt: Ausfahrt aus Schienenstücken Nachdem Fahrzeuge aus Schienenstücken ausfahren dürfen, muss jeder Typ von Schienenstück

- an die (richtige) Nachfolgerschiene die Nummer des ausfahrenden Fahrzeugs mitteilen und
- für ein Fahrzeug erst eine Ausfahrt vorsehen, wenn die Nachfolgerschiene frei ist.

Kategorie Belade- und Entladevorgänge

Teilaspekt: Auslagern aus dem Hochregallager Im Uppaal-Modell wird der gesamte Prozess des Auslagerns einer Palette aus dem HRL und die Übergabe derer an ein EHB-Fahrzeug sowie das Überstellen des zugehörigen EHB-Auftrages an ein Fahrzeug wie folgt modelliert.

Sobald ein EHB-Auftrag von einem Sammler durch den EBZKasten entgegengenommen wird, fordert der EBZKasten ein Fahrzeug vom Wartegleis an. Danach muss der EBZKasten genau die vom Sammler übergebene Palette samt Belieferungsinformationen (Bahnhof, Anzahl der abzuliefernden Waren, Palettenaufträge, die die Waren bekommen sollen) an das Fahrzeug weitergeben.

Es muss gewährleistet werden, dass

- nach der Übergabe eines EHB-Auftrages an den EBZ-Kasten ein Fahrzeug vom Wartegleis angefordert wird und in den EBZ-Kasten einfährt,
- die Übergabe der Palette vom Sammler an das Fahrzeug über den EBZKasten fehlerfrei funktioniert,

- die zu beladenden Fahrzeuge eine angemessene Zeit im EBZKasten verbringen, um die Beladungsdauer zu simulieren und (erst) nach der Beladung Ausfahrt aus dem EBZKasten erhalten können.

Teilaspekt: Kommissionieren am Bahnhof Neben dem schon unter Kapitel 4.8.3, Teilaspekt „Kommissionierung am Bahnhof“, beschriebenen Testfall müssen zusätzlich die Anforderungen, dass

- für jeden abgeladenen Artikel eine konstante Pickzeit verstreicht, bevor das Fahrzeug Ausfahrt erhält (dieses modelliert die Dauer, die der Kommissionierer in der Realität für das Picken der Artikel benötigt),
- nach Ausfahrt des zuletzt (nicht unbedingt vollständig) entladenen Fahrzeuges, die bereits im Bahnhof auf die Entladung wartenden Fahrzeuge vorrücken, sodass das nächste Fahrzeug an den Stellplatz, an dem es abgeladen werden kann, gelangt

erfüllt sein.

Teilaspekt: Wiedereinlagern in das HRL Neben der unter Kapitel 4.8.3, Teilaspekt „Wiedereinlagern in das HRL“, beschriebenen Anforderung ist weiterhin zu überprüfen, dass

- das Fahrzeug anschließend eine angemessene Zeit einen Entladeplatz blockiert (entfällt bei Rückgabe einer Leerpalette)
- das Fahrzeug nach der Entladung in das Wartegleis einfährt.

Kategorie Auftragserfüllung

Teilaspekt: Erfüllung eines Palettenauftrages Ein Palettenauftrag besteht aus einer Menge von Positionen. Eine Position besteht aus einer Artikelart, die durch eine eindeutige Nummer gegeben ist, und aus einer Artikelmenge.

Es muss gewährleistet werden, dass

- ein Palettenauftrag nur genau dann erfüllt sein darf, wenn jede Position abgearbeitet wurde, d. h. zu jeder benötigten Artikelart die entsprechende Artikelmenge kommissioniert wurde und
- nach Erfüllung eines Palettenauftrages ein Kommissionierplatz im Bahnhof freigegeben wird.

Teilaspekt: Erfüllung eines LKW-Auftrages Ein LKW-Auftrag setzt sich aus Palettenaufträgen zusammen. Es muss sichergestellt werden, dass

- ein LKW-Auftrag nur genau dann fertig sein darf, wenn alle zugehörigen Palettenaufträge abgearbeitet sind.

Teilaspekt: Erfüllung einer Batch Eine Batch besteht aus LKW-Aufträgen. Es ist sicherzustellen, dass

- eine Batch nur genau dann fertig sein darf, wenn alle LKW-Aufträge abgearbeitet sind.

4.9 Grenzen des Modells

Zur Halbzeit der Projektgruppe zeichnete sich ab, dass an Berechnungen auf einem großen Modell (Ikea-Dortmund-Mengede-Modell) nicht zu denken ist. Stattdessen konzentrierten wir uns auf die Verifikation kleiner Modelle, speziell auf die Gewinnung optimaler Abläufe (geringster Zeitverbrauch).

Ausgehend von einem einfachen Grundmodell mit zwei Bahnhöfen, zwei Fahrzeugen, drei Palettenaufträgen und drei Artikelarten wurden die letzten drei Parameter einzeln variiert, um ihre Auswirkung auf die benötigte Rechenzeit für eine Suche nach dem schnellsten Ablauf darzustellen. Die Tests wurden auf einem Opteron-System (Prozessorgeschwindigkeit 2400 MHz) und 8 GB Arbeitsspeicher von Programm `verifyta` (Bestandteil von Uppaal 4.0.1) durchgeführt. Der nutzbare Arbeitsspeicher war aufgrund der Programmbeschaffenheit auf 2^{32} Bytes ($\approx$ 4GB) beschränkt. Mit Hilfe des Programms `time` wurde die Rechenzeit („user CPU time“) der Modelchecking-Anfrage `E <> unsereBatch.Fertig` bestimmt. Die im Modell vergangene Zeit (hier *Modellzeit* genannt) ergibt sich aus dem Wert der Variablen `systemZeit` am Ende des im Erfolgsfall erstellten Modellablaufs.

In der Testreihe mit verschieden vielen Fahrzeugen ist ein starker Anstieg der Berechnungszeit vom Modell mit einem bis zum Modell mit vier Fahrzeugen zu erkennen (vgl. Tabelle 4.1). Die Modellzeit sinkt entsprechend ab bis auf ein Intervall von $[50, 55]$, das aber schon mit drei Fahrzeugen erreicht wird. Dass trotz großem Rechenzeitunterschied zwischen drei und vier Fahrzeugen die Modellzeit sich nicht ändert, ist rein zufällig. Es kommen alle drei bzw. vier Fahrzeuge zum Einsatz. Ein fünftes Fahrzeug kommt jedoch schon nicht mehr zum Einsatz, welches auch den dann nur noch geringen Anstieg der Rechenzeit erklärt.

In der zweiten Testreihe (Tabelle 4.2) wurde die Abhängigkeit der Rechenzeit von der Anzahl der Palettenaufträge untersucht. Während das für das Modell mit ein oder zwei Aufträgen in unter einer bzw. zwei Sekunden eine schnellste Lösung gefunden ist, schnellt die Rechenzeit ab dort explosionsartig in die Höhe. Dies liegt an der enorm gestiegenen Anzahl möglicher Systemzustände, die untersucht werden

# Fhzg.	Rechenzeit	Modellzeit	Bemerkung
1	29 s	[170, 185]	
2	48 s	[100, 110]	Grundmodell
3	112 s	[50, 55]	
4	400 s	[50, 55]	
5	418 s	[50, 55]	
6	442 s	[50, 55]	
7	465 s	[50, 55]	

Tabelle 4.1: Tests mit verschiedener Anzahl der Fahrzeuge.

müssen. Es gibt zwei Möglichkeiten für die Zuordnung zu einem Bahnhof, die dreifache Anzahl Möglichkeiten für die Reihenfolge der Palettenaufträge beim Melden der Bedarfe, und daher für die Bedarfssammler auch mehr Möglichkeiten, wann sie einen EHB-Auftrag generieren. Mit vier Palettenaufträgen bricht der Verifikationsprozess die Berechnung nach 520 Sekunden ab, weil kein weiterer Speicher mehr verwaltet werden kann.

# Auftr.	Rechenzeit	Modellzeit	Bemerkung
1	0 s	[35, 40]	
2	2 s	[95, 100]	
3	48 s	[100, 110]	Grundmodell
4	(520 s)	-	Abbruch

Tabelle 4.2: Tests mit verschiedener Anzahl der Palettenaufträge.

