5 Implementierung und Test

In diesem Kapitel wird auf die anschließende Umsetzung des Konzeptes eingegangen. Im Vordergrund steht zunächst die Integration der Klassenbibliothek *JavaSim*. Nach einer Einführung in das Konzept der Bibliothek wird aufgezeigt, wie die Simulationsklassen in das Modell aufgenommen werden können. Eine konkrete Realisierung der Methodenaufrufe zur Prozessverwaltung innerhalb der Simulationsbibliothek wird in Abschnitt 5.2 erläutert. Außerdem wird in diesem Abschnitt die Implementierung der Befehlssteuerung und der Schnittstelle zu den Steuerungsobjekten aufgegriffen. Zum Abschluss des Abschnittes wird ein Beispiel zur Verwendung der erstellten Klassen in einem Benutzerprogramm angegeben.

Im engen Zusammenhang mit der Realisierung des Modells steht der Test der Funktionsfähigkeit, der in Abschnitt 5.3 erörtert wird. Neben der Kontrolle des Programmablaufes wird auch das Laufzeitverhalten in Abhängigkeit zu der Anzahl der generierten Ereignisse gesetzt.

5.1 Die Klassenbibliothek JavaSim

Die Klassenbibliothek *JavaSim* unterstützt eine objektorientierte Simulation, die nach der Methode der diskreten Ereignisse aufgebaut ist. Innerhalb der ereignisgesteuerten Methoden wird der prozessorientierte Ansatz gewählt. Nach Ansicht der Autoren genügt die Bibliothek den folgenden Anforderungen [DCS99-ol]:

- leichte Erlernbarkeit und Benutzbarkeit
- flexible und erweiterbare Struktur
- effizientes Laufzeitverhalten

Die Bibliothek ist in drei Pakete[1] untergliedert, die jeweils eigenständige Bereiche zur Unterstützung von Simulationsmodellen bilden. Es existieren Pakete für:

- die Unterstützung der prozessorientierten Simulationsmethode
- Verteilungen von Zufallsfunktionen
- Statistiken

Eine grafische Darstellung der Simulationsläufe ist in *JavaSim* nicht vorgesehen. Im Folgenden wird auf den Aufbau und die Funktion des Paketes zur Zeitsteuerung eingegangen. Zur Bibliotheksbenutzung und Beschreibung der beiden anderen Pakete wird auf das Handbuch verwiesen [DCS99-ol].

[1] Die Programmiersprache *Java* erlaubt es, zusammengehörige Klassen in einem Paket (package) zusammenzufassen

Wie in Abschnitt 2.2.3 ausgeführt, werden beim prozessorientierten Ansatz Simulationselemente, die zyklischen Aktivitäten unterliegen, als Prozesse angesehen. Jeder Prozess berechnet in Abhängigkeit vom Systemzustand den Zeitpunkt, wann ein nächstes Ereignis eintreten kann, von dem er beeinflusst werden kann. Solange das Ereignis noch nicht eingetreten ist, darf die aktive Phase unterbrochen werden. Die Klassenbibliothek *JavaSim* übernimmt die Kontrolle zur Aktivierung und Deaktivierung der Simulationsprozesse. Den Kern stellt die Klasse *Scheduler*, die intern eine Datenstruktur mit allen Prozessen des Simulationslaufs führt. Die Datenstruktur ist in der vorliegenden Implementierung als *Queue* realisiert, die nach Zeitstempeln sortiert ist. Jeder Zeitstempel bezieht sich explizit auf einen Prozess. Ein Prozess ist höchstens einmal in der Struktur geführt.

Die Bearbeitung des Simulationslaufes ist nun durch die Aktivierung der Prozesse in der Reihenfolge ihrer Zeitstempel vorgegeben. Der Scheduler entnimmt aus der Schlangenstruktur den Prozess, der als nächstes zu einer Zustandsänderung führen kann, setzt die Systemzeit auf den Zeitpunkt der Aktivierung und übergibt dem Prozess die Kontrolle über den Simulationslauf. Dieser nimmt seine Bearbeitung an der Stelle wieder auf, an der er zuletzt unterbrochen worden ist. Nach der Berechnung übernimmt der Scheduler wieder den Ablauf der Simulation.

Der Scheduler kann in seiner Datenstruktur nur Objekte aufnehmen, die Instanzen der Klasse *SimulationProcess* sind. Diese Klasse erbt Eigenschaften der Klasse *Thread* aus der Bibliothek *java.lang.**. Ferner werden Methoden zur Verfügung gestellt, mit deren Hilfe jeder Simulationsprozess sich in die Warteschlange zur Aktivierung einfügen kann. Der Scheduler überwacht bei jeder Eintragung die Gültigkeit der Werte, d. h. die aktuelle Simulationszeit muss kleiner als die Aktivierungszeit des Prozesses sein und der Prozess darf nur einmal in der Warteschlange geführt werden.

Neben der Klasse *SimulationProcess*, die Grundfunktionen zur Steuerung der Simulationsprozesse bereitstellt, werden von *JavaSim* auch Klassen implementiert, die erweiterte Funktionen bereitstellen. Während mit den grundlegenden Funktionen Systeme simuliert werden können, die nach der prozessorientierten Sichtweise ihre Aktivitäten wohlgeordnet ausführen, ermöglichen die erweiterten Funktionen Unterbrechungen in diesem Ablauf. Sie werden eingesetzt, um auf Ereignisse reagieren zu können, die von außen auf das Modell einwirken. Hält bspw. ein Shuttle an einem Haltepunkt, wird der Prozess für eine unbestimmte Zeit unterbrochen. Erst durch eine Nachricht der Steuerung wird das Shuttle wieder gestartet. Simulationsobjekte (Entitäten), die solchen Anforderungen genügen müssen, erben Eigenschaften der Klasse *SimulationEntity*, die eine Erweiterung der Klasse *SimulationProcess* darstellt. Mit Hilfe der *TriggerQueue* ist es möglich, Objekte der Klasse *SimulationEntity* auf spezielle Ereignisse warten zu lassen.

