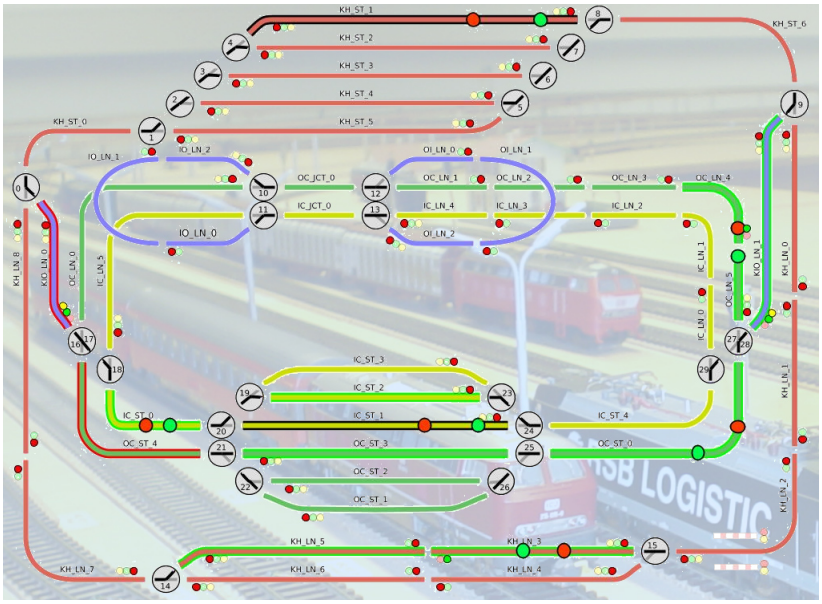
Modellbasierte Umgebungssimulation

für verteilte Echtzeitsysteme mit flexiblem Schnittstellenkonzept

Fallstudie einer Bahn-Anlage

Studienarbeit von Christian Motika



Christian-Albrechts-Universität zu Kiel
Institut für Informatik
Lehrstuhl für Echtzeitsysteme und Eingebettete Systeme

4. Oktober 2007

Betreuer: Dipl.-Inf. Hauke Fuhrmann

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Kiel,

_____

Abstract

As a part of the model railway practical laboratory course [7] in the summer term 2007 of the Embedded Systems Group at the Christian-Albrechts University at Kiel, Germany, a simulation for the model railway of the Embedded Systems Group had to be implemented using the SCADE synchronous modeling language that was used in the entire project. The main task of this course was the model-based implementation of a controller for the model railway.

The simulation of the model railway should facilitate the implementation and the debugging process of the model railway controller. A linked graphical visualization for the simulation illustrates the state of the model railway simulation and gives necessary (debugging) information to the programmer of the railway controller. Only the services of the model railway hardware are simulated, including the electrical peripheral controllers along with its connected semaphores, points, track segments and reed contacts.

To extend the possibilities of the test facility, another main task in this study thesis was the implementation of new interfaces to other programming languages. The simulation interface to the C language represents an important example of these efforts. There already exists a C library for the model railway. The interface was implemented following this existing C library so that it is possible for a controller to connect to both, the model railway itself and its simulation.

Furthermore a TCP/IP interface program for the model railway and the simulation was designed that allows the development of a model railway controller in various other languages supporting TCP/IP. A JAVA and a PHP class for that purpose come along with this work and with additional sample controllers that show how to use these interfaces.

Key words model railway, simulation, interface, C, TCP/IP, JAVA, PHP, model-based design

Zusammenfassung

Als Teil des Modellbahnpraktikums [7] im Sommersemester 2007 der Arbeitsgruppe für Echtzeitsysteme und Eingebettete Systeme an der Christian-Albrechts-Universität zu Kiel wurde modellbasiert eine Simulation für die Modellbahnanlage mit der SCADE Suite entwickelt, die auch für das gesamte Projekt als Entwicklungswerkzeug diente. Die Hauptaufgabe des Praktikums bestand darin, einen Controller für die Bahnanlage modellbasiert zu entwerfen und zu implementieren.

Die Simulation der Modellbahnanlage sollte den Entwicklungsprozess für den Controller unterstützen und das Debugging erleichtern. Dabei verdeutlicht eine an die Simulation gekoppelte grafische Visualisierung den Zustand der Simulation und versorgt den Programmierer des Controllers so mit den entsprechenden (Debugging-) Informationen. Die Simulation beschränkt sich auf die von der Bahnhardware angebotenen Dienste; dabei wird das Verhalten der Leistungselektroniken mit den angeschlossenen Signalen, Weichen, Steckenabschnitten und den Reed-Kontakten simuliert.

Eine weitere Hauptaufgabe in dieser Arbeit war, neue Schnittstellen zu anderen Programmiersprachen zu schaffen, um die Möglichkeiten der Versuchsanlage zu erweitern. Ein wesentliches Beispiel dafür stellt eine Anbindung an die Programmiersprache C dar. Da dies bisher die vorrangig eingesetzte Sprache zur Entwicklung von Bahncontrollern war und entsprechende Schnittstellen zur Bahn selbst bereits existieren, wurde auch die Schnittstelle zur Simulation an diese angelehnt. Somit ist es für einen in C geschriebenen Controller ohne Änderungen möglich, einerseits die reale Bahnhardware aber auch die Simulation zu steuern.

Zusätzlich wurde noch eine TCP/IP-Schnittstelle entworfen, mit welcher es möglich wird, einen Bahncontroller in einer Vielzahl von Programmiersprachen zu schreiben, sofern diese TCP/IP unterstützen. Für JAVA und PHP wurden solche Klassen bereits definiert und sind zusammen mit entsprechenden Beispiel-Controllern in dieser Arbeit enthalten.

Schlüsselwörter Modellbahn, Simulation, Schnittstellen, C, TCP/IP, JAVA, PHP, Modellbasierter Entwurf

Inhaltsverzeichnis

Verzeichnis der Abkürzungen

API	Application Programming Interface
EBNF	Extended Backus-Naur Form
CAN	Controller Area Network
FIFO	first in, first out
GCC	GNU Compiler Collection (ursprünglich GNU C Compiler)
GNU	Gnu's not Unix
GUI	Graphical User Interface
IDE	Integrated Development Environment
IP	Internet Protocol
KIEL	Kiel Integrated Environment for Layout
KPN	Kahn Process Network
OOP	Object Oriented Programming
PHP	Pre Hypertext Processor (ursprünglich Personal Home Page)
SCADE	Safty Critical Application Development Environment
SVG	Scalable Vector Graphics
TCP	Transmission Control Protocol
TTP	Time Triggered Protocol
XML	eXtensible Markup Language

Abbildungsverzeichnis

1. Einleitung

Simulationen sind heutzutage fast alltäglich geworden und haben in so gut wie allen Bereichen des Lebens Einzug gehalten. Ob Spielsimulationen, Unternehmensplanspiele, Schaltungssimulationen, Ausbildungssimulatoren, Windkanalsimulationen oder meteorologische Simulationen zur Wettervorhersage; allen Simulationen ist gemein, dass bei ihrer Durchführung, auch als Simulationsexperiment bezeichnet, nicht mit realen Objekten, sondern nur mit mehr oder weniger abstrakten Modellen dieser Objekte gearbeitet wird. Simulationen dienen oft der analytischen Betrachtung von Systemen, für die eine theoretische Betrachtung aufgrund der hohen Komplexität ausscheidet. Dies ist oft bei Systemen mit dynamischem Verhalten der Fall, da hier die Anzahl der möglichen verschiedenen Abläufe exponentiell wächst.

Der Zugverkehr auf einer Eisenbahnanlage stellt ein Musterbeispiel eines solchen realen Systems mit dynamischem Systemverhalten dar. Hier ist die Anzahl der Gleise beschränkt, nur an bestimmten Stellen ist ein Überholen möglich, Züge können sich leicht gegenseitig blockieren, Kollisionen sollten vermieden werden. Evtl. sind noch andere Anforderungen, wie bestimmte Fahrtrichtungen, Geschwindigkeiten oder sogar genaue Zeit- und Fahrpläne zu beachten. Die Aufgabe eines Controllers auf einer solchen Anlage ist es, neben der Berücksichtigung der aufgeführten Einschränkungen und Normativen, für möglichst viele Züge einen reibungslosen Verkehrsablauf sicherzustellen. Leicht einzusehen ist, dass die Komplexität bei diesen Aufgaben schon auf kleinen Streckennetzen auch mit wenigen Zügen schnell sehr hoch wird.

1.1. Die Modellbahnanlage des Instituts für Informatik

Die Modellbahnanlage des Instituts für Informatik [19] (s. Abbildung 1.1) an der Christian-Albrechts-Universität zu Kiel ist ein konkretes reales System mit dynamischem Verhalten, wie es eingangs beschrieben wurde. In diesem komplexen Labor sind mehr als 200 Sensoren und Aktuatoren verbaut, welche über mehrere alternative verteilte Systeme unterhalb der Bahnanlage von verschiedenen Knoten aus angesteuert werden können. Die einzelnen Knoten sind über Feldbussysteme miteinander verbunden. Zur Auswahl stehen hierbei CAN [1], TTP [22] und Ethernet [15].