Auch von der Anzahl der Artikelarten ist die benötigte Rechenzeit stark abhängig, wie das Ergebnis der dritte Testreihe zeigt (Tabelle 4.3). Mit Anstieg der Anzahl der Artikelarten steigt die Rechenzeit jeweils um Faktoren zwischen 3 und 16. Bei fünf verschiedenen Artikelarten liefert der Verifizierer aufgrund von Speichererschöpfung kein Ergebnis mehr. Es fällt auf, dass die Modellzeiten bei einer und zwei Artikelarten sowie bei drei und vier Artikelarten nahe beieinander liegen. Der Grund ist, dass besonders das Fahren von der Ent- und Beladezone zum Bahnhof Zeit verbraucht. Bei ein oder zwei Artikelarten bringt jedes der Fahrzeuge eine der beiden Artikelarten, während bei drei Artikelarten mindestens ein Fahrzeug seine Palette wieder an das Hochregallager abgeben muss, um mit der dritten Artikelart noch einmal den Bahnhof zu besuchen, an dem die beiden Paletten kommissioniert werden.

4.10 Bewertung und Ausblick

Die durchgeführten Korrektheitstests zeigen, dass das erstellte Modell wie gewünscht funktioniert.

# Art.	Rechenzeit	Modellzeit	Bemerkung
1	1 s	[30, 45]	
2	3 s	[40, 45]	
3	48 s	[100, 110]	Grundmodell
4	211 s	[110, 115]	
5	(532 s)	-	Abbruch

Tabelle 4.3: Tests mit verschiedener Anzahl der Artikelarten.

Die Uppaal-Teilgruppe verfolgte den Ansatz, Objekte des Dortmund-Mengede-Lagers in Automaten zu „gießen“. Zwar kann Uppaal mit einer recht großen Menge von Prozessen umgehen. Doch aufgrund der Dimension des geplanten Lagers und der Beschränkungen der Modellsprache von Uppaal sind derart viele Prozesse und Variablen nötig, so dass weder an eine Simulation geschweige denn an eine Verifikation eines solch großen Modells mit Uppaal zu denken ist. Es muss beispielsweise für jede Artikelart ein Prozess existieren, damit die Reihenfolge, in der Artikel zu Paletten gefahren werden, zufällig sein kann. Eine undokumentierte maximale Arraygröße würde jedoch schon das Einlesen der Routing- oder Bedarfstabelle verhindern.

Stattdessen wurden kleinere Modelle gebaut und ab Ende des ersten Halbjahres auch erfolgreich ausgeführt. Das Ziel, einen schnellsten Ablauf in so einem Modell mit gegebener Anzahl Palettenaufträgen, Fahrzeugen usw. zu finden, rückte vor allem durch zahlreiche kleine und größere Verbesserungen zu Beginn und während des zweiten Halbjahres wieder in den Vordergrund. Doch auch hier sind die Grenzen sehr eng (vgl. Abschnitt 4.9).

Vielleicht hätte eine noch stärkere Abstraktion bei der Modellierung zu besseren Ergebnissen führen können. Das Grundproblem der Zuordnung von Palettenaufträgen zu Bahnhöfen, des Zusammenstellens von EHB-Fahraufträgen und ihrer Reihenfolge bleibt jedoch schwierig.

5 UppaalVis

5.1 Motivation

In Uppaal werden Systeme als Netze von zeitgesteuerten Automaten realisiert. Diese Netze können mitunter sehr groß und unübersichtlich werden. Während der Simulation mit Uppaal hat man lediglich die Möglichkeit, die Zustandswechsel der einzelnen Automaten zu beobachten. Bei mehreren hundert dieser Automaten, wie sie in dem zu entwickelnden Modell vorkommen werden, ist es unmöglich, anhand der Simulation durch Uppaal zu erkennen, welche Schritte in dem Modell gerade ausgeführt werden. Da es sich hier um die Modellierung und Simulation eines *Schienensystems* handelt, ist eine Visualisierung sinnvoll, welche die Fahrzeugbewegungen auf den *Schienenelementen* grafisch darstellt. Dies erleichtert zum einen die Suche und das Beheben von Fehlern im Modell und ist zum anderen eine gute Hilfe bei der Präsentation der Ergebnisse. Somit wurde der Entschluss gefasst, ein entsprechendes Werkzeug für die Visualisierung des Modells sowie die Bewegungen der Fahrzeuge in diesem Modell anhand der *Trace*-Dateien zu entwickeln.

Ein weiteres Problem entsteht bei der Entwicklung des Modells. Die einzelnen Vorlagen für die Komponenten des Modells müssen zwar manuell implementiert werden, aber bei der Instanziierung der einzelnen Prozesse können Probleme auftreten. Das manuelle Erstellen der einzelnen Schienenelemente, der Fahrzeuge, der Aufträge sowie der *Routingtabelle* ist sehr zeitaufwändig und fehleranfällig. Hierbei ist ein Editor, mit dem man das Modell grafisch aufbauen kann, sehr hilfreich. Auf Basis des modellierten Schienensystems ist es dann möglich, die Parameter der zu instanziierenden Prozesse zum größten Teil automatisch zu berechnen.

In den folgenden Abschnitten werden Aufbau und Funktionsweise von *UppaalVis*, einem Werkzeug, das einen Editor und eine Möglichkeit zur Visualisierung der Simulation auf dem Modells bereitstellt, im Detail vorgestellt. Zusätzlich bietet UppaalVis die Möglichkeit Traces mit Hilfe des Kommondozeilenverifizierers von Uppaal zu erstellen.

5.2 Der Editor

Zunächst wird der Editor vorgestellt, mit dem man Schienenmodelle entwerfen kann, die im Uppaal-XML-Format gespeichert werden können. Diese Modelle können

dann entweder in UPPAAL selbst simuliert oder mit dem Kommandozeilenverifizierer ausgewertet werden. Der Verifizierer ist ebenfalls in UppaalVis eingebunden.

5.2.1 Bedienung

Der in Abbildung 5.1 dargestellte Editor ermöglicht es einfach und schnell auch komplexe Schienenmodelle zu erstellen. Hierzu platziert man die entsprechenden Elemente aus der Werkzeugleiste im Arbeitsbereich. Der Arbeitsbereich ist mit einem Raster versehen, in welches die einzelnen Komponenten eingesetzt werden können. Unterhalb der Werkzeugleiste kann man zusätzlich für die einzelnen Elemente Parameter wie Länge oder Geschwindigkeit angeben. Diese kann man sowohl für alle Elemente global als auch lokal für einzelne Komponenten festlegen. Die globalen Parameter gelten für alle Elemente, denen nicht explizit lokale Werte zugewiesen wurden. Lokale Parameter können wieder zurückgenommen werden, so dass für das entsprechende Element wieder die globalen Parameter gelten. Bereits platziere Elemente können mit einem rechten Mausklick wieder aus dem Arbeitsbereich entfernt werden.