In diesem Abschnitt wurde ein Überblick über die Klassen von *JavaSim* gegeben. In Bild 5-1 werden die Beziehungen, die sich aus den vorherigen Ausführungen ergeben, grafisch veranschaulicht.

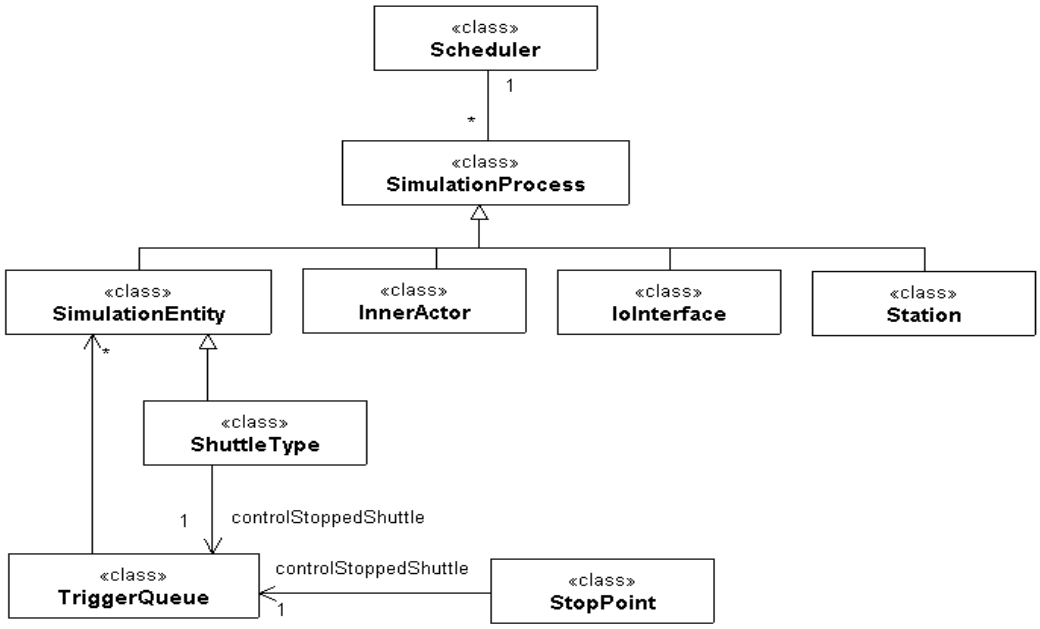

Bild 5-1: Klassendiagramm zur Integration der Klassenbibliothek JavaSim

Auffällig am Klassendiagramm (Bild 5-1) ist die Klasse *TriggerQueue*, die mit den Klassen zur Modellierung der Shuttles und Haltepunkte assoziiert ist. Diese Klasse erlaubt es, Prozesse auf asynchrone Meldungen warten zu lassen. Erreicht ein Shuttle einen Haltepunkt, wird es erst nach unbestimmter Zeit von der Steuerung aktiviert. Zur Modellierung dieses Sachverhaltes wird ein Shuttle, das an einem Haltepunkt wartet, in die Warteschlange der Klasse *StopPoint* eingefügt. Erhält der Haltepunkt von der Steuerung das Signal, dass das Shuttle die Fahrt fortsetzen kann (siehe Abschnitt 4.2.1), werden die Objekte, die sich in der *TriggerQueue* befinden, gestartet. Dieser Vorgang verläuft analog zum Verhindern des Auffahrens eines Shuttles auf das vorausfahrende.

5.2 Implementierung

Das Modell des Simulatorkerns besteht aus verschiedenen Bereichen. Neben der Abbildung der Systemkomponenten durch Klassen und der dynamischen Abhängigkeiten durch die Methodenaufrufe sind Schnittstellen zur Benutzungsoberfläche und zu den Steuerungsobjekten vorgesehen. Diese Gliederung soll in der Implementierung durch den Einsatz von Paketen beibehalten werden. In diesem Abschnitt wird der Aufbau der einzelnen Pakete: *LayoutData*, *SimCommand*, *SimControl*, *Simulation* und *CreateLayoutData* erläutert werden. Ziel ist es, durch

die Angabe ausgewählten Quellcodes[2] die Methodenaufrufe der Prozesssteuerung, Befehlsverwaltung und Schnittstellenbereiche zu verdeutlichen. Im Folgenden werden die einzelnen Pakete nacheinander erläutert:

LayoutData

In diesem Paket sind alle Klassen, die Modellelemente realer Systemkomponenten im Simulatorkern bilden, enthalten. Zumal die Elemente, die keine eigenen Prozesse in der Simulation bilden, durch Klassen realisiert sind, die sich unmittelbar aus der Modellierung ergeben, wird in diesem Abschnitt die Implementierung der Simulationsprozesse vorgestellt. Die Simulationsprozesse implementieren jeweils eine *run*-Methode. Diese Methode zeigt den Verlauf der Prozesse von ihrer Aktivierung bis zur Deaktivierung auf. Die Kontrolle der Prozesse erfolgt nach Abschnitt 5.1 durch die *Scheduler-Klasse* der Klassenbibliothek *JavaSim*. Die folgenden Quellcodebeispiele sollen anhand des übersichtlichen Prozessverlaufes der Klasse *InnerActor* die Benutzung von Methoden zur Aktivierung und Passivierung (siehe Abschnitt 4.2) der Prozesse aufzeigen:

```
public class InnerActor extends SimulationEntity implements Actor {
  ...
  public void run() {
    for (;;) {
      try {
        Passivate();
      }catch(RestartException re){...}
      SendToIoBus sio= new SendToIoBus (inputBus, sensorPort[position]);
      sio.startCommand();
    }
  } ...
}
```

Bild 5-2: Die run-Methode der Klasse „InnerActor“. Nach der Aktivierung des Prozesses sendet das „SendToIoBus-Objekt“ eine Nachricht, dass die Position eingestellt ist.