Die Bahnanlage besteht aus dem Streckennetz [20], das in Abbildung 1.2 schematisch dargestellt wird, und der Kommunikationshardware, die für die Ansteuerung zuständig ist. Jeder zusammenhängende Gleisabschnitt, der im Streckenplan optisch von anderen Abschnitten getrennt ist, kann einzeln angesteuert werden und hat einen eigenen Namen (z.B. KH_ST_5). Die meisten dieser Abschnitte verfügen über zwei

Abbildung 1.1.: Modellbahnanlage der Informatik

sog. *Reed-Kontakte*; einen am Anfang und einen am Ende eines solchen Gleisabschnitts und in der Abbildung als kleiner blauer Balken gekennzeichnet. Diese Kontakte werden durch das Überfahren eines Zuges ausgelöst, dienen damit als Sensoren und können vom Bahncontroller abgefragt werden. Außerdem wird von der Bahnhardware noch zusätzlich eine Geschwindigkeit und die Richtung des auslösenden Zuges ermittelt. Die Leistungselektroniken, an die jeder Gleisabschnitt angeschlossen ist, können über Messungen zum Spannungsabfall zusätzlich feststellen, ob sich eine Lok auf dem Streckenteil befindet und ob ein Kurzschluss vorliegt. Die 29 Weichen sind durchnummeriert und lassen sich ebenfalls über die Peripherie ansteuern, wie auch die zu jedem Gleisabschnitt gehörenden Signale. An einer Stelle im System befindet sich darüber hinaus noch ein steuerbarer Bahnübergang.

Grundsätzlich lässt sich das Streckennetz der Modellbahn in die folgenden drei miteinander verbundenen Kreise unterteilen:

1. Kicking Horse Pass
2. Outer Circle
3. Inner Circle

Der Kicking Horse Pass ist in der Abbildung rot eingezeichnet. Der Outer Circle hat die Farbe grün und mit orange wurde der Inner Circle markiert. Übergänge von einem Kreis in den anderen sind z.B. die mittig angeordneten blauen Wendeschleifen

oder die ebenfalls blau gekennzeichneten Ausfahrten aus dem Kicking Horse Bahnhof in den Outer Circle bzw. Inner Circle und zurück. Jeder der drei Kreise verfügt über einen Bahnhof. Der Inner Circle und der Outer Circle kann bzw. darf jeweils nur in eine Richtung befahren werden, und zwar gegen und mit dem Uhrzeigersinn; auf dem Kicking Horse Pass ist es als Besonderheit erlaubt, in beide Richtungen zu fahren und es existiert noch eine zusätzliche Ausweichmöglichkeit. Die Standardhauptfahrtrichtung im Kicking Horse Pass ist ebenfalls im Uhrzeigersinn.

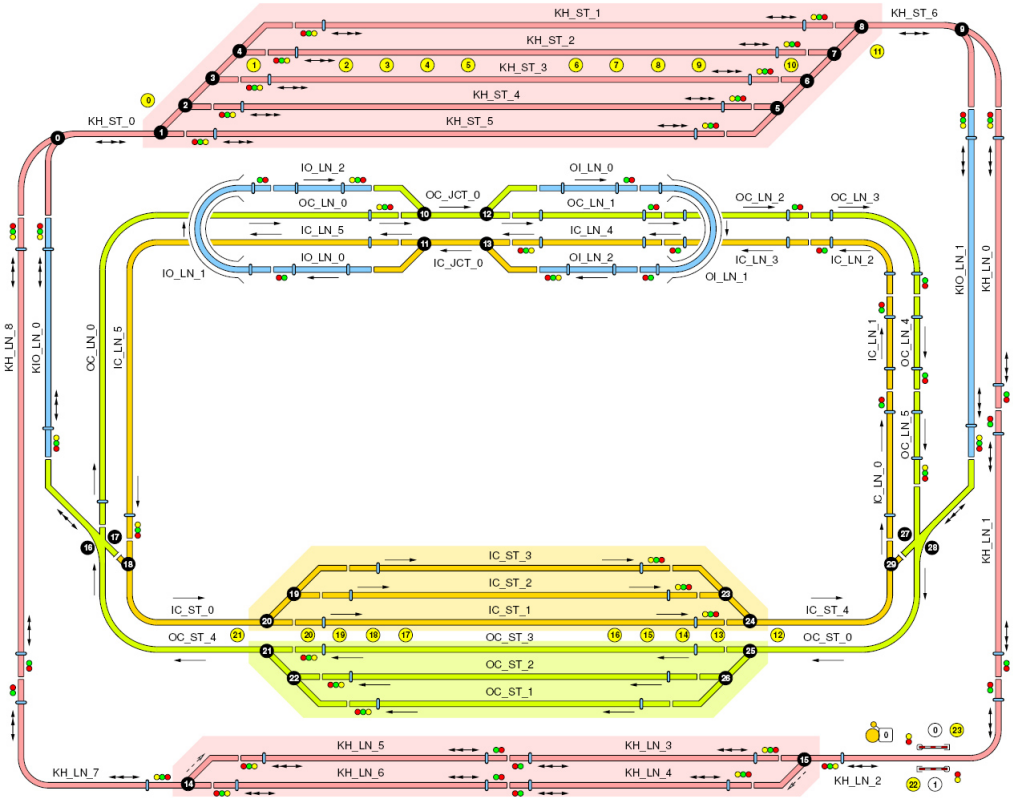


Abbildung 1.2.: Das vereinfachte Streckenlayout

Der genaue Aufbau der Bahnanlage und eine detaillierte Beschreibung der Komponenten kann auf der Homepage der Modellbahn [19] nachgelesen oder der Diplomarbeit von Stephan Hörmann [20] entnommen werden.

1.2. Aufgabenstellung

Im Rahmen des Modellbahnpraktikums 2007 [7] sollte ein verteilter Bahncontroller entwickelt werden, der auf den neun zur Verfügung stehenden TTP-Knoten läuft und die Bahnanlage so ansteuert, dass möglichst viele Züge gleichzeitig fahren und dabei individuelle Fahrpläne abarbeiten.

Um das Verhalten solch eines Controllers schon bei der Entwicklung analysieren zu können und so möglichen, zum Teil logischen Fehlern frühzeitig auf die Spur zu kommen, ist der Einsatz einer Simulation der Bahnanlage sinnvoll, wie dies der einführende Abschnitt nahe legt.

Ziel dieser Arbeit war die modellbasierte Entwicklung einer solchen Simulation für die Modellbahnanlage der Informatik. Dabei sollte die Schnittstelle der Simulation möglichst ähnlich zum tatsächlichen Interface der realen Anlage sein. Neben der dabei unterstützten Programmiersprache C war das Ziel, die Modellbahn und die Simulation auch anderen Programmiersprachen zugänglich zu machen. So besteht nun durch ein entsprechendes Interfaceprogramm die Möglichkeit, beides z.B. über einen JAVA-Controller anzusteuern.

1.3. Gliederung der Arbeit

Diese Ausarbeitung gliedert sich in fünf Kapitel, welche den modellbasierten Ansatz der Simulationsentwicklung über die grafische Visualisierung bis hin zu den bereitgestellten Controller-Schnittstellen abdecken.

Im Kapitel 2 wird zunächst ein Vergleich der eingesetzten Entwicklungswerkzeuge und Sprachen angestellt. Anschließend folgt die Vorstellung der verwendeten Entwicklungsumgebung für die grafische Modellierung. Dem schließt sich eine Übersicht über die reale Modellbahn an, welche es zu simulieren gilt. Im weiteren Verlauf wird dann genauer auf das erstellte Modell der Anlage eingegangen. Den Schluss des Kapitels bilden zwei praktische Abschnitte über die Generierung von C-Code aus dem Modell und über den internen Simulator der Entwicklungsumgebung.

Das ModelGUI-Projekt, welches zur Visualisierung eines Simulationslaufs eingesetzt wird, präsentiert das Kapitel 3. Hier folgen anschließend noch detaillierte Betrachtungen zur Anbindung an das erstellte Simulationsmodell.

Das Kapitel 4 beschreibt zunächst die Aufgaben einer Schnittstelle für Controller der Bahnanlage. Dann wird die konkrete C-Implementierung vorgestellt und im weiteren Verlauf auf das TCP/IP-Schnittstellenprogramm eingegangen. Gegen Ende des Kapitels finden sich praktische Anleitungen für die Verwendung von JAVA- und PHP-Controllern.

Einige Anregungen für zukünftige Erweiterungen der Simulation und ein abschließendes Fazit der Arbeit gibt das letzte Kapitel 5.

Der Anhang schließt mit einer Kurzreferenz zur Implementierung entsprechender Bahncontroller sowie mit der Dokumentation zur erstellten Software an.

2. Modellbasierte Entwicklung in SCADE

2.1. Motivation

Die Entwicklung der Simulation war u.A. ein Bestandteil des Modellbahnpraktikums im Sommersemester 2007 am Lehrstuhl für Echtzeitsysteme und Eingebettete Systeme. Da das Praktikum auf die Vorlesung zum modellbasierten Entwurf aufbaute und hier bereits die Grundlagen für synchrone Sprachen und grafische Modellierung vermittelt wurden, lag es nahe, für den Entwicklungsprozess die SCADE Suite [11] einzusetzen. Dabei handelt es sich um ein Werkzeug, welches sowohl eine grafische Modellierung von Datenfluss als auch Statecharts unterstützt. Es wurden damit sowohl der Bahncontroller als auch die Simulation realisiert.

Als Vorzüge einer realistischen Simulation während des Entwicklungsprozesses und zu Demonstrationszwecken sind die folgenden zu nennen:

- Unabhängigkeit von Zeit und Ort (Modellbahnhardware)
- Einsparung von Kompilierzeit
- Einsparung des *Flash*-Prozesses auf die eingebetteten Systeme
- Ermöglichung verschiedener (Simulations-)Ausführungen zur selben Zeit
- Schutz der echten Bahnhardware
- Exzessives Testen und Debuggen
- Leichtes Simulieren von Verhalten in Fehlerzuständen
- Minimierung von Softwarefehlern in der Endversion
- Vereinfachte Demonstrationen (z.B. von Strategien des Controllers)

Gerade für die Simulation bietet sich die modellbasierte Entwicklung schon deshalb an, weil das fertige Modell auf diese Weises später leicht verändert oder erweitert werden kann.