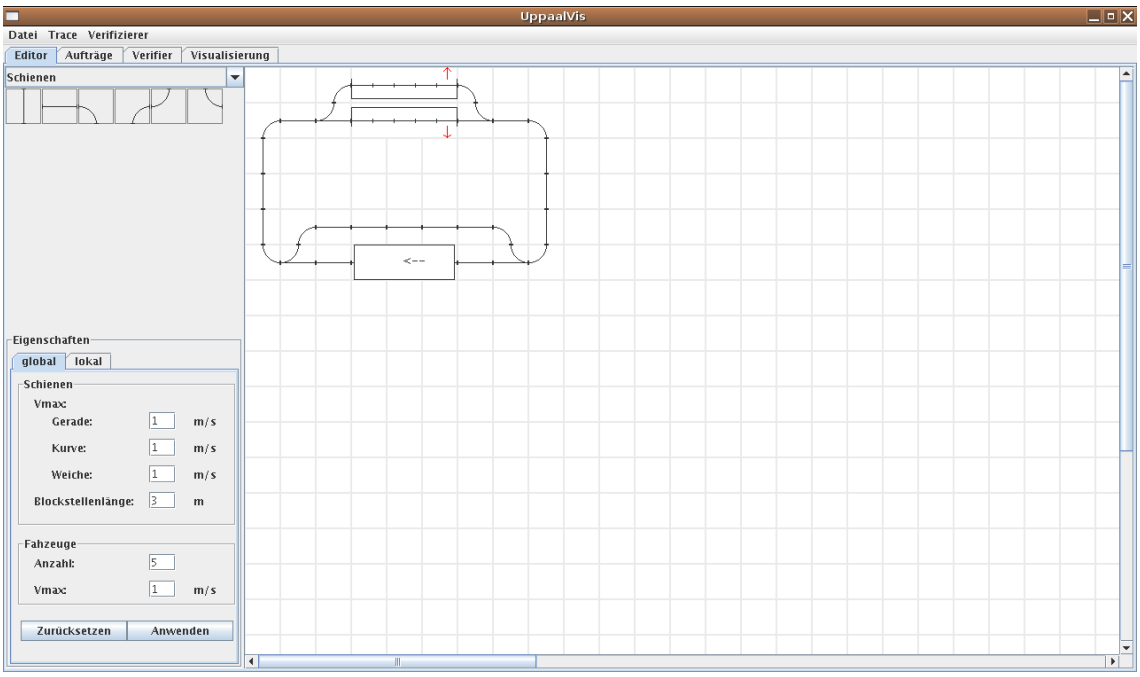


Abbildung 5.1: Die Editorumgebung.

Auf diese Weise kann ein komplexes Schienensystem modelliert werden. Bevor dieses Modell in eine UPPAAL-XML-Datei exportiert werden kann, müssen folgende Punkte beachtet werden:

- Das modellierte System muss genau eine Blackbox enthalten. Die Blackbox enthält die logischen UPPAAL-Komponenten `Quelle`, `Wartegleis` und `EBZ-Kasten`.
- Es handelt sich um ein Einbahnstraßensystem, d. h. alle Schienenelemente können nur in einer Richtung durchfahren werden.
- Das Schienenmodell muss unter Berücksichtigung der Fahrtrichtung in sich geschlossen sein.

Werden jedoch einer oder mehrere dieser Punkte nicht eingehalten, kann das Modell nicht gespeichert werden und man erhält eine entsprechende Fehlermeldung.

Zu einem vollständigen Modell gehört neben einem korrekten Schienensystem auch eine Batch, durch die die abzuarbeitenden Aufträge definiert werden. Diese kann man mit Hilfe des in Kapitel 3.3.2 vorgestellten Batchgenerators erzeugt werden, der ebenfalls in UppaalVis eingebunden ist. Wurde keine Batch erzeugt, so gilt das Modell als unvollständig und kann ebenfalls nicht gespeichert werden.

5.2.2 Modellerzeugung

Hat man ein Modell im Editor fertiggestellt und eine Batch generiert, so kann man es als UPPAAL-XML-Datei speichern.

Diese XML-Datei hat folgende Struktur:

```
<nta>
<declaration>
... //Definition globaler Variablen, Strukturen und Funktionen.
</declaration>
<template>
... //Syntaktische Beschreibung einer Vorlage des Systems.
</template>
<template>
...
</template>
.
.
.
<system>
... //Definition der Prozesse (Namenszuweisung und Parameterübergabe)
system
... //Auflistung der Prozesse, die das System beschreiben.
</system>
</nta>
```

Das modellierte System wird in einem `Vector` gespeichert. Dieser Vector enthält wiederum Vektoren. Diese dynamische, zweidimensionale Datenstruktur wird beim

Bearbeiten des Modells erzeugt und speichert Elemente des Typs `ModelComponent`. Die Instanzen dieser Klasse enthalten die notwendigen Parameter zur Beschreibung der Schienenelemente. Zur Identifizierung der Form der Schiene dient die Enumeration `Comp`, die für jedes Schienenelement eine Konstante definiert. Der Konstanten der `Comp`-Enumeration sind in Tabelle 5.1 aufgelistet.

ID	Name	Symbol
1	RAIL_TOP_BOTTOM	
2	RAIL_LEFT_RIGHT	
3	RAIL_BOTTOM_RIGHT	
4	RAIL_BOTTOM_LEFT	
5	RAIL_TOP_RIGHT	
6	RAIL_TOP_LEFT	
7	SWITCH_BOTTOM_RIGHT	
8	SWITCH_BOTTOM_LEFT	
9	SWITCH_TOP_RIGHT	
10	SWITCH_TOP_LEFT	
11	SWITCH_LEFT_TOP	
12	SWITCH_LEFT_BOTTOM	

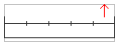
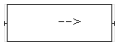
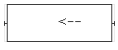
ID	Name	Symbol
13	SWITCH_RIGHT_TOP	
14	SWITCH_RIGHT_BOTTOM	
15	SWITCH_LEFT_TOP_BOTTOM	
16	SWITCH_RIGHT_TOP_BOTTOM	
17	SWITCH_TOP_LEFT_RIGHT	
18	SWITCH_BOTTOM_LEFT_RIGHT	
19	STATION_TOP	
20	STATION_BOTTOM	
21	STATION_RIGHT_TOP	
22	STATION_RIGHT_BOTTOM	
23	BLACKBOX	
24	BLACKBOX_LEFT	

Tabelle 5.1: Verwendete Schienenelemente.

Global definierte Variablen werden in einem `Properties`-Objekt gespeichert. Jedes der Schienenelemente enthält eine boolsche Variable `localDefined`, die angibt, ob für dieses Objekt die globalen oder die lokalen Eigenschaften gelten sollen. Die `Properties`, der zweidimensionale `Vector`, der das Schienenmodell repräsentiert sowie eine Referenz auf einen `OrderController`, der Informationen über die Aufträge enthält, werden als Parameter an den Konstruktor eines `ModelBuilder` übergeben. Dieser implementiert die Methoden für den Aufbau der UPPAAL-XML-Datei.

Der `ModelBuilder` hat nun alle benötigten Informationen, um das Modell zu erstellen. Hierzu wird zunächst eine Tiefensuche durchgeführt, mit deren Hilfe die Übergabepunkte der Komponenten verteilt werden. Das Schienenmodell wird dazu als ein der Fahrtrichtung entsprechend gerichteter Graph interpretiert. Die Tiefensuche startet an der einzigen Blackbox im Modell. Gibt es keine oder mehrere Blackboxen, so bricht der Algorithmus ab und liefert eine Fehlermeldung. Dieser Algorithmus entscheidet anhand der aktuellen `Comp`-Konstante und der Fahrtrichtung, an welcher Position im zweidimensionalen Vektorfeld die nächste Komponente zu finden ist. Erreicht der Algorithmus eine Weiche, die durch ihre Form und der Fahrtrichtung eindeutig als verzweigend identifiziert wurde, so wird er für jede aus-

gehende Richtung rekursiv aufgerufen. Trifft er hingegen auf eine zusammenführende Weiche, die bereits besucht wurde, bricht der Aufruf an dieser Stelle ab. Gleiches gilt für das erneute Erreichen der Blackbox. Wird während der Suche eine Komponente gefunden, deren Fahrtrichtung nicht mit der erwarteten übereinstimmt, bricht der Algorithmus ab und meldet einen Fehler. Im Fall einer fehlenden Komponente ist das Modell nicht geschlossen, worauf ebenfalls mit einer Fehlermeldung und dem Abbruch des Algorithmus reagiert wird. Auf diese Weise wird sichergestellt, dass alle Elemente innerhalb des geschlossenen Modells abgearbeitet werden.

Nach der Berechnung der Übergabepunkte für die einzelnen Komponenten, muss die Routingtabelle erstellt werden. Diese Tabelle enthält für jede verzweigende Weiche und für jedes Ziel – Bahnhöfe oder EBZKasten – die zu fahrende Richtung. Um diese Tabelle zu erstellen wird von jeder verzweigenden Weiche eine Tiefensuche gestartet, um die Ziele zu finden. Die Richtung, an der der Algorithmus bei der Weiche weiterläuft, wird gespeichert und ist beim Finden eines Ziels der Eintrag der Routingtabelle. Als Annahme gilt hierbei, dass für einen kürzesten Weg von einer Weiche zu einem Ziel kein weiteres Ziel durchlaufen werden. Daher bricht die Suche in einem Zweig ab, in dem ein Ziel gefunden wurde. Die Länge des Weges wird mit dem aktuellen Eintrag verglichen und ersetzt, falls der Weg kürzer ist, als der zuvor Gefundene.