Die Methode *Passivate* entfernt den Simulationsprozess aus der Warteschlange des Schedulers und setzt ihn in den Status *Passiv* (vgl. Abschnitt 4.2). Aus dem in Bild 5-2 dargestellten Quellcode ist zu entnehmen, dass ein Prozess der Klasse *InnerActor* nach seiner Aktivierung zunächst passiviert wird. Der Prozess wartet, bis ein anderer Prozess seine erneute Aktivierung veranlasst. In diesem Fall muss die Methode zur Einstellung der Streckenführung vom zugeordneten *IoInterface* Prozess aufgerufen werden. Dieser Vorgang ist in Bild 5-3 angegeben.

[2] Der Quellcode liegt der Arbeit auf einer CD-ROM bei.

```
public class InnerActor extends SimulationEntity implements Actor {
  ...
  public void startEvent (int position) {
    ....
    //den aktuellen Prozess nach einer Verzoegerungszeit fortsetzen
    try {
      ActivateAt(Scheduler.CurrentTime()+delay[position]);
    } ...
  } ...
}
```

Bild 5-3: Aktivierung eines „InnerActor-Prozess“ durch den Methodenaufruf „startEvent“. Der Textauszug ist auf den Aufruf zur Aktivierung des Prozesses reduziert.

Der Aufruf der *ActivateAt*-Methode reiht den Prozess zu einem festgelegten Zeitpunkt wieder in die Prozesswarteschlange ein. In diesem Beispiel wird die Verzögerung bei der Einstellung der variablen Streckenelemente nachgebildet.

SimCommand

Die Reaktion auf Ereignisse erfolgt im Simulatorkern über den Einsatz von Befehlen, die Operationsfolgen eines Ereignisses kapseln (Abschnitt 4.3). Die Anweisungen sind im Paket *SimCommand* implementiert.

```
public class SendIdToIoBus extends Command {
  ...
  public SendIdToIoBus(IoInterface r,IRcvCommand rcv, int port) {
    ...
  }

  public void startCommand () {
    int shuttleId = ((ShuttleType)rcv).getElementId();
    String strId = String.valueOf(shuttleId);
    myIoInterface.sendId(port,strId);
  } ...
}
```

Bild 5-4: Der „SendIdToIoBus-Befehl“ erhält die Identifizierung des Elementes und sendet sie an den angegebenen Steuerungsknoten.

Ein Objekt der Klasse *SendIdToIoBus* wird instanziert, wenn die Identifizierung eines Shuttles ausgelesen werden soll. Objekte, die diesen Befehl auslösen, sind einem Prozessinterface (*IoInterface*) an einer vorgegebenen Portnummer zugeordnet. Hingegen besteht keine Verbindung zu dem Shuttle-Objekt. Der Verweis auf den entsprechenden Speicherbereich wird mit dem Methodenaufruf *Create-*

Command dem erzeugenden Objekt übergeben. Um die Änderungen der im Befehl gekapselten Operationen nur im Code dieser Befehlsklasse vornehmen zu müssen, wird dieser Methode als Argument ein Objekt, das die Schnittstelle *IRcvCommand* implementiert, angegeben. So kann das Befehlsobjekt den Typen des mitgelieferten aktuellen Parameters selbst festlegen und das erzeugende Objekt ist unabhängig von Typänderungen. Die Konvertierung zwischen *IRcvCommand* und *ShuttleType* wird vorgenommen, da nicht jedes befehlsempfangende Objekt die Methode zum Auslesen einer Identifizierung implementieren soll. In der vorliegenden Version des Simulatorkerns wird diese Funktion nur von einem Shuttle-Objekt unterstützt. Werden in Folgeversionen weitere Elemente eingeführt, die eine Identifizierung an das Prozessinterface senden müssen, ist an Stelle des konkreten Typen eine Schnittstelle einzusetzen.

SimControl

Der Simulatorkern modelliert mit der Klasse *IoInterface* die Prozessschnittstelle des Steuerknotens. Die Anwendungsprogramme sind als Java-Objekte in den Instanzen der Klasse *ControlObject* zu verwalten. In der Konfigurationsdatei wird zu diesem Zweck angegeben, welche Programme auf dem entsprechenden Knoten ausgeführt werden und welche Ports zur Verfügung stehen (Bild 5-5). Da die Steuerungsobjekte nicht im Kern implementiert sind, muss ein dynamisches Laden der Klassen unterstützt werden. Der Name des Kontrollprogramms muss mit dem Namen einer Klasse übereinstimmen, die im Paket *SimControl* abgelegt ist. Die Festsetzung der Ports ist vom Entwickler der Steuerungsobjekte vorzunehmen.