Einführend wird sich dieses Kapitel zunächst einem Vergleich einiger Entwicklungswerkzeuge und Sprachen in Bezug auf die Simulation widmen. Im darauf folgenden Abschnitt wird dann die bei der Entwicklung hauptsächlich verwendete SCADE Suite (Version 6) vorgestellt. Dem schließt sich ein Überblick über die reale Modellbahn-

anlage an, die es zu simulieren galt. Darauf folgt eine detailliertere Vorstellung des Simulationsmodells und der SCADE-Operatoren aus denen es besteht.

2.2. Vergleich der Entwicklungswerkzeuge

Der Zugverkehr auf der Modellbahnanlage ist, wie eingangs erwähnt, ein Musterbeispiel für ein dynamisches und reaktives System. Dies gilt nicht nur für einen Controller, der auf Kontaktmeldungen, den allgemeinen Zustandsstatus und Benutzereingaben reagieren muss, sondern auch für eine Simulation der Anlage. Diese muss ständig auf die Kommandos des Controllers reagieren, dabei die Modellbahnanlage möglichst realistisch nachahmen und schließlich entsprechende Zustandsinformationen liefern.

Vor der Entwicklung einer solchen Simulation stellt sich die Frage, welche Sprache dafür am besten geeignet ist. Insgesamt müssen bei der Auswahl vor allem die folgenden Kriterien in Betracht gezogen werden:

Robustheit : Die Anzahl der durch den Programmierer verursachten, unbeabsichtigten Programmierfehler sollte bereits durch die Möglichkeiten der Sprache beschränkt und gering gehalten werden. Insbesondere ist eine aktive Unterstützung des Programmierers in Bezug auf das Finden und Eingrenzen solcher Fehler erwünscht.

Lesbarkeit : Der Programmcode sollte von sich aus bereits mit möglichst wenigen Kommentaren auskommen und ansprechend, übersichtlich aber vor allem eindeutig sein. Dies erhöht insbesondere auch die Wartbarkeit des Programms, was bei einer Simulation, die ständig an das zu simulierende reale System angepasst werden können muss, äußerst wichtig ist.

Flexibilität : Durchzuführende Operationen müssen sich in der Sprache leicht ausdrücken lassen. Dazu sollte die Sprache bereits in ihren Grundzügen entsprechende Konstrukte bereitstellen. Für die Simulation sind das vor allem Parallelität mit Synchronizität z.B. aufgrund der Modellierung vieler parallel existierender Gleiselemente, die Reaktion auf Ereignisse und die Kapselung von Funktionalität zur Komplexitätsbeherrschung.

Verwendbarkeit : Die Sprache sollte weit verbreitet und leicht zu verstehen sein, damit sich die Weiterentwicklung und Wartung nicht wegen exklusiver Kenntnisse auf einen kleinen dazu fähigen Personenkreis beschränkt.

Portierbarkeit : Eine spätere Bindung der Simulation an bestimmte Hard- oder Software aufgrund der verwendeten Sprache ist unerwünscht.

Für einen Einsatz der Sprache C zur Implementierung einer solchen Simulation sprechen dabei, dass die Sprache sehr weit verbreitet ist und eine Vielzahl von Tools zur Code-Analyse (z.B. *style checker*) existieren. Außerdem ist der Einsatz

von C sehr effizient, da C relativ hardwarenah ist. Letzteres ist aber an dieser Stelle auch ein Problem der Sprache, denn das Verhalten von C-Programmen kann manchmal plattformabhängig sein, z.B. wenn Betriebssystemfunktionen verwendet werden. Allen voran ist hierbei die für die Simulation wichtige nebenläufige Abarbeitung mehrerer Aufgaben zu nennen, die in C z.B. mit Threads oder Prozessen realisiert werden kann. Leider ist das Scheduling in diesem Fall vom Betriebssystem abhängig, was in einer portierbaren Echtzeit-Simulation nicht tolerierbar ist. Dadurch könnten schlimmstenfalls einige Züge (bzw. Gleisabschnitte) zeitlich vor anderen simuliert und so kein realistisches Modellverhalten mehr gewährleistet werden. Die Hardwarenähe der Sprache C bereitet auch hinsichtlich der Lesbarkeit Probleme, denn sie bietet kaum Möglichkeiten zur Komplexitätsbeherrschung großer Programme, wie dies höhere und abstraktere Sprachen anbieten. Es fehlt z.B. aufgrund der flachen Struktur an Mitteln, Hierarchie einfach auszudrücken oder an ausgefeilten Fehlerbehandlungsmechanismen. Dazu kommt, dass die Robustheit von C-Code oft stark unter den sog. *traps und (lexical) pitfalls* [14], also sprachbedingter, unbeabsichtigter Programmierfehler leidet. Als Beispiele hierfür wären zu nennen:

- Ungenügende Klammerung von Ausdrücken
- Zuweisung satt Vergleich
- Vermischung von Kommentar und Dereferenzierung
- Verschachtelte Headerdateien und damit das Einbinden nicht beabsichtigten Codes
- Fehlende Block-Klammern
- Kein `Else`-Zweig oder kein `default` bei `if`- und `switch`-Anweisungen
- Ungenügende Verwendung statischer und damit vorwiegend (standardmäßig) externer Funktionen
- Nicht initialisierte Variablen
- Verschachtelte `if`-Anweisungen und ein `else`-Zweig
- Schwache Typisierung (`enum` vs. `int`)

Viele dieser Fehler können durch entsprechende Compilermeldungen oder weitere Analysewerkzeuge vermieden oder nachträglich beseitigt werden. Grundsätzlich trifft die Sprache selbst aber keine Vorkehrungen zu ihrer Vermeidung.

JAVA bietet im Gegensatz zu C, durch die Verwendung von Klassen im Sinne der objektorientierten Programmierung (OOP), bereits deutlich mehr Möglichkeiten einer Hierarchiebildung. Auch Fehlerbehandlungsmechanismen existieren hier. Durch die weite Verbreitung von C und die syntaktische Ähnlichkeit beider Sprachen ist

auch JAVA in Bezug auf die Verwendbarkeit positiv zu bewerten. Allerdings ergeben sich auch hier Probleme bei sprachlichen Konstrukten (s.o.) und daneben trübt der Nichtdeterminismus des *Garbagecollector*s hier das Bild für eine Verwendung als einsetzbare Programmiersprache in Bezug auf ein deterministisches Zeitverhalten. Vorallem aber lässt sich die geforderte Synchronizität und damit auch eine deterministische Nebenläufigkeit, ebenfalls wie bei C, nicht leicht ausdrücken.

Gerade was den letztgenannten Punkt betrifft liegt es daher nahe, eine Sprache zu wählen, in der dieses Konzept von Grund auf existiert. In den 1980er Jahren haben sich zur Beschreibung von dynamischen, reaktiven Systemen mit dem Statechart-Dialekt *SyncCharts* als Erweiterung von *Mealy machines* und kontrollfluss- sowie datenflussorientierter Sprachen die Familie der *synchronen Sprachen* entwickelt.

Die Sprache *Lustre* [21] gehört zu dieser Sprachfamilie. Hierbei handelt es sich um eine formal definierte, deklarative und vor allem synchrone Datenflusssprache. Sie ist elementar auf Nebenläufigkeit und Hierarchie ausgerichtet und basiert auf einem synchronen Zeitmodell[1], in welchem die Zeit in diskrete sog. *Ticks* unterteilt ist. Datenflusssprachen eignen sich besonders gut für signalverarbeitende Prozesse, wie dies bei der Modellbahn und auch ihrer Simulation der Fall ist.

Das in dieser Arbeit hauptsächlich eingesetzte Modellierungswerkzeug SCADE [11] baute ursprünglich auf der Sprache *Lustre* auf. Inzwischen wurde daraus die um Kontrollflussstrukturen (*Statemachines*) erweiterte, textuelle SCADE-Sprache. Die SCADE Suite bietet dem Programmierer eine Oberfläche an, auf der Modelle grafisch erstellt werden können. Die Flexibilität ist dadurch gewährleistet, dass neben der grafischen Repräsentation von Modulen[2] auch eine textuelle erlaubt ist und zudem importierte C-Operatoren genutzt werden können. Die Operatoren lassen eine Schachtelung, also eine Verwendung anderer Operatoren und damit eine übersichtliche Hierarchiebildung, im Sinne der Komplexitätsbeherrschung zu. Die grafische Notation ist ansprechend und leicht interpretierbar. Dies sorgt für eine gute Lesbarkeit der Modelle. Es ist jedoch etwas aufwändiger diese wirklich übersichtlich zu gestalten, denn hierfür gibt es bisher kein integriertes *automatisches Layouting*, an welchem z.B. an der Universität Kiel geforscht wird (*KIEL*-Projekt [4]). Mittels Strukturierung ist aber eine gute Wartbarkeit des Simulationsmodells gegeben.

Im Punkte der Verwendbarkeit mangelt es derzeit noch an einem höheren Verbreitungsgrad und damit an fachkundigen Programmierern. Für den Einsatz am Lehrstuhl ist dies jedoch unerheblich, da sich hier die Forschung u.A. auf diesen Bereich konzentriert.