Im Anschluss an die Berechung der Routingtabelle und an die Verteilung der Übergabepunkte, kann das Modell in eine Uppaal-XML-Datei geschrieben werden. Dazu werden die errechneten Informationen zu einem `String` zusammengefügt und mit Hilfe eines `FileWriters` in eine ausgewählte Datei geschrieben. Die Information über die Automaten werden dabei statisch übernommen, lediglich die globalen Variablen, die Instanziierung der Prozesse sowie der Aufbau der Systemdefinition wird anhand des modellierten Schienensystems ergänzt. Diese Datei kann nun in Uppaal geöffnet oder mit dem *ModelChecker* verifiziert werden.

Neben der XML-Datei wird eine zusätzliche Datei erzeugt. Diese Datei hat den gleichen Namen wie die XML-Datei, endet aber auf „.pos“ anstatt auf „.xml“. In dieser Positionsdatei werden die X- und Y-Positionen sowie die als Integer kodierten Formen der Schienenelemente im Raster gespeichert. Die erste Zeile der Positionsdatei enthält die Anzahl der globalen Variablen des zugehörigen Modells. Diese zusätzliche Datei ist nötig, da das eigentliche XML-Modell keine Informationen zur Positionierung der einzelnen Schienenelemente speichert. Auf diese Weise wird sichergestellt, dass ein bereits modelliertes System wieder in die Editorumgebung geladen werden kann. Ebenfalls notwendig ist diese Datei, um ein Modell in der Visualisierungsumgebung grafisch darzustellen. Im Abschnitt 5.4 wird diese im Detail vorgestellt.

5.3 Der Verifizierer

Nach dem endgültigen Entwurf aller zeitgesteuerten Automaten, die für den Aufbau eines der Aufgabenstellung entsprechenden Modells benötigt werden, wurde Uppaal lediglich dazu genutzt, die mit UppaalVis erstellten Modelle zu laden und mit Hilfe des ModelCheckers sowie einer geeigneten Anfrage sich einen Trace berechnen zu lassen. Dieser kann in eine Datei gespeichert werden und Uppaal wieder beendet werden, da die Simulation der Fahrzeugbewegungen entlang des modellierten Schienensystems mit UppaalVis dargestellt werden soll. Um sich diesen Umweg zu ersparen, kann aus UppaalVis heraus der Kommandozeilenverifizierer von Uppaal gestartet werden, der einen entsprechenden Trace erzeugt und in eine Datei speichert.

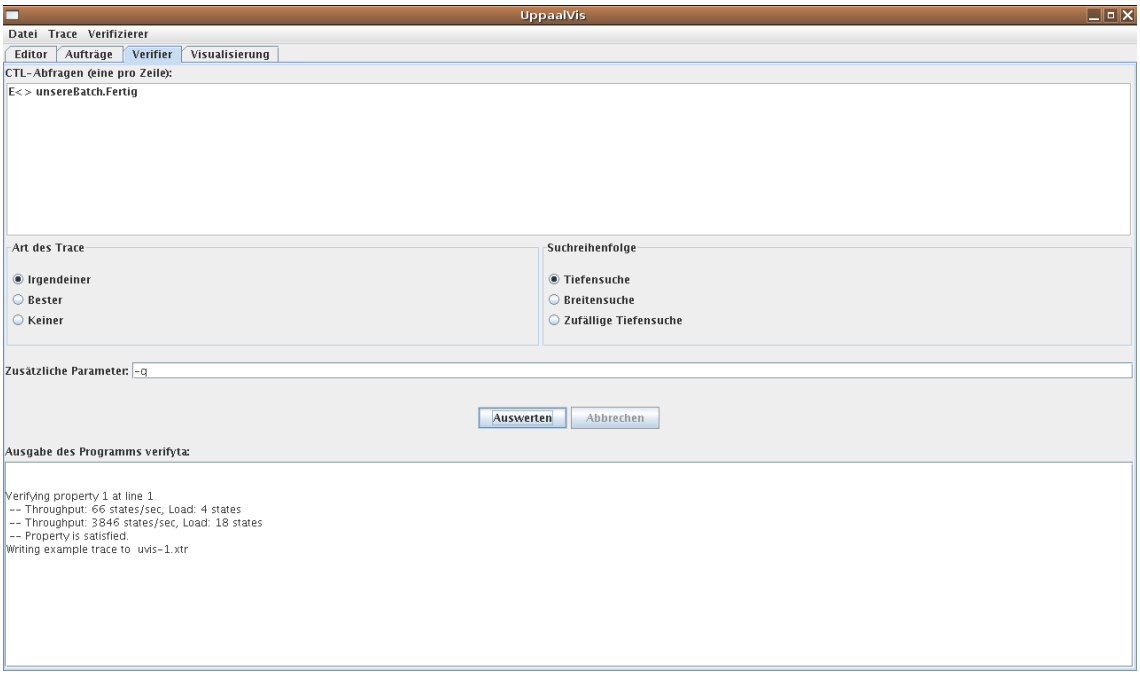


Abbildung 5.2: Der Verifizierer.

5.3.1 Einstellungen

Bevor man den Verifizierer benutzen kann, muss man den Pfad festlegen, wo dieser sich befindet. Dies kann man unter „Verifizierer“ -> „Verifizierer-Einstellungen...“. Des Weiteren muss man die Art des Traces und den Suchalgorithmus des ModelCheckers angeben. Bei der Art des Traces wird zwischen einem zufälligen, dem besten oder dem schnellsten Trace unterschieden. Die Suchreihenfolge ist entweder Breitensuche, Tiefensuche oder Tiefensuche mit zufällig gewähltem Pfad. Weitere Parameter des

Verifizierers kann man manuell in der entsprechenden Zeile festlegen. Abbildung 5.2 zeigt eine Darstellung der grafischen Einbindung des Verifizierers in UppaalVis.

5.3.2 Bedienung

Zunächst muss eine Anfrage in CTL eingegeben werden. Genaueres hierzu ist in Kapitel 2.2.4 nachzulesen. Nachdem die entsprechenden Einstellungen des Verifizierer festgelegt wurden, kann diese Anfrage ausgewertet werden. Im Ausgabefenster können die Statusmeldungen des Kommandozeilenverifizierers verfolgt werden. Der aus dieser Anfrage resultierende Trace wird in eine Datei geschrieben, die zur Simulation der Fahrzeugbewegungen in der Visualisierungsumgebung genutzt werden kann.

5.4 Visualisierung

Die *Visualisierungsumgebung* von UppaalVis ermöglicht eine grafische Darstellung eines modellierten Schienensystems. Auf Basis einer Trace-Datei, die mit Hilfe von Uppaal oder dem in Kapitel 5.3 beschriebenen Verifizierers erzeugt wurde, ist es möglich die Bewegung der Fahrzeuge in diesem Schienensystem zu simulieren. In Abbildung 5.3 ist der Aufbau der Visualisierungsumgebung dargestellt.