Konfigurationsdatei

```
****Steuerknoten 1
begin IoInterface
1
begin program
Joiner (Programm)
1,2,3,4,5,6,7,8    (Port)
Brancher
10,11,12,13,14,15,16,17
end program
end IoInterface
```

Auszug aus dem Konstruktor der Klasse ControlObject

```
...
//suchen der Klasse mit den Namen prog
cl = Class.forName("SimControl."+prog);
//ermitteln der Konstruktoren
cs = cl.getConstructors();
//Parameter des Konstruktors setzen
ob = {shuttleNode, programs.elementAt(i+1)};
//Instanz der Klasse im ControlObject sichern
program[i/2] = (ControlProgram)cs[0].newInstance(ob);
...
```

Bild 5-5: Konfiguration eines Steuerknotens zum Laden externer Steuerprogramme. Die Konfiguration ermöglicht es, mehrere Programme in einem Knoten zu verwalten. Die Belegung der Ports ist vom Entwickler der Steuerung festzulegen und muss je Knoten eindeutig sein.

Die Möglichkeit, Steuerungsobjekte über die Konfigurationsdatei austauschen zu können, erweist sich auch im Hinblick auf die Anbindung des Simulatorkerns an

FUJABA als vorteilhaft. Werden unterschiedliche Steuerungsprogramme spezifiziert und der entsprechende Code generiert, können diese durch die Angabe der Klassennamen und der Portnummern beliebig in den Simulationslauf eingebunden werden.

Simulation

Der in dieser Arbeit entwickelte Programmteil des Simulators kann als Bibliothek zur Simulation dezentral gesteuerter schienengebundener Transportsysteme eingesetzt werden. Benutzer der Bibliothek importieren das Paket *Simulation*, das neben der Schnittstelle zum Auslesen der Simulationsdaten (Abschnitt 4.4) auch die Klasse *SimulationController* bereitstellt. Mit Hilfe dieser Klasse kann der Anwender die Konfiguration der Anlage einlesen, die Simulation starten und beenden (Bild 5-6). Des Weiteren verfügt diese Klasse über eine statische *main-Methode*, die ein Fenster zur Steuerung einer Simulation öffnet.

```
import Simulation.*;

public static void main (String argv[]) {

        //Erstellen der Simulationskontrolle
        //false: beende Simulationsprozess beim Halten nicht
        SimulationController sc = new SimulationController(false);

        //Einlesen der Konfigurationsdatei
        sc.loadLayoutData("Querverschiebung.txt");

        //Anmelden eines SimObserver-Objektes
        try {
                DisplayElement de = DisplayElement.getDisplayElement("Shuttle1");
                de.addObserver(new SimObserver() { ....});
        }catch....

        //Starten des Simulationslaufes
        sc.startSimulation();
}
```

Bild 5-6: Beispiel zur Verwendung der Simulationsbibliothek

CreateLayoutData

Das Paket *CreateLayoutData* besteht in der vorliegenden Version aus der Schnittstelle *DataClassLoader* und der Klasse *MyFileLoader*, die eine Konfigurationsdatei einlesen kann. Gemäß der Spezifikation wird eine Klasse erzeugt, die alle Streckenmodule mit ihren Verbindungen zu anderen Objekten beinhaltet. Der Aufbau der Konfigurationsdatei (Anhang B.1) entspricht noch nicht den Anforderungen an ein integriertes System. Da diese Datei zumeist *„von Hand“* erzeugt

worden ist, erfordert der Aufbau eine Struktur, die vom Anwender gelesen werden kann. Von einem Mitarbeiter der FASTEC ist ein Konvertierungsprogramm geschrieben worden, das die Erzeugung einer Konfigurationsdatei aus dem bisher zur Entwicklung der Streckentopologie eingesetzten Programm LONTROL MFS durchführt. Wird der Simulatorkern in dem in [KNN+00] beschriebenen System eingesetzt, kann auf einen Topologieeditor zurückgegriffen werden, der im Rahmen der Projektgruppe *JEVOX (Java Based Visualisation of Executable UML-Specifications)* von der Arbeitsgruppe Softwaretechnik entwickelt worden ist. Dieser Editor erlaubt die Spezifikation einer Topologie, wobei die einzelnen Modellelemente als Instanzen einer domänenspezifischen Klassenbibliothek aufgefasst werden. Er stellt somit ein spezielles Objektdiagramm dar, in dem Weichen, Shuttles und Arbeitsstationen positioniert und angeordnet werden können. Es sind Erweiterungen zur Positionierung von Sensoren und Aktoren vorgesehen, die über Ports den aus der Codegenerierung erzeugten Steuerungsobjekten zugeordnet sind.

5.3 Test

In diesem Abschnitt wird die Funktionsfähigkeit des Simulatorkerns validiert. Hierzu wird der Simulationsablauf mit dem in der Anforderungsanalyse (Abschnitt 3.3) beschriebenen System verglichen. Ferner wird das Laufzeitverhalten in Abhängigkeit zu der Anzahl der Ereignisse ausgewertet.

Im Simulatorkern sind Objekte zur Modellierung der Systemkomponenten: Shuttle, Start-/Stoppeinheiten, Identifikationseinheiten, Passagesensoren und Streckenelemente implementiert. Zusätzlich ist eine Schnittstelle zur Steuerungsebene realisiert. Das komplexe Streckenmodul der Querverschiebung benötigt zur Ablaufsteuerung diese Elemente (Bild 5-7). Die erste Testumgebung bezieht sich daher auf eine Querverschiebung, um anhand des komplexen Ablaufes an diesem Modul den Zusammenhang der Modellelemente zu überprüfen. Die Konfiguration der Querverschiebung ist in Anhang B.2 angegeben.