Neben der Synchronizität setzt sich SCADE aber vor allem im Punkt der Robustheit von C und JAVA ab. Hierbei unterstützt das Entwicklungswerkzeug den

[1] Das diskrete Zeitmodell ist auch unter dem Begriff der *multiform notion of time* bekannt. Dabei *vergeht* die physikalische Zeit als ein extern auftretendes Ereignis. Dieses Konzept ist essenziell für die deterministische Reproduzierbarkeit des Systemverhaltens.

[2] Diese werden in SCADE als *Operatoren* bezeichnet.

Programmierer z.B. mit sog. *quick checks*[3], mit dessen Hilfe einzelne Operatoren überprüft werden können. Die Typisierung ist wesentlich härter als dies bei C der Fall ist. Es existieren keine `if`-Konstrukte ohne einen `else`-Zweig. Ungenügende Klammerung von Ausdrücken wird von vornherein nicht akzeptiert. Die Sichtbarkeit von (lokalen) Variablen ist eindeutig und beschränkt sich nur auf den aktuellen Operator. Nicht initialisierte Datenströme werden ebenso abgelehnt. Neben den erwähnten *quick checks* sind noch weitere Werkzeuge zur statischen Code-Analyse und formalen Verifikation integriert. SCADE kann aus korrekten Modellen automatisch C-Code generieren. Damit wird der vorletzte Punkt der Verwendbarkeit wieder erfüllt, denn dies war eine Stärke der Sprache C. Auch die Effizienz von C-Code kann damit für SCADE ausgenutzt werden, obgleich bei generiertem Code möglicher Overhead entstehen kann, welcher jedoch durch verschiedene Optimierungslevel minimierbar ist. Aufgrund der automatischen Generierung des Quellcodes sind sprachbedingte, unbeabsichtigte Programmierfehler ausgeschlossen. Nebenläufigkeit wird durch die *multiform notion of time* zur Gleichzeitigkeit der Berechnung innerhalb eines Ausführungsticks, und es werden somit keine Threads oder Prozesse benötigt. Dabei existiert, von außen betrachtet, nur noch eine einzige C-Funktion die aufgerufen wird, um einen solchen Tick auszuführen. Der Code ist jedoch für sich nur schwer lesbar oder wartbar. Daher kommt für solche Programmkonstruktionen einzig eine automatische Codegenerierung in Frage. Es werden weiterhin nur plattformunabhängige[4] Sprachkonstrukte im generierten Code verwendet. Damit bleibt der C-Code des Modells portierbar. Die Entwicklungsumgebung selbst ist dies jedoch nicht; sie benötigt ein Windows-System zu ihrer Ausführung.

Neben SCADE hätte sich für die Implementierung der Simulation auch noch eine andere grafische Modellierungssprache angeboten, nämlich Matlab/Simulink [5] zur Modellierung in Verbindung mit dem Real-Time Workshop [8] zur Codegenerierung. Dabei gehört diese Werkzeugkette, verglichen mit SCADE, derzeit vermutlich sogar zu der verbreiteteren. Vorzüge von Simulink sind u.a., dass neben der diskreten Zeit auch ein kontinuierliches Zeitmodell (durch Differentialgleichungen) angeboten wird. Gerade für Simulationen der realen Welt stellt dies einen enormen Vorteil dar.

Zum einen wurde Simulink aber bereits in früheren Praktika eingesetzt, zum anderen sollten in dieser Arbeit gerade die Vorzüge der neuen SCADE-Version gegenüber Matlab/Simulink genutzt und untersucht werden. Dazu zählt eine wesentlich integriertere Entwicklungsumgebung, in welcher neben dem qualifizierten Codegenerator auch eine automatische Dokumentationsgenerierung enthalten ist, vor allem aber eine Kombination aus Datenfluss und Kontrollfluss ermöglicht wird. Zudem unterscheiden sich auch die beiden enthaltenen Simulatoren grundsätzlich: Bei Simulink werden alle Simulationsdaten vorab berechnet und anschließend grafisch dargestellt; mit SCADE ist es möglich die Simulation von Modellen Schritt für Schritt durchzuführen und

[3] Mit *quick checks* ist es möglich, Operatoren auf ihre Zyklenfreiheit hin zu überprüfen. Weiterhin können z.B. syntaktische Fehler in Formeln oder Typ-Inkonsistenzen aufgedeckt werden.

[4] Dies schließt die eigenen importierten C-Operatoren aus.

dabei den (auch detaillierten) Zustand des gesamten Modells zu beobachten und zu analysieren.

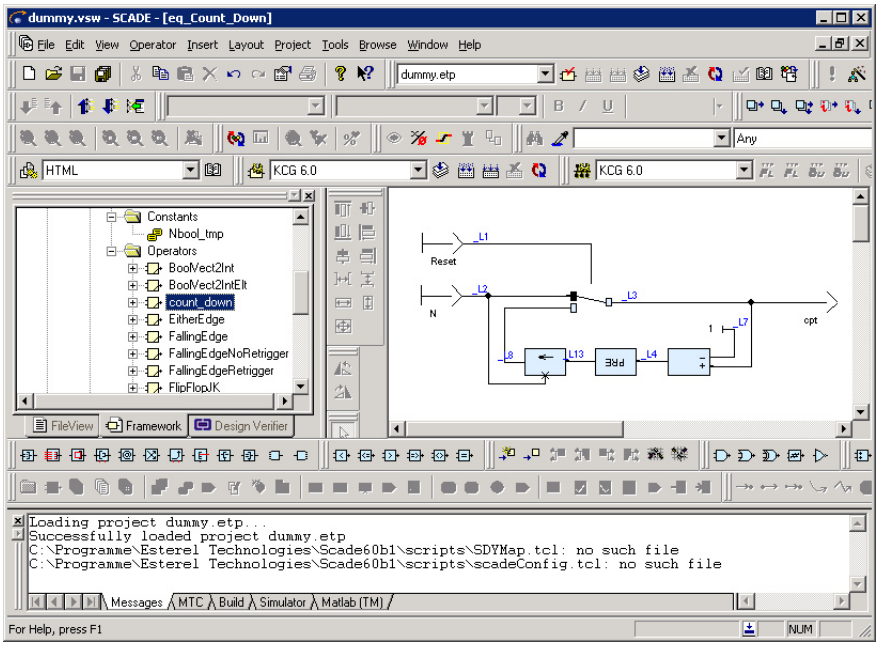


Abbildung 2.1.: Die SCADE-Entwicklungsumgebung

2.3. Die SCADE Suite

SCADE [5] ist ein Entwicklungswerkzeug für sicherheitskritische Systeme und basiert auf der grafischen, formalen, synchronen und Datenfluss orientierten SCADE-Sprache. Es bietet eine integrierte Entwicklungsumgebung (IDE) an, in der sowohl grafische als auch textuelle Modelle erstellt und mit welcher aus diesen Modellen später C- oder ADA-Code generiert werden kann. Als Entwicklungsumgebung (s. Abbildung 2.1) in diesem Projekt wurde die SCADE 6 Suite zunächst in der Prototypen Version, später in der Beta 1 Version und schließlich mit der FCSa Version verwendet. Das Final Release, welches für den Herbst 2007 erwartet wird, stand während der Arbeit noch nicht zur Verfügung. In früheren Versionen vorwiegend auf Datenfluss ausgerichtet, ermöglicht SCADE ab der Versionsnummer 6 die Kombi-

[5] SCADE steht für **S**afty **C**ritical **A**pplication **D**evelopment **E**nvironment und findet, wie es der Name bereits vorweg nimmt, seinen Einsatz vor allem bei der Entwicklung von sicherheitskritischen Systemen in Bereichen wie z.B. der Luftfahrtindustrie bei Flugkontrollsystemen, Autopiloten, Turbinensteuerung, Bremssystemen oder dem Kraftwerksbau bei der Reaktorüberwachung, Energieregelung und Alarmsteuerung.

nation aus Datenflusssprache und Kontrollflussstrukturen (*Statemachines*[6]), die sich in vielen Situationen als sehr hilfreich erwiesen hat. Im Entwicklungsprozess wird in SCADE zunächst grafisch mit Datenflussdiagrammen und *Statemachines* oder textuell ein Modell entworfen. Anschließend wandelt SCADE dieses Modell mit der Generierung von C-Quelltext in einen kompilierbaren Programmcode um. Dieser kann schließlich mit gängigen C-Compilern verwendet werden.

Während der Entwicklung des SCADE basierten Controllers im Modellbahnpraktikum importierte das Simulationsprojekt ein von dem SCADE-Zusatzmodul *SCADE Link* erstelltes Simulationsmodell vom Controller und simulierte damit das gesamte System inklusive des verwendeten TTP-Feldbusses [22]. Das jetzige SCADE-Simulationsprojekt bietet allerdings zusätzliche Schnittstellen nach außen an, welche eine Verwendung des generierten C-Codes mit einem Controller unabhängig von der SCADE-Entwicklungsumgebung zulassen.

2.3.1. Verknüpfung von Daten- und Kontrollfluss

Einer der besonderen Neuerungen von SCADE in seiner jetzigen Version, mit einer Aufhebung der Trennung von Kontroll- und Datenfluss, sei dieser folgende Abschnitt gewidmet.

Grundlegend für Datenflusssprachen ist die fundamentale Verankerung von Gleichzeitigkeit. Mehrere nebenläufige Prozesse arbeiten stets nach dem selben Schema:

```
Empfangen, Verarbeiten, Versenden, ..., Empfangen,
                Verarbeiten, Versenden
```

Dabei kommunizieren diese Prozesse nur über FIFO-Puffer miteinander; sie lesen beim Empfangen Daten aus den Puffern und beschreiben diese beim Versenden. Der Vorteil ergibt sich aus einer wesentlich besseren Vorhersagbarkeit als bei gemeinsam genutztem Speicher (*shared memeory*). Diese Prozesse lassen sich grafisch am besten in Blockdiagrammen darstellen, wobei die Blöcke den Prozessen selbst und die Verbindungslinien zwischen ihnen den Kommunikationspuffern entsprechen.