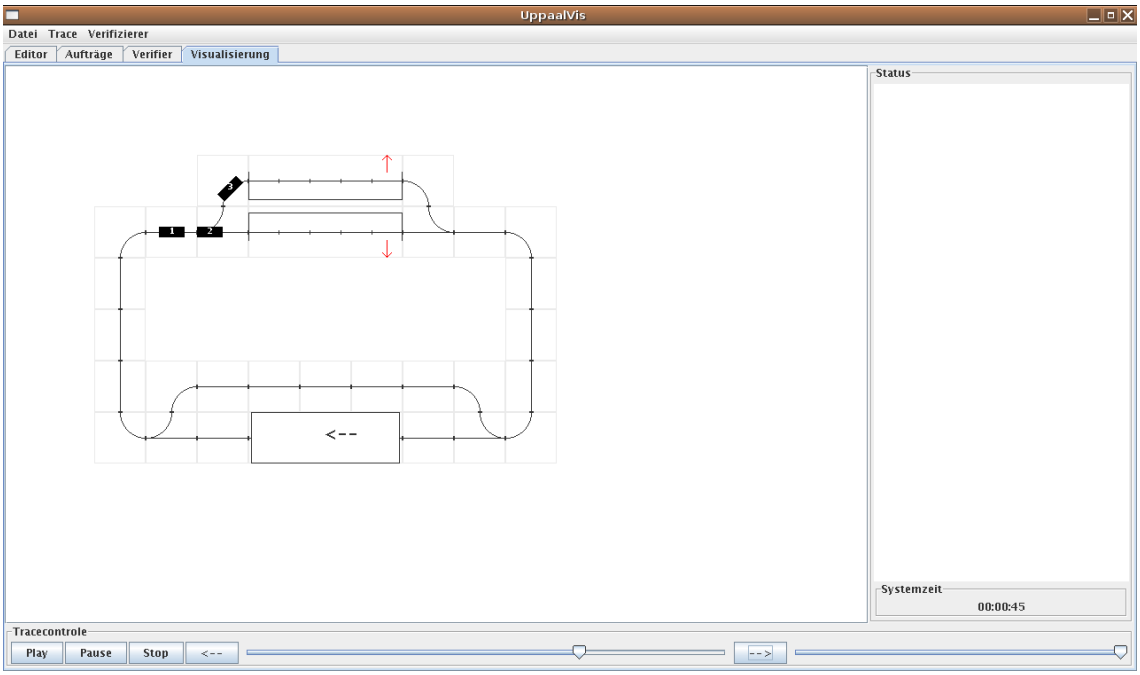


Abbildung 5.3: Die Visualisierungsumgebung.

5.4.1 Bedienung

Der größte Bereich des Fensters stellt das geladene Schienensystem grafisch dar. Es besteht die Möglichkeit, mit Hilfe des Mausrades die Darstellung zu vergrößern. Auf diese Weise ist sichergestellt, dass man sich einzelne Bereiche eines großen Modells detailliert anzeigen lassen kann. Im Gegensatz dazu erlaubt eine verkleinerte Darstellung, alle Fahrzeugbewegungen im gesamten Modell zu verfolgen. Unterhalb des Schienensystems befindet sich eine *Kontrollleiste*, mit deren Hilfe der Ablauf der Simulation gesteuert werden kann. Man hat die Möglichkeit, eine Simulation zu starten und diese zu jedem Zeitpunkt auch anzuhalten. So kann man sich für den entsprechenden Zeitpunkt einen Überblick über die Fahrzeugverteilung verschaffen. Die Simulation kann sowohl schrittweise durchgeführt werden also auch automatisch ablaufen. Bei einer automatischen Simulation kann mit Hilfe eines Schiebereglers die Simulationsgeschwindigkeit festgelegt werden.

Im *Statusfenster* kann man sich Informationen über ausgewählte Fahrzeuge und Schienenelemente anzeigen lassen. Unterhalb des Statusfensters wird die simulierte Zeit dargestellt.

5.4.2 Laden eines Modells

Das zu simulierende Modell muss zunächst in die Visualisierungsumgebung geladen werden. Zu der Uppaal-XML-Datei muss zusätzlich eine in Kapitel 5.2 beschriebene Positionsdatei existieren. Daher können nur Modelle geladen werden, die auch mit dem Editor von UppaalVis erstellt wurden.

Zunächst werden aus der Positionsdatei die Informationen über die geografische Anordnung der einzelnen Komponenten und die Anzahl der globalen Variablen gelesen. Die Schienenelemente werden dann entsprechend im Raster der Visualisierungsumgebung platziert. Die Übergabepunkte werden mit dem in Kapitel 5.2 vorgestellten Tiefensuche-Algorithmus vergeben. Dies stellt sicher, dass die Verteilung der Punkte dieselbe ist, wie sie im Modell kodiert sind. Diese Vorgehensweise erspart das zeitaufwändige Parsen der Übergabepunkte aus dem XML-Modell.

Aus der XML-Datei werden daraufhin die Prozessdefinitionen mit Hilfe regulärer Ausdrücke geparst und als entsprechende Objekte instanziiert. Diese Prozesse werden für das Parsen der Trace-Datei benötigt. Zu jedem Prozess wird die Anzahl der lokalen Variablen auf Basis der Vorlage, die der Prozess instanziiert, gesetzt.

Nun sind alle benötigten Informationen des Modells verfügbar, die zur Simulation einer Trace-Datei gebraucht werden.

5.4.3 Laden einer Trace-Datei

Nachdem man ein Modell in die Visualisierungsumgebung geladen hat, kann man nun eine aus diesem Modell erstellte Trace-Datei laden. Der Aufbau der Trace-Datei

ist u. a. auf **http://groups.yahoo.com/group/uppaal/message/293** beschrieben und sieht wie folgt aus:

```
<trace>      ::= <state> (<state> <transition>)*
<state>      ::= <locvec> <zone> <varvec>
<locvec>     ::= (<location> <nl>)* <dot> <nl>
<zone>       ::= (<clock> <nl> <clock> <nl> <bound> <dot>)* <dot> <nl>
<varvec>     ::= (<NUM> <nl>)* <dot> <nl>
<transition> ::= (<process> <space> <edge> <nl>)+ <dot> <nl>
```

Ein *Trace* besteht aus einem *Startzustand* und einer endlichen Folge von *Zustands-Transitions-Paaren*. Zustände werden durch einen *Knotenvektor*, eine *Zone*, die die Werte der Uhren repräsentiert und einen *Variablenvektor* beschrieben. Der Knotenvektor besteht aus Integerwerten, die durch Zeilenumbrüche getrennt sind. Er wird durch einen Punkt abgeschlossen. Die Integerwerte repräsentieren die entsprechenden Knoten der Zustände.

Eine Zone ist eine endliche Folge von Blöcken mit drei Integerwerten, gefolgt von einem Punkt. Alle Zeichen werden durch einen Zeilenumbruch getrennt. Die ersten beiden Integerwerte geben zwei Uhren an, deren Differenz im dritten Wert kodiert wird. Die Werte kodieren Bedingungen der Form $x - y < b$ oder $x - y \leq b$. Der dritte Wert definiert die Differenz als $2 * b$ für „echt kleiner“ und als $2 * b + 1$ für „kleiner“. Die gesamte Zone wird ebenfalls mit einem Punkt abgeschlossen.

Variablenvektoren sind eine Auflistung der Variablenwerte. Die Reihenfolge der Werte ergibt sich aus der Reihenfolge der *Systemdefinition* im Modell. Zunächst werden die globalen Variablen aufgelistet, darauf die lokalen Variablen. Die Werte werden ebenfalls durch Zeilenumbrüche getrennt und von einem Punkt abgeschlossen.

Eine Transition wird als Folge von zwei Werten, getrennt durch ein Leerzeichen, beschrieben. Der erste Wert gibt die Nummer des Prozesses an, der zweite die Kante, die den Übergang der Transition markiert. Transitionen werden ebenfalls durch einen Punkt abgeschlossen. Eine Trace-Datei könnte wie folgt aussehen:

```
1       //Knotenvektor des Startzustandes (es gibt nur einen Automaten)
.       //Ende des Knotenvektors
0       //-------------------------
1       //Zone des Startzustandes |
0       //-------------------------
.       //Ende des Dreierblockes
.       //Ende der Zone
1       //-----------------------------------
0       //                                  |
0       //Variablenvektor des Startzustandes |
0       //                                  |
0       //-----------------------------------
```

```
13 .       //Ende des Zustandes
14 0       //Knotenvektor des nächsten Zustandes
15 .       //Ende des Knotenvektors
16 0       //------
17 1       //Zone |
18 0       //------
19 .       //Ende des Dreierblockes
20 .       //Ende der Zone
21 2       //-----------------
22 0       //                |
23 0       //Variablenvektor |
24 0       //                |
25 0       //-----------------
26 .       //Ende des Variablenvektors
27 0 2     //Transition
28 .       //Ende der Transition
29 1       //Knotenvektor
30 .       //Ende des Knotenvektors
31 0       //------
32 1       //Zone |
33 0       //------
34 .       //Ende des Dreierblockes
35 .       //Ende der Zone
36 2       //-----------------
37 1       //                |
38 0       //Variablenvektor |
39 0       //                |
40 0       //-----------------
41 .       //Ende des Variablenvektors
42 0 1     //Transition
43 .       //Ende der Transition
```