Die Protokollausgabe in Bild 5-7 enthält die Fahrbewegung des Shuttles und die Einstellungen der variablen Streckenabschnitte. Der Ablauf wird vom Steuerungsknoten kontrolliert, der auf der Anwendungsebene das entsprechende Programm installiert hat. Das Kontrollprogramm steuert die Elemente der Querverschiebung so, dass ein Shuttle vom Streckenteilabschnitt *101002* seine Bewegung auf Abschnitt *101005* fortsetzt.

Während bisher der Ablauf an einem Streckenmodul getestet wurde, wird nachfolgend ein Materialflusssystem mit einer geschlossenen Streckentopologie simuliert. Das Ziel dieses Funktionstests ist, die Reaktion der Steuerung auf unterschiedliche Shuttleidentifizierungen und –positionen aufzuzeigen. Weiterhin ist

die Funktionsfähigkeit der Abstandsüberwachung einzelner Shuttles zu kontrollieren. Die eingesetzte Darstellung der Ereignisse basiert auf dem Java3D-API. Sie zeigt eine Anbindung der Objekte zur dreidimensionalen Darstellung an die Modellelemente des Kerns. Das Layout der Streckenführung wurde mit dem Programm *Lontrol* erstellt, das zum zweidimensionalen Entwurf einer Streckentopologie von der FASTEC entwickelt worden ist. Aus dieser Topologie wird eine Java3D-Darstellung generiert (Bild 5-8). Die Klassen des Paketes *Simulation3D* sind von mir im Rahmen meiner Arbeit als Studentische Hilfskraft am Heinz Nixdorf Institut entwickelt worden. Ein Beispielprogramm liest die Daten des zweidimensionalen Layouts ein und erzeugt eine Darstellung, die aus Java3D-Objekten besteht. Jedes Objekt hat im Simulatorkern ein zugeordnetes Objekt, das die dargestellte Systemkomponente repräsentiert. Die Nachrichten über die Veränderungen des Systemzustandes werden nach der im Abschnitt 4.4 spezifizierten Schnittstelle an die Benutzungsoberfläche weitergeleitet.

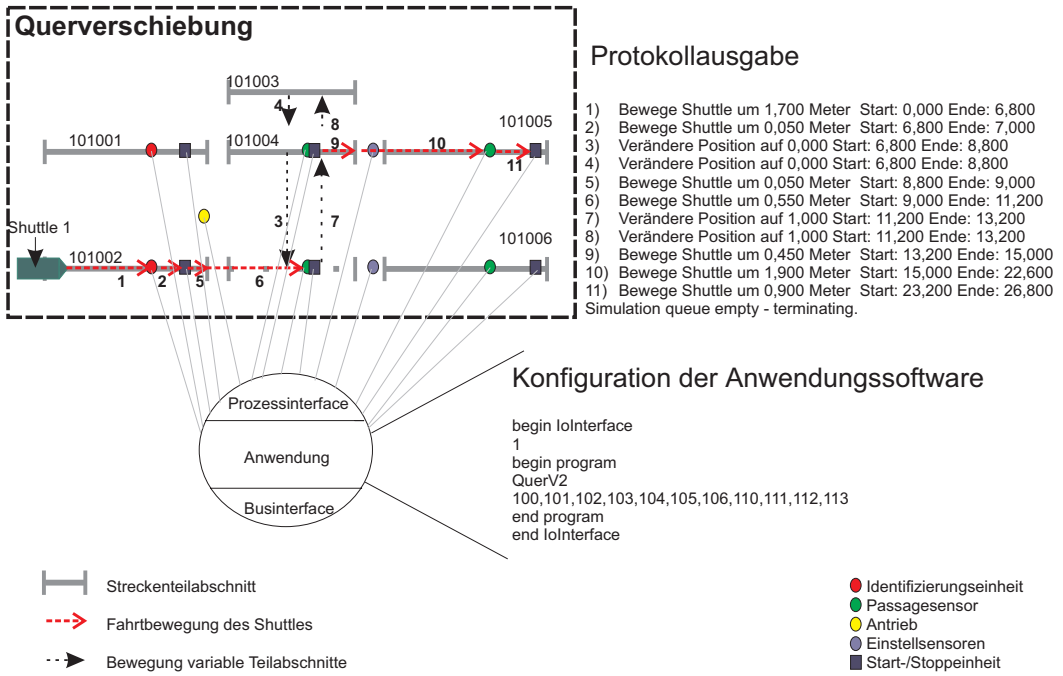

Bild 5-7: Modell einer Querverschiebung. Die angegebene Ausgabe beschreibt den Verlauf der Simulation.

Die Generierung beschränkt sich in der eingesetzten Version auf die Streckenführung. Eine Platzierung der Shuttles und Konfiguration der Kontrollelemente ist in der bisherigen Version des *Lontrol* nicht vorgesehen und wurde zusätzlich vorgenommen.

In Bild 5-9 werden die Prozesse zur Überwachung der exklusiven Elemente, der Verzweigung an einer Weiche und der Abstandskontrolle aufgezeigt.