In synchronen Datenflusssprachen ist es ebenfalls üblich, Blockdiagramme zu verwenden. Ursprünglich von Edward Lee und David Messerschmitt 1987 entwickelt, waren sie eine Weiterentwicklung einer der ersten Datenflusssprachen [18], der *Kahn Process Network*s (KPN) [16], die als ein allgemeines Paradigma für parallele Programmierung gedacht war. Die Hauptidee der im Rahmen dessen durchgeführten Veränderungen war es, dass Prozesse nur noch eine ganz bestimmte Anzahl von Daten schreiben können wenn sie *feuern*.

[6] In SCADE werden damit, abweichend von der allgemeinen Bedeutung dieses Begriffs, spezielle synchrone Zustandsmaschinen beschrieben, welche sowohl Hierachie, Parallelität, zusammengesetzte Ereignisse als auch *Broadcast*-Mechanismen unterstützen.

Grundlegend hingegen für Kontrollflusssprachen sind die sie selbst beschreibenden erweiterten Zustandsautomaten (Statemachines). Sie stellten schon in ihrer ursprünglich von David Harel entwickelten Form der *Statecharts* eine Erweiterung der *Mealy machines* dar und unterstützen dabei Programmierparadigmen wie Parallelität, Hierarchie, Rundsendung und zusammengesetzte Ereignisse [13]. Die Synchronizität erhielt dann über den Statechart-Dialekt *SyncCharts* [9] Einzug. Die in SCADE implementierbaren *Statemachines* sind an letztere angelehnt.

Bei der Vermischung beider, auf der Synchronizitätshypothese (s. Abschnitt 2.2) aufbauender Konzepte werden mit Blockdiagrammen für den Datenfluss und *Statemachines* für den Kontrollfluss beide Formalismen unter bestimmten Bedingungen gleichzeitig zugelassen.

Das ganze sei an einem sehr einfach gehaltenen SCADE-Beispiel veranschaulicht, wie es in Abbildung 2.2 zu finden ist. Es zeigt die Implementierung eines *Operator*s, der wiederum in anderen Datenflussdiagrammen verwendet werden kann. Er hat als Eingänge definierte Datenströme `in_onswitch` und `in_value` und die als Ausgänge definierten `out_active` und `out_value`. Damit sich der Operator in einem Datenflussdiagramm verwenden lässt, muss sichergestellt sein, dass die Ausgänge zu jeder Zeit (d.h. innerhalb jedes Makroticks) einen definierten Wert besitzen. Zudem trifft dies auch auf lokale Variablen innerhalb ihres Gültigkeitsbereichs zu.

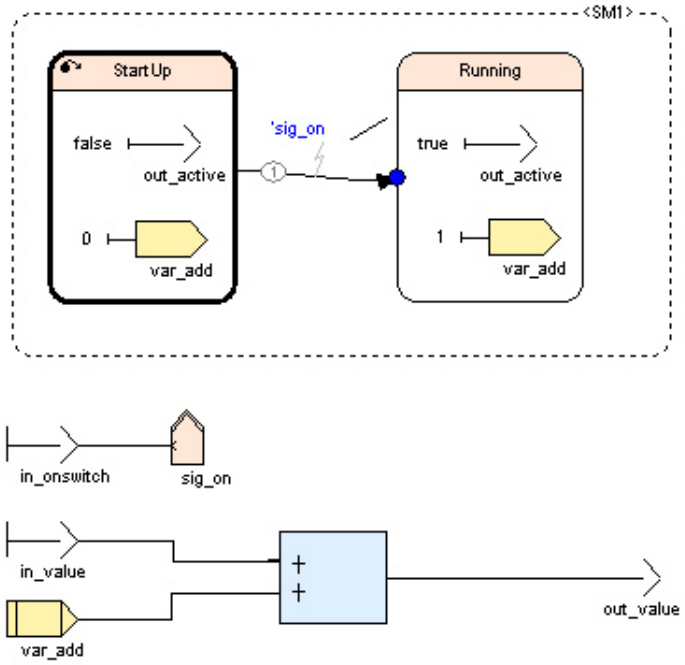


Abbildung 2.2.: Verknüpfung von Daten- und Kontrollfluss

Die dargestellte *Statemachine* hat die beiden Zustände `StartUp` und `Running`. In diesen wird der Ausgangswert von `out_active` mit einem zustandsspezifischen Wert belegt, nämlich einmal mit `false` und einmal mit `true`. Damit ist `out_active`, wie gefordert, in allen möglichen Zuständen definiert. Ähnliches gilt für die lokale Variable `var_add`. Ihr Gültigkeitsbereich erstreckt sich auf den gesamten Operator, da sie auch außerhalb der *Statemachine* verwendet wird. Sie hat ebenfalls in allen möglichen Zuständen einen (und zwar nur genau einen) definierten Wert, nämlich `0` oder `1`. Ungültig wäre eine zweite Wertzuweisung an diese Variable z.B. außerhalb der Zustandsmaschine, da dann die Eindeutigkeit in diesem Fall nicht mehr gegeben wäre. Aufgrund der Tatsache, dass `var_add` stets einen Wert besitzt, kann es für die Addition im Datenflussanteil außerhalb der *Statemachine* verwendet werden. Diese bildet zusammen mit `in_value` den Ausgabewert des Operators `out_value`. Der boolesche Datenstrom `in_onswitch` wird auf ein Signal `sig_on` abgebildet. Dieses ist immer genau dann emittiert, wenn `in_onswitch == true` gilt. In der Zustandsmaschine kann dieses Signal abgefragt werden und die Transition zwischen den beiden Zuständen auslösen.

Dieses Konzept ermöglicht es also einerseits, innerhalb von Kontrollflussdiagrammteilen erneut Datenflusselemente zu definieren, andererseits aber auch mit Datenflusselementen den Kontrollfluss zu beeinflussen.

2.3.2. Parallelität und Hierarchie in SCADE

ABRO ist ein bekanntes Beispiel für einen Zustandsautomat, in welchem die beiden Konzepte der Parallelität und der Hierarchie verdeutlicht werden können. In der Abbildung 2.3 findet sich dieser Automat als SCADE-*Statemachine*. Die initialen Zustände sind dabei durch stärkere Umrandungen hervorgehoben; finale Zustände erscheinen mit doppelter Umrandung.

Dem ABRO-Automaten liegt die folgende Spezifikation zugrunde:

- Emittiere das Signal `O` (unverzüglich) sobald die Signale `A` und `B` nacheinander oder Gleichzeit aufgetreten sind oder auftreten.
- Emittiere weder beim Start des Systems etwas noch beim Zurücksetzen.
- Setze den Automaten (unverzüglich) zurück, falls das Signal `R` auftritt.

Der ABRO-Automat setzt die obige Spezifikation wie folgt um: Die `ABRO_SM`-*Statemachine* befindet sich initial im Zustand `ABRO_state`. Dieser beinhaltet eine weitere *Statemachine* `ABO_SM` die sich anfangs im Zustand `ABO_state` befindet. Hierin finden nun die beiden parallelen *Statemachine*s `A_SM` und `B_SM` Platz. In diesen wird jeweils in einem ersten Zustand auf das Auftreten von `A` bzw. von `B` gewartet und dann in einen zweiten, finalen Zustand übergewechselt. Sind beide parallelen *Statemachine*s in ihrem finalen Zustand, dann wird die sog. *Synchro*-Transition genommen und dabei die Ausgabe `O` emittiert. Tritt zu einem beliebigen Zeitpunkt das

Signal `R` auf, so wird über eine *Strong-Abort*-Transition der Zustand `ABRO_state` neu betreten und damit der Automat zurückgesetzt, ohne dabei eine Ausgabe zu produzieren.

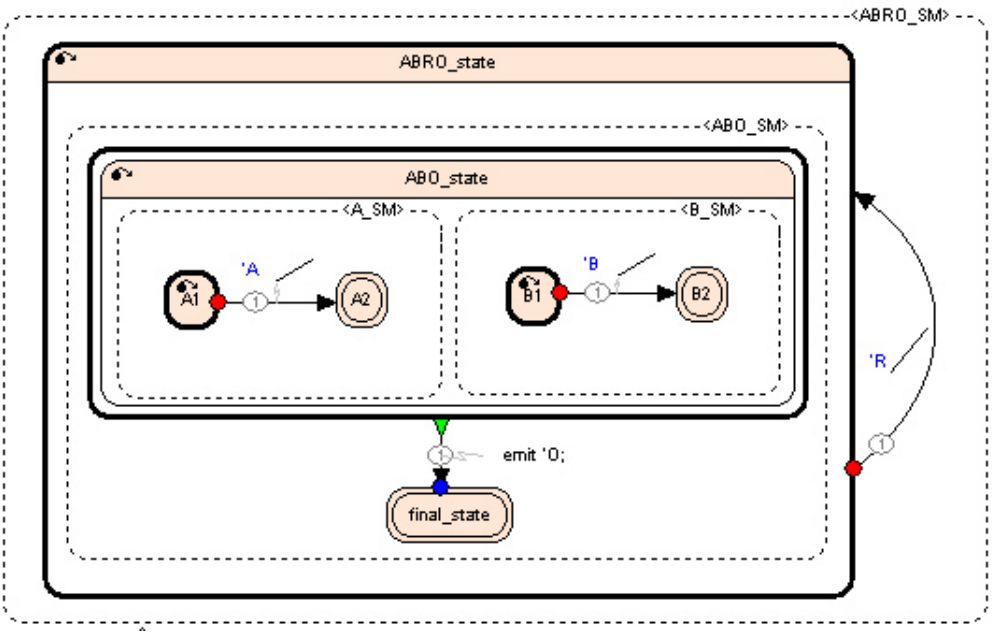


Abbildung 2.3.: ABRO-Automat in SCADE

2.4. Beschreibung des Simulationsmodells

Nachdem einführend ein Überblick über das zu simulierende reale Modellbahnsystem gegeben und nun die SCADE-Entwicklungsumgebung vorgestellt wurde, soll die Implementierung des zur Bahnanlage gehörenden Modells beschrieben werden.