Der Menüpunkt „Trace“ ermöglicht es eine Trace-Datei zu laden und Einstellungen am Parser vorzunehmen. Man kann mit Hilfe des Unterpunktes „Parser Einstellungen...“ (siehe Abbildung 5.4) festlegen, ob die Trace-Datei statisch oder dynamisch geparst werden soll. Im Falle des statischen Parsens wird jeder Simulationsschritt aus der Trace-Datei gespeichert. Somit muss die Datei nur ein einziges Mal durchlaufen werden. Bei großen Dateien kann das komplette Parsen jedoch sehr zeitaufwändig sein. Deshalb hat man zusätzlich die Möglichkeit des dynamischen Parsens. Hierbei wird die Datei während der Simulation geparst. Die einstellbare Puffergröße erlaubt es jedoch, die Anzahl der Zustände festzulegen, die als Objekte in einer Liste abgelegt werden. Auf diese kann während der Simulation direkt zugegriffen werden. Innerhalb dieses Puffers kann während der Simulation ohne nennenswerte Verzögerung hin und her gesprungen werden. Die maximale Größe des Puffers ist auf 100 Zustände begrenzt. Wenn die Puffergröße die Anzahl der Zustände überschreitet, wird die Größe entsprechend angepasst. Ist die Anzahl der wichtigen Zustände gleich der Puffergröße, so besteht kein Unterschied zwischen statischem und dynamischem

Parsen.

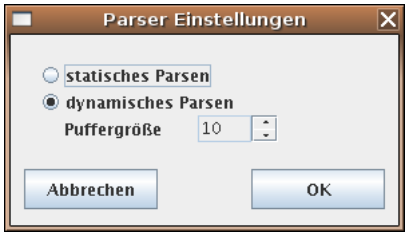


Abbildung 5.4: Einstellungen des Parsers.

Hat man eine zu ladende Trace-Datei gewählt, werden zunächst die Variablenvektoren der Trace-Datei bestimmt, welche entscheidend für die spätere Simulation sind. Das sind die Vektoren, in denen sich entweder die Position eines Fahrzeuges, dessen Ladung oder die Menge der transportierten Artikel ändert. Die Zeilenposition, die jeweils den Anfang eines solchen Variablenvektors beschreibt, wird zusammen mit dem zugehörigen Zeitpunkt der Simulation in einer Liste gespeichert.

Nach dieser Vorverarbeitung der Trace-Datei kann die Simulation gestartet werden. Hierzu werden nacheinander die in der Liste gespeicherten Positionszeilen der Variablenvektoren angesprungen. Zur Auswertung der Variablenvektoren müssen zunächst die globalen Variablen, deren Anzahl aus der Positionsdatei bekannt ist, übersprungen werden.

Auf die globalen Variablen folgen die lokalen Variablen der einzelnen Prozesse. Die Anzahl der zu überspringenden Zeilen nicht benötigter Prozesse ergibt sich aus der Reihenfolge der Prozesse in der Systemdefinition des Modells und deren Anzahl lokaler Variablen. Diese Informationen wurden bereits beim Parsen des Modells bestimmt. Die benötigten Variablen, die für die Simulation von Bedeutung sind, können nun ausgelesen werden. Die Positionen und Ladungen aller Fahrzeuge werden in einem Zustand gespeichert, der an die GUI übergeben wird. Die Fahrzeuge können nun anhand dieses Zustandes positioniert werden. Auf Basis der berechneten Übergabepunkte der Schienenelemente ergibt sich eine eindeutige Position für jedes Fahrzeug.

Eine in UppaalVis geladene Trace-Datei kann zusätzlich in eine Austauschdatei im XML-Format exportiert werden, die von *jABC* importiert werden kann. Auf diese Weise können die durch den Verifizierer errechneten Abläufe ebenfalls mit *jABC* simuliert werden. Auf das Austauschformat wird in Kapitel 6 näher eingegangen.

5.5 Tests

Zur Vervollständigung der Entwicklung von UppaalVis wurden die wichtigsten Algorithmen getestet. Zu diesem Zweck wurden die einzelnen Testfälle in einer *jUnit*-Testklasse zusammengefasst.

5.5.1 Verteilung der Übergabepunkte

Um die Korrektheit des in Kapitel 5.2 beschriebenen Tiefensuche-Algorithmus zur Verteilung der Übergabepunkte zu testen, wurden vier verschiedene Modelle manuell erstellt:

1. Ein korrektes geschlossenes Modell, das alle Komponenten enthält,
2. Fehlerbehandlung eines Modells, das mehr als eine Blackbox enthält,
3. Fehlerbehandlung eines nicht geschlossenen Modells,
4. Fehlerbehandlung eines Modells, das kein Einbahnstraßensystem beschreibt.

Die erstellten Modelle dienen als Eingabe für den zu testenden Algorithmus. Im ersten Fall entsprach die Verteilung der Übergabepunkte des Algorithmus der erwarteten und zuvor manuell berechneten Verteilung. In den übrigen Fällen wurde, wie erwartet, der Algorithmus mit einer entsprechenden Fehlerbehandlung abgebrochen.

5.5.2 Routingtabelle erstellen

Für das Testen des Algorithmus zur Erzeugung der Routingtabelle wird zunächst ein korrektes Modell erzeugt. Nach dem Aufruf des Algorithmus zur Vergabe der Übergabepunkte, wird das Routing durchgeführt. Der dadurch erstellte String, der als Ergebnis die Routingtabelle beschreibt, wird mit der erwarteten, manuell erstellten Rückgabe verglichen.

5.5.3 Parametereinstellungen

Den Komponenten eines manuell erstellten Modells wurden unterschiedliche Parameter übergeben. Dieses Modell wurde in einem `File`-Objekt gespeichert. Die Prozessdefinitionen wurden mit dem entsprechenden Algorithmus wieder ausgelesen. Die Parameterliste der einzelnen Prozesse entsprach den zuvor übergebenen Parametern.

5.6 Fazit

Mit UppaalVis wurde ein Tool erstellt, das die Arbeit mit Uppaal im Hinblick auf die Aufgabenstellung enorm erleichtert. Da erstellte Schienensysteme in Uppaal lediglich als Automaten dargestellt und simuliert werden können, hat man nicht die Möglichkeit, Fahrzeugbewegungen und die Abläufe innerhalb des Systems genau zu verfolgen. UppaalVis bietet hier eine komfortable Hilfe zur Erstellung von komplexen Modellen und deren Simulation. Die Einbindung des Verifizierers in UppaalVis macht

es möglich, auf die Anwendung von Uppaal nach der Erstellung der Automaten komplett zu verzichten. Man erstellt ein Modell mit Hilfe des Editors und erzeugt eine zugehörige Batch. Dieses Modell kann nun verifiziert werden, um einen Trace zu erzeugen. In der Visualisierungsumgebung kann das Modell mit dem erstellten Trace nun simuliert werden. Dieser Ablauf ist weitaus komfortabler, übersichtlicher und weniger fehleranfällig, als es die Arbeit mit UPPAAL gewesen wäre. Zusätzlich hat man so eine Schnittstelle zwischen den beiden Werkzeugen UPPAAL und *jABC* geschaffen.

6 Zusammenführung der Modelle

6.1 Motivation

Die Aufgaben der Projektgruppe wurden, wie bereits beschrieben, von zwei Teilgruppen bearbeitet. Um die Ergebnisse der Arbeit mit den beiden Werkzeugen, *jABC* und UPPAAL, miteinander vergleichen zu können, benötigt man ein entsprechendes einheitliches Modell. Die Ergebnisse des UPPAAL-Verifizierers müssen dabei als Eingabe für das *jABC* verwendet werden können. Somit hat man sich auf ein Austauschformat geeinigt, das mit UppaalVis erstellt und von *jABC* eingelesen werden kann. Der gleiche Ablauf, der von UPPAAL berechnet wurde, kann somit vom *jABC* simuliert werden.