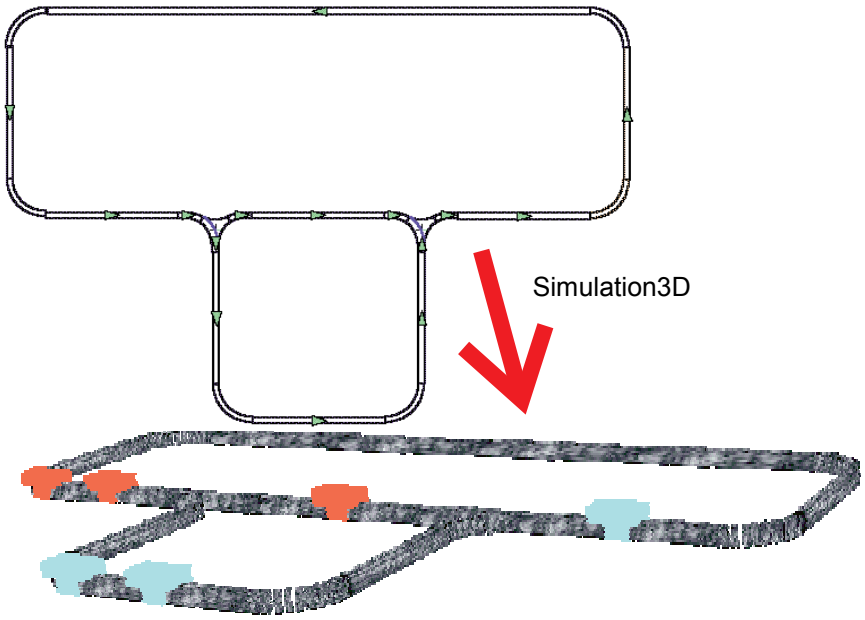

Bild 5-8: Generierung eines Java3D-Layout aus „Lontrol“

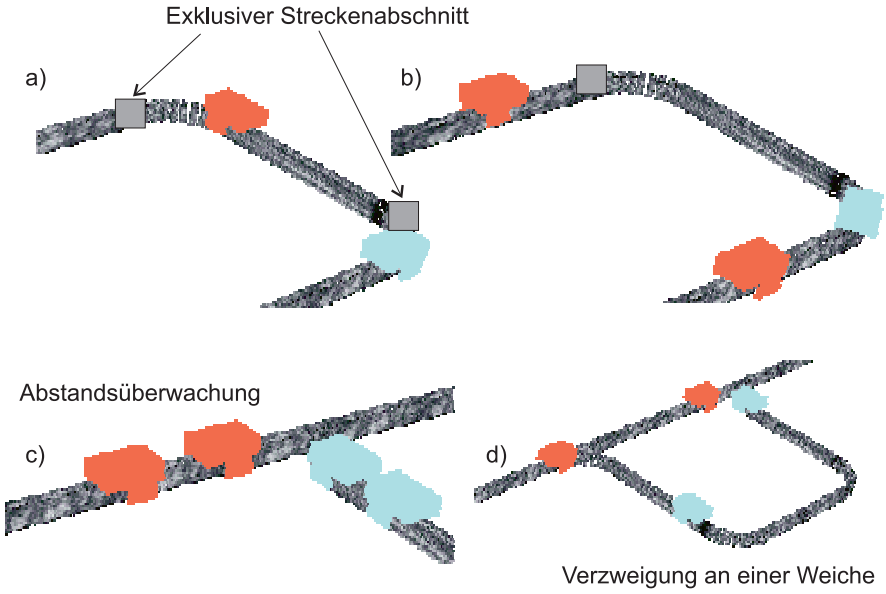

Bild 5-9: Ausschnitte zur Kontrolle der Simulationsergebnisse. Die Bilder a) und b) veranschaulichen den Ablauf an einem exklusiven Streckenabschnitt. Bevor das rote Shuttle den exklusiven Abschnitt nicht verlassen hat, darf das türkis gefärbte nicht weiterfahren. Das Bild c) zeigt, dass wartende Shuttles nicht aufeinander auffahren. Bild d) verdeutlicht die Steuerung an einer Weiche. Die Streckenführung für rote Shuttles wird auf „geradeaus“ eingestellt, während türkise Shuttles abbiegen müssen.

Laufzeitverhalten

Neben der korrekten Berechnung der Ereignisse, ist die Rechenzeit der Simulationsläufe zu beachten. Im Hinblick auf eine interaktive Darstellung mit Echtzeitverhalten muss der Simulatorkern die Ereignisse mindestens so schnell berechnen, dass die virtuelle Zeit so weit wie die reale fortgeschaltet ist. Um Eingriffe zur Laufzeit zu ermöglichen, darf die Simulationszeit auch nicht zu schnell laufen, da sonst auf Benutzereingaben nicht reagiert werden kann. Es ist zu berücksichtigen, dass zukünftige Zustandsveränderungen im gewissen Rahmen im Voraus berechnet werden müssen, damit auch bei kurzfristigen Schwankungen eine fließende Darstellung gewährleistet ist. Die hier aufgeführten Messungen geben Anhaltspunkte für den Einsatz des Simulatorkerns in einer interaktiven Simulation. Zu diesem Zweck wird zunächst der Zusammenhang zwischen der Anzahl der Ereignisse in einem Zeitintervall und der Laufzeit des Simulatorkerns aufgezeigt. Danach werden Faktoren behandelt, welche die Anzahl der Ereignisse beeinflussen.

Die aktuelle Version der prozesskontrollierenden Instanz koordiniert die Synchronisation der Simulationsprozesse. Die Simulationszeit läuft unabhängig von der realen Zeit. Der Scheduler ist daher um eine Funktion zum Zeitabgleich erweitert worden. Es findet ein Vergleich zwischen der Simulations- und der Systemzeit statt.

Die Berechnungen wurden mit einem *Intel-Celeron-Prozessor* ausgeführt, der mit einer Taktrate von 366 MHz betrieben wird. In den Tests steht der Prozessor allein den Berechnungen zur Verfügung; eine grafische Animation wird nicht gleichzeitig ausgeführt, um eine maximale Systemgrösse für interaktive Simulationen in Echtzeit zu bestimmen. Es wird die Streckentopologie, die zuvor in Bild 5-8 dargestellt wurde, berechnet. Die Anzahl der Shuttles variiert im weiteren Verlauf zwischen sechs und 84. In den Kommentaren zu den Auswertungen wird jeweils auf die relevanten Parameter eingegangen.