Zur Simulation der Bahnhardware muss, in Anlehnung an die Schnittstellen der echten Anlage respektive der Leistungselektroniken, im Speziellen das folgenden Verhalten modelliert werden:

- Züge müssen initial gesetzt werden können
- Fahrstrom muss für jeden Gleisabschnitt festgelegt werden können
- Züge müssen von einem in den nächsten Gleisabschnitt fahren können
- Die Position der Züge innerhalb eines Gleisabschnitts muss simuliert werden
- Gleisabschnitte müssen folgendes melden können:
  - Gemessene Geschwindigkeit

  - Gemessene Richtung bei Kontaktauslösung
  - Die Kontaktauslösung selbst
  - Status, ob sich eine Lok auf ihnen befindet
  - Kurzschluss, falls vorhanden (nur reale Modellbahn)
- Weichen und Signale müssen gesetzt bzw. gestellt werden können
- Der Bahnübergang sollte ansteuerbar sein

Das SCADE-Modell der Simulation besteht dabei aus verschieden Operatoren, die gemeinsam die obigen Anforderungen erfüllen und von denen die wichtigsten im Folgenden beschrieben werden.

2.4.1. Struktur und Informationsfluss

Die grundlegende Struktur für das Modell der Bahnhardware sollte, angelehnt an die reale Anlage, genauso modular aufgebaut und erweiterbar sein wie diese. Um die Modularität sicherzustellen, wurde als kleinste Einheit ein Streckenelementsoperator definiert, aus welchem sich das gesamte Modell der Anlage zusammensetzt [6]. Der synchrone Informationsfluss wird wie folgt gewährleistet:

Für die Speicherung der Informationen sind spezielle zusammengesetzte Datentypen (Strukturen) vorgesehen, aus denen ein Array gebildet wird. Jedes Streckenelement hat dabei einen festen Index (synonym zum verwendeten Namen) und damit ein dazugehöriges Strukturelement aus diesem Array, in welchem Informationen gespeichert werden können, ohne die Daten anderer Streckenelemente zu beeinflussen. Dieses große Array über alle Streckenelemente muss alle Streckenelementsoperatoren in einem Makrotick erreichen, damit die Synchronizität gegeben bleibt. Es wird im Modell als *globale Daten* bezeichnet und enthält alle Kommandos vom Controller sowie alle Kontaktereignisse und Sensorwerte der simulierten Streckenabschnitte. Weiterhin gibt es zwischen je zwei Streckenelementen noch *lokale Daten*. Diese werden dazu benutzt, einen Zug bei der Überfahrt von einem Streckenelement in das nächste zu übergeben. Die lokalen Daten betreffen dabei nur je zwei Streckenelemente und können somit bedenkenlos einen Ausführungstick lang verzögert werden, ohne die Synchronizität der für die Modularität wichtigen globalen Daten zu gefährden. Sie müssen dies sogar, um die Zyklenfreiheit gewährleisten zu können, wie im Folgenden noch gezeigt wird.

2.4.2. Simulation

Der Operator `Simulation` ist in Abbildung 2.4 dargestellt und der oberste in der Hierarchie aller Operatoren. Seine Ein- und Ausgänge, im Folgenden als *Interface*

[6] Daneben werden noch Weichenoperatoren und später erläuterte Verzögerungsoperatoren verwendet.

eines Operators bezeichnet, stellen die Schnittstelle der Simulation nach außen dar. Zu diesem Interface zählen sowohl die `controllerCommands`, als eingehende Daten vom Controller, als auch die `controllerFeedbackSensor`-Daten, welche als ausgehende Daten dem Controller zur Verfügung gestellt werden. Um später noch ein GUI für die Visualisierung ansteuern zu können, existieren zudem noch die `displayData`-Daten, welche einer Teilmenge aller Simulationsdaten entsprechen.

In der Struktur `controllerCommands` werden alle Anweisungen des Controllers an die Bahnhardware verpackt:

- Weichenkommandos
- Fahrtrichtungskommandos
- Geschwindigkeitskommandos
- Signalkommandos
- Initiale Züge

Auch die initiale Zugbelegung, die sowohl vom Controller als auch (realistischer) mit der Simulation festgelegt werden können soll, findet hierin Platz.

In der Struktur `controllerFeedbackSensor` werden dagegen alle Sensorwerte der Bahnhardware für den Controller gekapselt:

- Kontaktmeldungen mit gemessener Richtung
- Besetztmeldungen der Gleisabschnitte
- Geschwindigkeitsmessergebnisse
- Kurzschlussmeldungen
- Meldungen über das Vorliegen neuer Messergebnisse

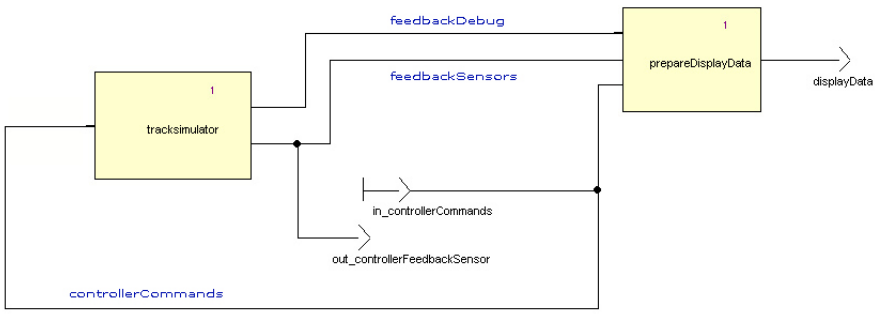


Abbildung 2.4.: Der Operator `Simulation`

Die beiden Strukturen `controllerCommands` und `controllerFeedbackSensor` sind Bestandteil der *globalen Daten* und damit an jedem Streckenelement des Modells zu jedem Tick aktuell verfügbar. Denkt man sich die Schnittstelle, bestehend aus diesen beiden Strukturen, als Controller, so kann an dieser Stelle von einer *closed-loop control* (siehe dazu auch [17]) gesprochen werden, bei der der Controller Sensorinformationen bekommt und aufgrund dessen Kommandos setzt, die wiederum zu neuen Sensorwerten führen. Um die Zyklenfreiheit bei einem reinen SCADE-Controller zu gewährleisten, müssten die Kommandos vom Controller der Simulation um einen Tick verzögert übergeben werden, da dies andernfalls instantan geschehen würde und damit zu einem unerlaubten Zyklus führte.

Die eigentliche Simulation des Streckennetzes findet im `Tracksimulator`-Operator statt, der im Folgenden beschriebenen wird.

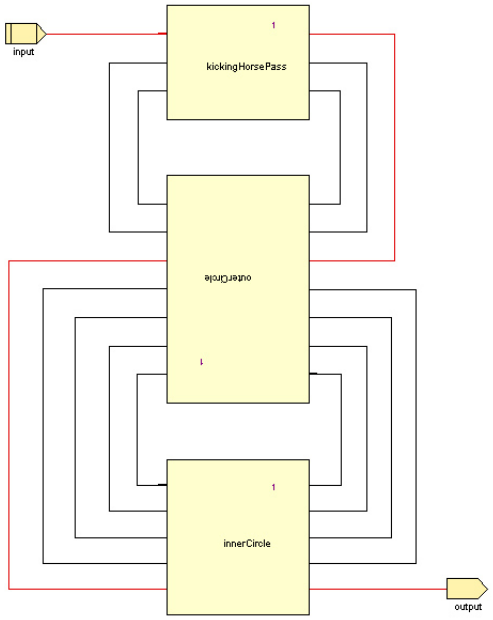


Abbildung 2.5.: Der Operator `Tracksimulator`

2.4.3. Tracksimulator

Der Tracksimulator spiegelt den zuvor erwähnten modularen Grundaufbau des Simulationsmodells entsprechend seines realen Vorbilds wider, unterteilt in die drei Kreise Kicking Horse Pass, Outer Circle und Inner Circle.

An seiner Schnittstelle bekommt der Tracksimulator die bereits erläuterten Daten `controllerCommands` und stellt die `controllerFeedbackSensor`-Daten bereit. Beides wird als *globale Daten* durch alle drei Kreise geschleift und ist in der Abbildung 2.5 rot markiert. Die jeweiligen Operatoren der Kreise sind dafür verantwortlich, diese Daten instantan weiterzuleiten, damit sie auch beim nächsten

Kreis noch im gleichen Makrotick anliegen und zudem auch als Feedback noch den Controller erreichen. Dies ist notwendig, um auf Controller-Eingaben noch im selben Tick Simulations-Ausgaben produzieren zu können, wie es die Synchronizität fordert. Es müssen daher sowohl die Controller-Eingaben alle erforderlichen Operatoren innerhalb eines solchen Ticks erreichen als auch die Simulations-Ausgaben all dieser Operatoren als Ausgabe des gesamten Simulations-Operators dem Controller bereitstehen. Beides stellt die instantane Weiterleitung der globalen Daten sicher. Die schwarz markierten Verbindungen stellen die *lokalen Daten* dar, die nur zwischen je zwei Streckenelementen ausgetauscht werden und entsprechen genau den Gleisübergängen von einem auf den anderen Kreis im Streckenlayout (vgl. Abbildung 1.2). Die einzelnen Teilkreise, aus welchen sich der `Tracksimulator` zusammensetzt, werden nachfolgend näher betrachtet.