6.2 Das gemeinsame Modell

Das Modell, das zum Vergleich der Ergebnisse der beiden Teilgruppen dient, ist in den Abbildungen 6.1 (Darstellung in UppaalVis) und 6.2 (Darstellung in *jABC*) zu erkennen. Die Komplexität des Modells ist durch die Größe des Suchraums, den der UPPAAL-Model-Checker aufbaut, stark eingeschränkt. Ein Modell der Größe des IKEA-Lagers ist nicht verifizierbar.

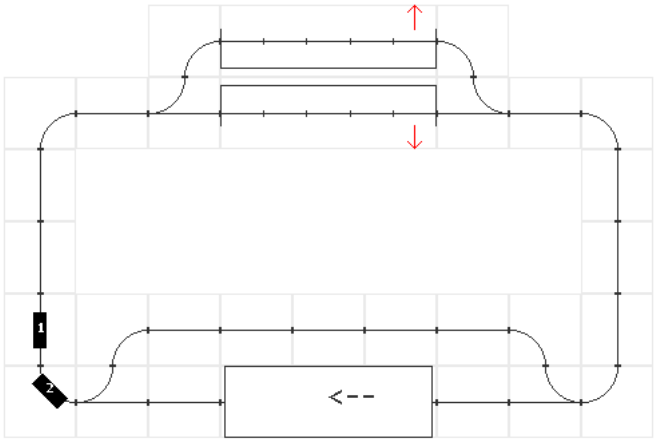


Abbildung 6.1: Das gemeinsame Modell in UppaalVis.

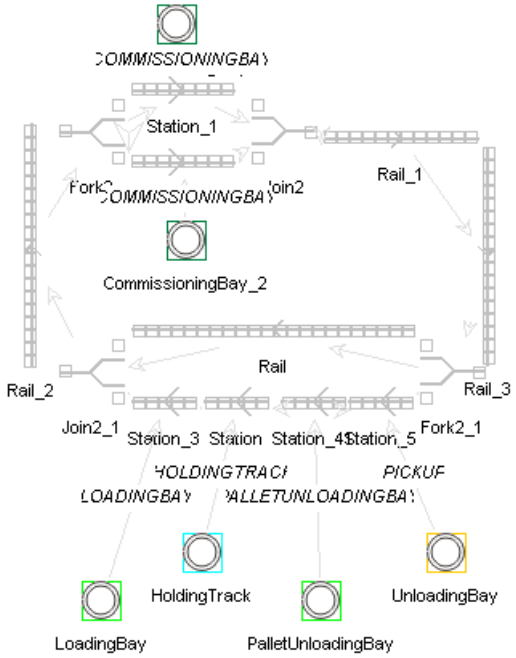


Abbildung 6.2: Das gemeinsame Modell in *jABC*.

Aus diesem Grund wurde ein einfaches Modell, das zwei Bahnhöfe und eine abstrahierte Be- und Entladezone beinhaltet, zugrunde gelegt. Die physikalische Länge simulierter Schienenstücke beträgt jeweils 3 Meter, die zu fahrende Maximalgeschwindigkeit für Schienen und Fahrzeuge jeweils 1 Meter pro Sekunde. Die Be- und Entladezeiten wurden in beiden Modellen angeglichen, so dass der komplette Prozess zur Abarbeitung einer Batch jeweils die gleiche Zeit beansprucht.

Die Anzahl der LKW-Aufträge, der Palettenaufträge, der Positionen pro Palette, der unterschiedlichen Artikel im Lager sowie die Anzahl der Fahrzeuge können jeweils noch bestimmt werden. Dabei ist jedoch zu beachten, dass auch diese Parameter durch den Verifizierer von Uppaal eingeschränkt sind. Da der Suchraum mit der Menge der Aufträge exponentiell wächst, haben wir uns auf einen LKW-Auftrag mit drei Palettenaufträgen beschränkt. Jede Palette erhält zwei unterschiedliche Artikel, wobei fünf Artikelarten im Lager verfügbar sind. Die Anzahl der Fahrzeuge wurde auf zwei beschränkt.

6.3 Berechnen einer optimalen Lösung in Uppaal

Das oben beschriebene Modell wird mit Hilfe des Uppaal-ModelCheckers verifiziert. Die Auswertung der Anfrage `E<> unsereBatch.Fertig` mit der Einstellung „Breitensuche“ ergibt gemäß der Uppaal-Modellierung den optimalen Ablauf, um die Batch, die sich aus den LKW-Aufträgen zusammensetzt, abzuarbeiten. Das Ergeb-

nis wird als Trace-Datei abgespeichert und kann anschließend als Austauschformat exportiert werden.

Dieses Format wird durch eine XML-Datei beschrieben, die von *jABC* eingelesen werden kann. Sie enthält Informationen über die Verteilung der Aufträge auf die Bahnhöfe, sowie die Reihenfolge und die Zeitpunkte zu denen die einzelnen Fahrzeuge die Be- und Entladezone verlassen. Zusätzlich werden die Ladung und die Ziele der Fahrzeuge übergeben. Das Austauschformat ist durch die Datei „UppaalJABC.dtd“ wohldefiniert. Die Struktur einer solchen XML-Datei ergibt sich wie folgt:

```
<trace>
        <palletorders>
                <pallet id="id">
                        <station>id</station>
                        <cargocount id="id">count</cargocount>
                        .
                        .
                        .
                        <cargocount id="id">count</cargocount>
                </pallet>
                .
                .
                .
                <pallet id="id">
                        <station>id</station>
                        <cargocount id="id">count</cargocount>
                        .
                        .
                        .
                        <cargocount id="id">count</cargocount>
                </pallet>
        </palletorders>

        <movement>
                <car id="id" time="time">
                        <cargocount id="id">count</cargocount>
                        <targets>
                                <unload id="id">count</unload>
                                .
                                .
                                .
                                <unload id="id">count</unload>
                        </targets>
                </car>
                .
                .
                .
                <car id="id" time="time>
                        <cargocount id="id">count</cargocount>
                        <targets>
```

```
41                                  <unload id="id">count</unload>
42                                  .
43                                  .
44                                  .
45                                  <unload id="id">count</unload>
46                          </targets>
47                  </car>
48          </movement>
49  </trace>
```

Auf diese Weise wird sichergestellt, dass der optimale, von Uppaal errechnete Ablauf eindeutig beschrieben wird und von *jABC* simuliert werden kann.

6.4 Übernahme von Uppaal in das *jABC*

Bei der Übernahme in das *jABC* sind eine Reihe von Modifikationen am Modell notwendig. Da in Uppaal lediglich ein einzelnes Bauelement für das Beladen von Paletten, das Entladen von Leerpaletten, das Entladen von Anbruchpaletten und das Wartegleis vorgesehen ist, muss dies im *jABC* Modell berücksichtigt werden. Hierzu ist eine besondere Reihenfolge der einzelnen Elemente notwendig, um spätere Diskrepanzen der Modelle während der Simulation zu vermeiden. Es wird daher im *jABC* zuerst die UnloadingBay, dann die PalletunloadingBay, dann der HoldingTrack und zuletzt die LoadingBay integriert (siehe Abbildung 6.2). So ist eine Durchfahrt in korrekter Reihenfolge, so wie es auch in Uppaal geschieht, realisierbar. Beispielsweise ist es einem EHB-Fahrzeug somit möglich eine Leerpalette oder auch eine Anbruchpalette zu entladen, danach unmittelbar in den HoldingTrack zu fahren und hier ein neues Ziel entgegenzunehmen. Fährt ein EHB-Fahrzeug aus dem HoldingTrack heraus, so fährt es als erstes immer durch die LoadingBay, wodurch keine Verzögerungen aufkommen. Damit entsteht ein identischer Fahrtweg zu Uppaal. Hierfür ist es ebenfalls notwendig, dass alle Bahnhöfe in Uppaal und in *jABC* identisch bezeichnet sind. Ansonsten ist es nicht möglich Ziele korrekt zuzuordnen. Weiterhin mussten diverse Parametereinstellungen angeglichen werden, welche in Uppaal anders oder gar nicht vorgesehen sind. Beispielsweise wurden sämtliche Geschwindigkeiten der Fahrzeuge und die erlaubten Geschwindigkeiten auf den Streckenabschnitten fest auf $1m/s$ gesetzt.