Die Abbildung 5-10 zeigt den Zusammenhang zwischen den zu bearbeitenden Ereignissen und der Dauer einer Simulation. Auffällig ist der hohe Aufwand in der Startphase des Simulationslaufes. In dieser Phase weisen alle drei durchgeführten Testläufe Verzögerungen von über einer Sekunde auf. In den Berechnungen mit durchschnittlich 36,325 bzw. 72,258 Ereignissen je Sekunde werden nach einer Zeitspanne, in der die Berechnungen zu langsam ausgeführt werden, die Daten wieder in ausreichender Geschwindigkeit erzeugt.

Der Zeitabschnitt, in dem nicht genügend Informationen bereitgestellt werden, muss in einem Zwischenpuffer gespeichert werden. Eine kontinuierliche Animation bei 36,325 Ereignissen je Sekunde benötigt nach der Initialisierungsphase nur einen geringen Speicher, da der Kurvenverlauf geringe Schwankungen aufweist. Bei einer Verdopplung der Ereignisanzahl benötigt ein Simulationslauf ca. 20 Sekunden, um sich der realen Zeit anzunähern. Nach dieser Spanne liegen zeit-

weise Abweichungen vor, die durch einen Datenpuffer abgefangen werden können. Werden im Simulationsmodell durchschnittlich 90 mögliche Zustandsänderungen ermittelt, reicht die Rechenleistung nicht mehr aus, um allein die Berechnungen in realer Zeit umzusetzen.

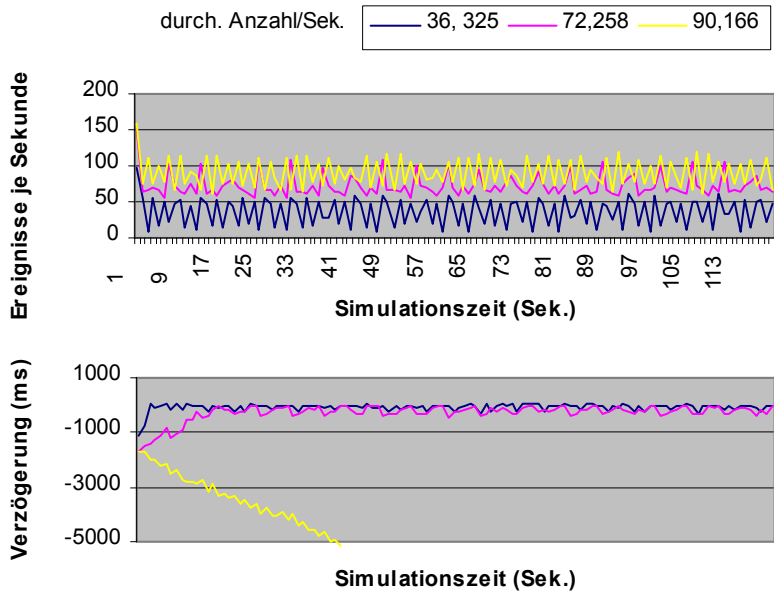

Bild 5-10: Auswirkungen der Anzahl „Ereignisse je Sekunde“ auf das Laufzeitverhalten der Simulation

Die vorherigen Ausführungen weisen darauf hin, dass die Ereignisse die Laufzeit stark beeinflussen. Obschon die angegebenen Werte eng mit der Rechenleistung des Testrechners verbunden sind, wird deutlich, dass vor der Entwicklung interaktiver Simulationen zu prüfen ist, welche Leistung ein System zur Simulation erfüllen muss. Auf der anderen Seite bekräftigt die begrenzte Anzahl, der maximal je Sekunde eintretenden Ereignisse, die Auswahl einer ereignisgesteuerten Methode. Ein Ansatz mit einem fixen Zeitinkrement erfordert mit einem Intervall von 10 Millisekunden 800 Ereignisse[3].

Die Zahl der Berechnungen, die je Zeiteinheit auszuführen sind, ergibt sich bei der Methode mit festgewählten Zeitabschnitten unmittelbar aus der Anzahl der Systemkomponenten, die ihren Zustand verändern können und der gewählten Zeitspanne. Im Folgenden wird darauf eingegangen, welche Parameter die Ereignisanzahl im entwickelten Simulatorkern bestimmen.

In Abschnitt 4.2 wurden Simulationsprozesse erläutert, die Veränderungen der Zustände auslösen können. Die Prozesse zur Modellierung eines Antriebes vari-

[3] Die Anzahl ergibt sich aus sechs Shuttles und zwei Weichen in der Simulation, die bei diesem Interval 100 mal je Sekunde aktualisiert werden.

abler Streckenteile bzw. zur Repräsentation der Arbeitsstationen reagieren zumeist auf Anweisungen des zugeordneten Prozessinterface. Die Schnittstellen zwischen dem Kern und den Steuerungsobjekten reagieren auf Nachrichten von den Kontrollelementen. Ein aktives Auslösen der Ereignisse bewirkt nur der Shuttleprozess. Im Bild 5-11 ist der Zusammenhang der Anzahl der Ereignisse in Bezug zu den Shuttles im Simulationsmodell aufgezeigt. Damit in der Simulation die angegebenen Shuttles eingesetzt werden konnten, wurde das Streckenlayout entsprechend verlängert. Es wird deutlich, dass die Menge der Shuttles im Modell die Anzahl der Ereignisse beeinflusst.