2.4.4. InnerCircle, OuterCircle und KickingHorsePass

Der `Tracksimulator` besteht aus drei Gleiskreisen, welche wiederum modular nach dem Vorbild der realen Bahnanlage aufgebaut sind, und zwar aus allen Streckenelementen, aus denen auch das echte Gleissystem besteht. Im Simulationsmodell entsprechen diese Elemente gerade den `Track`-Operatoren. Beispielhaft sei das Aufbauprinzip eines solchen Kreises anhand des Kicking Horse Pass durch Abbildung 2.6 veranschaulicht. Als Streckenelemente können die größeren der beiden hauptsächlich verwendeten Operatoren identifiziert werden. Die kleineren stellen lediglich Verzögerungsoperatoren dar, die Zyklen unterbinden sollen. Angeordnet sind die `Track`-Operatoren wie sie es das vereinfachte Streckenlayout vorgibt. Neben diesen gibt es noch weitere größere Operatoren, die auf einer Seite mit jeweils zwei anderen Track-Operatoren verbunden sind. Dabei handelt es sich um `SwitchPoint`-Operatoren, welche der Modellierung von Weichen aus dem realen Gleissystem dienen.

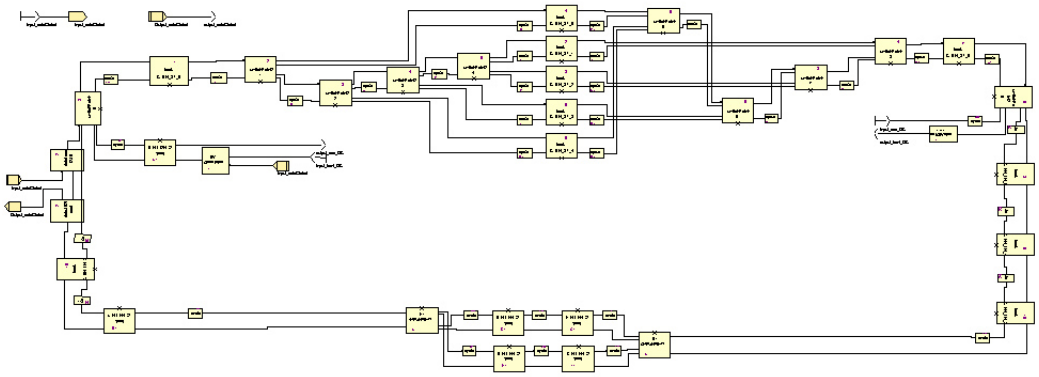


Abbildung 2.6.: Der Operator `KickingHorsePass`

Auffällig bei der Betrachtung ist, dass die Streckenelemente immer jeweils doppelt miteinander verbunden sind. Dies begründet sich in der Tatsache, dass es konstruktionsbedingt möglich sein soll, auf den modellierten Schienen sowohl vorwärts[7] als auch rückwärts zu fahren.

In der oberen Verbindung werden sowohl lokale als auch globale Daten übermittelt bzw. durchgeschleift. Die lokalen Daten der oberen Verbindung sind die Informationen für die Hauptfahrtrichtung. Der Datenfluss der unteren Verbindung verläuft in genau der *entgegengesetzten* Richtung. Hier werden nur lokale Daten und zwar die der Gegenfahrtrichtung übermittelt. Damit zwischen zwei Track-`Operatoren` kein Zyklus entsteht, ist es zwingend erforderlich, die sog. `Cyclebreaker`[8] dazwischen zu schalten. Da SCADE bei der Prüfung auf Zyklenfreiheit nur die aller oberste Ebene der Operatoren berücksichtigt, bzw. noch die in der *Expansion List* aufgelisteten, sind die Verzögerungen nicht in den `Track`-Operator inkludiert. In dem Fall müsste SCADE jeden dieser Operatoren expandieren. Zu den anderen Gleiskreisen existieren an dieser Stelle nur noch Interfaces, die lokale Datenverbindungen vorsehen und welche letztlich im `Tracksimulator`-Operator verbunden sind.

Eine Besonderheit stellen noch die `switchAlignment`-Operatoren im Inner Circle dar. Sie befinden sich bei den Übergängen zum Outer Circle/Kicking Horse Pass. An diesen beiden Punkten treffen gerade jeweils zwei Gleisabschnitte mit unterschiedlicher Hauptfahrtrichtung aufeinander. Deshalb muss beim Überfahren eines Zuges das eine Streckenelement in Haupt- und das andere in Gegenfahrtrichtung geschaltet sein. Damit der Zug bei diesem Wechsel nicht seine Orientierung ändert, ist es erforderlich, die Zugausrichtung hier durch den `switchAlignment`-Operator zu korrigieren.

2.4.5. Track

Der `Track`-Operator ist der als kleinste Einheit definierte Streckenelementsoperator. Er modelliert genau ein Streckenelement aus dem Gleisplan (s. Abbildung 1.2).

Im vorangegangenen Abschnitt 2.4.4 wurde diskutiert, warum der Operator je zwei Schnittstellen für In- und Outputs besitzt. Nachfolgend sollen die Funktionsweisen dieses grundlegenden Operators genauer beschrieben werden.

Informationsfluss

Der `InterfaceFlow` ist in ein eigenes Diagramm ausgelagert. Hier werden sowohl die globalen als auch die lokalen Datenströme behandelt. Erstere befinden sich, wie die lokalen Daten für die *primary* Hauptfahrtrichtung, am `InputFront`-Eingang

[7] Da diese Bezeichnung irreführend sein kann und um konsistent zu den im Modell verwendeten Begriffen zu bleiben, wird im Folgenden bei der Fahrtrichtung zwischen der Hauptfahrtrichtung (*primary*) und der Gegenfahrtrichtung (*secondary*) unterschieden.

[8] Bei den `Cyclebreaker`-Operatoren handelt es sich gerade um die kleineren, bereits erwähnten Verzögerungsoperatoren.

der Operatorschnittstelle. Die lokalen Daten für die *secondary* Gegenfahrtrichtung finden sich dagegen im `InputRear`-Eingang wieder.

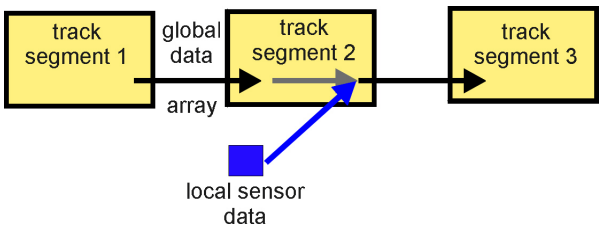


Abbildung 2.7.: Einfügen von Sensorwerten im `InterfaceFlow`-Diagramm

Wie bereits in Abschnitt 2.4.1 erläutert und in der Abbildung 2.7 noch einmal veranschaulicht, ist in den globalen Daten jeweils genau ein Index für einen der 47 Streckenelemente reserviert. Hier werden zum einen die Kommandos des Controllers für diesen Streckenabschnitt ausgewertet, wobei es sich im Speziellen um die Felder `motormode` (Fahrtrichtungen) und `speed` (Geschwindigkeiten) handelt. Zum anderen dient das Diagramm darüber hinaus auch der Einschleusung von aktuellen Informationen wie Kontaktmeldungen, Zugpositionen, Fehlermeldungen und Sensorwerten in die entsprechenden Arrays durch `Modifier`-Operatoren. Diese einfachen Operatoren sind in C implementiert und speziell an die jeweiligen Datentypen angepasst. Hierbei handelt es sich um eine Hilfskonstruktion, die im Abschnitt 2.5 näher beschrieben wird. Fehlermeldungen der Weichen werden unverändert weitergereicht. Zu beachten ist hierbei, dass die globalen Daten selbst auf dieser untersten Ebene instantan weitergeleitet (und evtl. verändert) werden.

Zugübernahme

Die lokalen Daten dienen der Übergabe von Zügen zwischen zwei Streckenelementen. Im Diagramm `InterfaceFlow` werden die lokalen Daten der *primary* und der *secondary* Fahrtrichtung miteinander verglichen. Je nach dem, von welcher Seite ein Zug übergeben wird, erfolgt dabei die Emittierung des Signals `entersFront` bzw. `entersRear` (oder im Fehlerfall die Emittierung beider, s. Abschnitt 2.4.5). Außerdem findet hier eine Abspeicherung der vorherigen Geschwindigkeit und des *Alignments*[9] in lokalen Variablen statt.

Die Berechnung der aktuellen Zugausrichtung erfolgt, wie in Abbildung 2.8 dargestellt, anhand der letzten Zugausrichtung, da es für einen Zug unmöglich ist, sich auf einem Streckenabschnitt selbst umzudrehen. Die Zugausrichtung wird nur dann

[9] Mit *Alignment* ist hier die Ausrichtung des Zuges gemeint, d.h. ob er in Hauptfahrtrichtung oder in Gegenfahrtrichtung orientiert steht.
[10] Das Streckenelement wird wiederum durch seinen Index innerhalb des Arrays identifiziert.

aktualisiert, wenn ein neuer Zug das Streckenelement befährt, also `entersFront` oder `entersRear` emittiert ist. Schließlich kommt es in der Initialisierungsphase (d.h. solange das Signal `initTrain` emittiert ist) zusätzlich zur Übernahme der Zugausrichtung aus der Initialbelegung der Züge.