Um die entsprechende XML-Datei in *jABC* einlesen zu können, wurde ein neuer Menüeintrag „Uppaal-Interpretierer“ für das Plugin erstellt, welcher das Auswählen dieses speziellen Dateiformates erlaubt. Je nach Anzahl benötigter Fahrzeuge werden diese dann im HoldingTrack bereit gestellt. Die entsprechenden Kommissionierpaletten werden in den entsprechenden Kommissionierzonen erzeugt und entsprechende Artikelpaletten beim Hochregallager in Auftrag gegeben, ähnlich wie dies bei der Übernahme einer Batch passiert. Eine nähere Beschreibung hierzu wird im folgenden Kapitel 6.5 vorgenommen.

Nach der Initialisierung ist es nun möglich, die Simulation wie gewohnt zu starten und auch zu jeder Zeit wieder anzuhalten, um beispielsweise bestimmte Situationen genauer zu betrachten.

6.5 Verteilung der Paletten in das $jABC$ Modell

Die UPPAAL-Lösung gibt die Verteilung der Paletten auf die Kommissionierbahnhöfe so wie den Zeitpunkt der Auslagerung einzelner Paletten an. Dementsprechend wird die Verteilung sowie der Auslagerungsvorgang für die Bearbeitung von UPPAAL-Lösungen umgestellt. Die Kommissionierbahnhöfe werden, wie sonst auch, auf noch freie Plätze für Kommissionierpaletten überprüft. Wird ein solcher Platz gefunden, so wird nun nach möglichen Paletten im UPPAAL-Trace gesucht, die für diesen Kommissionierbahnhof vorgesehen sind. Da es sich um eine sequentielle Suche handelt, wird die korrekte, durch UPPAAL vorgegebene, Reihenfolge eingehalten. Wird eine passende Palette gefunden, so wird diese zugeordnet und aus der Liste der zu verteilenden Paletten gelöscht. (Für die Datenstrukturen sei auf Kapitel 3.7 verwiesen). Das Auslagern von Paletten geschieht nun zu dem von der UPPAAL-Lösung vorgegebenen Zeitpunkt. Jede Runde der Simulation wird die Auslagerungszeitpunkt jeder Ladung mit der aktuellen Simulationszeit verglichen. Sind diese gleich, so wird die Paletten dem Auslagerungsvorgang zugeführt. Da als Datenstruktur eine sortierte Liste gewählt wurde, wird hier ebenfalls eine optimierte sequentielle Suche durchgeführt.

7 Abschlussbewertung

Das erste Semester nutzen den beiden Teilgruppen in erster Linie für die Erstellung einer Modellierungs- bzw. Simulationsbasis. Die *jABC*-Teilgruppe erstellte SIBs für die verschiedenen Bereiche des Lagers und des Schienensystems. Die UPPAAL-Teilgruppe setze die Vorgaben in entsprechende Automatenvorlagen um und implementierte das Visualisierungswerkzeug UppaalVis. Das Routing und die Realisierung eines Fahralgorithmus wurden ebenfalls von beiden Teilgruppen implementiert. Der erste Teil des zweiten Semesters wurde in der *jABC*-Teilgruppe zu weiteren Verbesserungen genutzt. So wurde die gesamte Simulationsumgebung an eine zwischenzeitlich erschienene, neue *jABC*-Version angepasst. Das Routing wurde verfeinert und der Fahralgorithmus optimiert. Die UPPAAL-Teilgruppe optimierte stetig die Automatenvorlagen hinsichtlich Speicherverbrauch und Verifikationsgeschwindigkeit. Der Schwerpunkt der Bemühungen wurde auf UppaalVis verlagert, um die Möglichkeit einer Modellerstellung über UppaalVis zu schaffen.

Die ursprüngliche Vorgabe, mit UPPAALoptimale Abläufe zu gewinnen, um daraus gute Strategien abzuleiten, beispielsweise, welcher Bahnhof zuerst mit Aufträgen und Ladung zu versorgen ist, stellte sich bald als nicht direkt umsetzbar heraus. Die Gewinnung der optimalen Abläufe ist ein, im komplexitätstheoretischen Sinne, schwieriges Problem. Das bedeutet ebenfalls, dass die Speicher- und Rechenzeitanforderungen der Simulation des realen Schienensystems schlicht zu groß sind. Ein Ableiten optimaler Abläufe für das reale Schienensystem, und damit von Strategien, ist damit ebenfalls nicht möglich. Aufgrund dieser Tatsache verlagerte sich der Schwerpunkt der *jABC*-Teilgruppe auf die Entwicklung einer komfortablen, leicht erweiterbare Simulationsumgebung. Dies beinhaltet unter anderem Funktionen für das unkomplizierte Generieren von zu kommissionierenden Ladungen, eine Replay-Funktion, diverse Lade- und Speicherfunktionen sowie Inspektoren um aktuelle Informationen zu Simulationparametern zu erhalten. In der UPPAAL-Teilgruppe wurde UppaalVis zu einer komfortablen Schnittstelle zu den erstellten Automatenvorlagen weiterentwickelt. Es wurden Möglichkeiten für die Erstellung von Schienenmodellen, Visualisierung von Fahrsimulationen und Durchführen von Verifikationsabfragen geschaffen. Gleichzeitig wurde ein minimales, per Hand optimiertes, Schienenmodell einer Simulation von zwei Bahnhöfen, zwei Fahrzeugen und zwei Kommissionierpaletten erstellt. Dieses kann vollständig simuliert und verifiziert werden.

Mit diesem Modell wurde eine Zusammenführung der beiden Teilgruppen erreicht. Damit ist eine Lösung des Problems mit UPPAAL und eine anschließende Simulation mit *jABC* möglich.

Abschließend lässt sich sagen, dass die Projektgruppe mit ihren Gesamtergebnissen die gesteckten Ziele erreicht hat, auch wenn man die ursprünglich angestrebte Lösung eines realen Problems aus der Praxis nicht erreichen konnte.

Literaturverzeichnis

[Ali99] Alicke, K.: *Modellierung und Optimierung von mehrstufigen Umschlagsystemen*. Karlsruhe, 1999.

[Beh04] Behrmann, Gerd: *A Tutorial on Uppaal*. Technischer Bericht, Department of Computer Science, Aalborg University, Denmark, 2004.

[BS98] B. Sauer, K. Welti: *Einfach-Optimal-Effizient*. VDI-Verlag, 1998.

[EC00] E. Clarke, O.Grumberg, D.Peled: *Model checking*. MIT Press, 2000.

[Gud99] Gudehus, T.: *Logistik*. Springer-Verlag, 1999.

[JAB] *jABC – The Java Application Building Center*. `http://www.jabc.de`.

[JST] *XStream – a simple library to serialize objects to XML and back again*. `http://xstream.codehaus.org`.

[JUN] *JUnit – a simple framework for writing repeatable tests*. `http://junit.sourceforge.net`.

[Kwi92] Kwijas, R.: *Kommissionieren in Industrie und Handel*. VDI-Verlag, 1992.

[Mar04] Martin, H.: *Transport- und Lagerlogistik*. Vieweg Verlag, 2004.

[RA94] R. Alur, D. L. Dill: *A Theory of Timed Automata*. Theoretical Computer Science, 1994.

[RJ99] R. Jünemann, T. Schmidt: *Materialflußsysteme*. Springer-Verlag, 1999.

[Sch85] Schwarting, C.: *Optimierung der ablauforganisatorischen Gestaltung von Kommissioniersystemen*. Doktorarbeit, Technische Universität Berlin, 1985.

[Upp05] Uppsala University and Aalborg University: *Hilfe zu Uppaal 3.4.11*, Juni 2005. Die Hilfe ist ein Programmbestandteil von Uppaal.