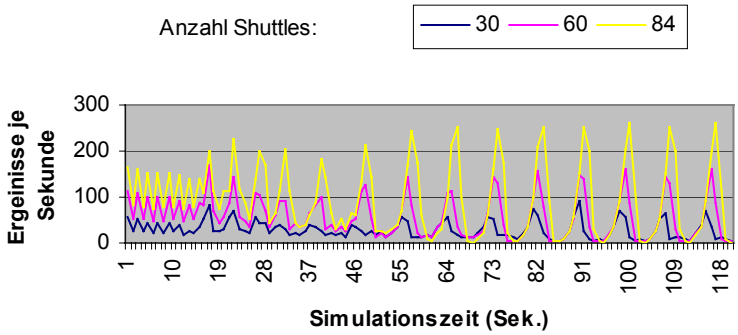

Bild 5-11: Ansteigen der zu bearbeitenden Ereignisse in Abhängigkeit zu der Anzahl der Shuttles

Aus den bisherigen Ausführungen lässt sich folgern, dass die Anzahl der Shuttles bestimmt, ob eine Simulation mindestens in Realzeit durchgeführt werden kann. Zumal eine kritische Grenze mit 90 Ereignissen je Sekunde erreicht wird, kann davon ausgegangen werden, dass die Grenze in der eingesetzten Testumgebung zwischen 60 und 84 Shuttles liegt. Diese Vermutung wird durch die Abbildung Bild 5-12 widerlegt. Die dargestellten Kurven zeigen einen Verlauf mit jeweils 84 Shuttles, aber unterschiedlichen Abständen zwischen diesen. Da die eingestellten Abstände sich während des Simulationslaufes ändern können, kann nur ein initialer Wert angegeben werden. Des Weiteren sind die Shuttles auf unterschiedliche Streckenabschnitte verteilt, so dass der Abstand sich nur auf aufeinanderfolgende Shuttles bezieht. Die Einstellungen genügen jedoch, um die qualitative Aussage: *Je mehr Abstand zwischen den Shuttles, desto geringer ist die Laufzeit*, zu verdeutlichen.

Die angegebenen Entfernungen ergeben sich aus der Positionierung der Punkte zur Repräsentation der Shuttles in der Modellwelt (vgl. Abschnitt 3.3.1). Die bisherigen Testläufe sahen in Schritten von 0.5 m aufeinanderfolgende Shuttles vor. Unter Berücksichtigung der Shuttleplatte (0.2 m) ergibt sich ein Abstand von 0.3 m. Um eine Testkonfiguration zu erzeugen, die eine Entfernung von 4.8

Metern[4] zwischen den Shuttles erlaubt, wurde der Abstand der Positionierungspunkte um den Faktor 10 erweitert. Auffällig in Bild 5-12 ist, dass der Verlauf der blauen Kurve in den ersten 20 Sekunden stark oszillierend ist. In dieser Phase fahren die Shuttles dicht hintereinander. In einem Zeitintervall von 0.6 Sekunden[5] muss jeweils eine neue freie Fahrstrecke berechnet werden (Abschnitt 4.2.1). Da die Ereignisse innerhalb einer Sekunde zusammengefasst werden, schwankt der Kurvenverlauf zwischen einem Wert von ca. 80 (Ereignis nach 0.6 Sekunden) und ca. 160 (Ereignis nach 1.2 und 1.8 Sekunden)[6]. Im weiteren Verlauf schwingt die Kurve zunehmend zwischen dem Höchstwert 250 und 0. Die Ursache dafür ist, dass die Fahrzeuge die Weichen erreicht haben und anhalten, bis das vorderste Shuttle von der Steuerung das Signal zur Weiterfahrt erhält. Startet dieses Shuttle, so folgen alle wartenden, bis das nächste an der Weiche angehalten wird.

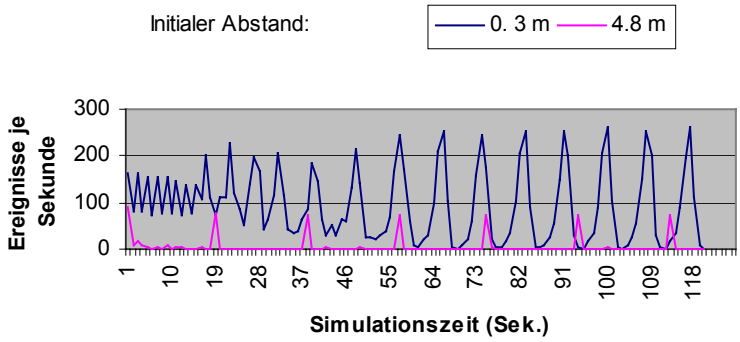

Bild 5-12: Einfluss der Abstände zwischen den Shuttles auf die Anzahl der Ereignisse

Die rote Kurve in der Abbildung 5-12 zeigt in einem Intervall von 20 Sekunden jeweils einen Spitzenwert. Dieser Wert ergibt sich aus dem Zwischenraum von 4.8 Metern zwischen den Shuttles. Da die Entfernung für die meisten Shuttles identisch ist, steigt die Anzahl der Ereignisse zu diesen Zeitpunkten an.

---

[4] Der Abstandswert ergibt sich entsprechend aus einer Entfernung der Punkte von 5 m und der Länge der Shuttleplatte von 0.2 m.

[5] Ein Shuttle fährt bei der gegebenen Konfiguration 0,25 m/Sek.. Da das vorausfahrende Shuttle 0,3 m entfernt ist und ein Mindestabstand von 0.15 m gewährleistet sein muss, wird nach 0.6 Sekunden geprüft, ob es weiterfahren kann.

[6] Das gegebene Beispiel soll aufzeigen, wie sich die angegebenen Werte ergeben. Zur Verdeutlichung sind die ersten drei Ereignisse eines Shuttleprozesses aufgeführt.