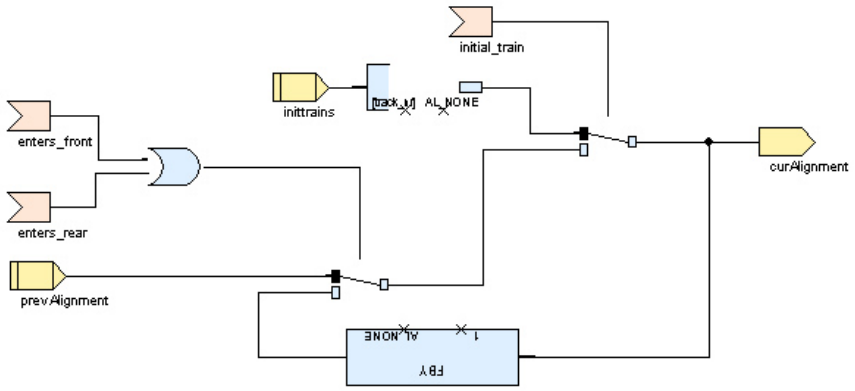


Abbildung 2.8.: *Alignment*-Übergabe im Diagramm `InterfaceFlow`

Initiale Züge

Um initial Züge zu positionieren, muss ein entsprechender Eintrag im Datenfeld `initialTrainArray`, welches Teil der `controllerCommands` ist, vorhanden sein. Neben der Information, ob sich auf einem bestimmten Streckenelement ein initialer Zug befindet[10], muss hier auch gleich dessen Ausrichtung angeben sein. Es bestehen also für die Belegung eines Feldelements die folgenden Möglichkeiten:

- **AL_NONE :** Kein initialer Zug vorhanden
- **AL_FORWARD :** Initialer Zug mit Ausrichtung in Hauptfahrtrichtung
- **AL_BACKWARD :** Initialer Zug mit Ausrichtung in Gegenfahrtrichtung

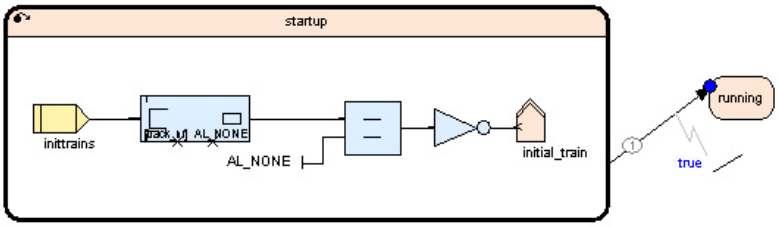


Abbildung 2.9.: Initiale Züge

Die Initialisierungsphase der Simulation (s. Abbildung 2.9) dauert genau einen Tick und ist anschließend abgeschlossen. Falls innerhalb dieses Ticks ein initialer

Zug, egal welcher Ausrichtung, im angesprochenen Datenfeld gesetzt ist, wird das Signal `initTrain` emittiert. Dieses Signal dient sowohl zur Übernahme der Zugausrichtung im Diagramm `InterfaceFlow`, als auch der Übernahme des Zuges selbst (als Zugposition) im Diagramm `PositionComputation`.

Positionsinformationen

Die eigentliche Simulation der Zugbewegungen auf einem Streckenabschnitt findet im Diagramm `PositionComputation` statt. Hier wird, solange das Gleis durch einen Zug belegt und eine zu diesem Diagramm gehörende Zustandsmaschine (s. Abbildung 2.10) im Zustand `track_occupied` ist, die Variable `position_head` und damit die Position des Anfangs[11] des Zuges in jedem Tick neu berechnet. Über die maximale Zuglänge, die als Konstante vorliegt, kann dann auch die Position des Zugendes (`position_tail`) bestimmt werden. Diese beiden Variablen sind nur lokal verfügbar. Nach außen sichtbar (und in die globalen Daten eingeschleust) sind lediglich die modifizierten Variablen `out_pos_head` und `out_pos_tail`, in denen dann schon die Streckenabschnittsgrenzen (alles unter 1 und alles über 100) berücksichtigt sind und außerdem auch die berechnete Zugausrichtung mit eingeht.

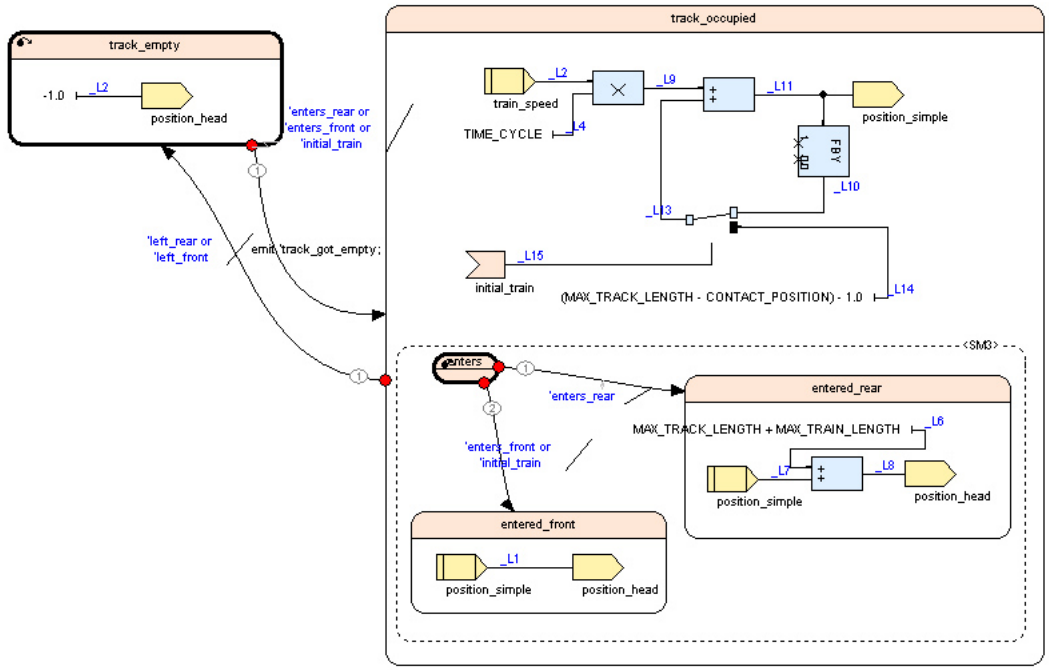


Abbildung 2.10.: Positionsberechnung im `Track`-Operator

[11] Der Zuganfang entspricht der Position der Lokomotive, das Zugende der Position des letzten Waggons.

Zugfahrtrichtung

Die Zugfahrtrichtung (s. Abbildung 2.11) wird im selben Diagramm berechnet. Grundlage hierfür ist, ähnlich wie bei der Zugausrichtung, die Fahrtrichtung im letzten Tick, falls das Kommando des Controllers für die Fahrtrichtung (`motormode`) des Gleisabschnitts `OFF` oder `BRAKE` ist. Sollte die Fahrtrichtung für den Gleisabschnitt auf `PRIMARY` bzw. auf `SECONDARY` gesetzt sein, so wird die gespeicherte Zugfahrtrichtung als der selbigen entsprechend in diese Schleife übernommen.

Kontaktauslösung

Die Auslösung der Kontakte erfolgt mit Hilfe des `ContactSetting`-Operators. Da ein Zug beim Überfahren eines Kontakts diesen je einmal sowohl mit der Lok als auch mit dem Zugende auslöst, werden beide Positionsangaben für die Berechnung herangezogen. Außerdem benötigt der Operator darüber hinaus die Position der Kontakte auf dem Gleisabschnitt und die aktuelle Zugfahrtrichtung. Letztere wird auf der realen Bahnanlage durch ein Reed-Kontaktpaar gemessen. Die Simulation greift hier auf die zuletzt gespeicherte Zugfahrtrichtung zurück, die, wie im vorangegangenen Abschnitt gezeigt, bestimmt wird.

Zusätzliche Sensorwerte

Die Modellbahnhardware verfügt, neben der Messung der Zugfahrtrichtung an einem Kontaktpaar, noch über weitere Messmethoden, um den Controller mit zusätzlichen Sensorwerten zu versorgen. Dazu gehören die folgenden:

TrackOccupied : Zeigt an, ob sich eine Lokomotive auf dem Gleis befindet. Auf der realen Bahnanlage wird dies über Spannungsabfallmessungen bestimmt. Die Simulation bedient sich dabei der Positionsangaben des Zuganfangs.

Speed : Ist die gemessene Geschwindigkeit des Zuges. Hier wird die simulierte Geschwindigkeit verwendet, mit der auch die simulierte Zugposition verändert wird.

TrackShutdown : Zeigt auf der echten Modellbahnanlage an, dass ein Kurzschluss vorliegt und ist in der Simulation dauerhaft mit `false` belegt.

SpeedNewData : Die Leistungselektronik der Bahnhardware misst alle 0,2 Sekunden einen neuen Geschwindigkeitswert. Das Ereignis wird durch das Vorhandensein eines neuen Messergebnisses ausgelöst. In der Simulation ist dies durch einen Rückwärtszähler implementiert.

Alle aufgeführten Sensorwerte sind in einer Struktur `TrackExtenedData` zusammengefasst und werden, wie auch die Positionsangaben und Kontaktmeldungen, im Diagramm `InterfaceFlow` instantan in die globalen Daten eingeschleust.

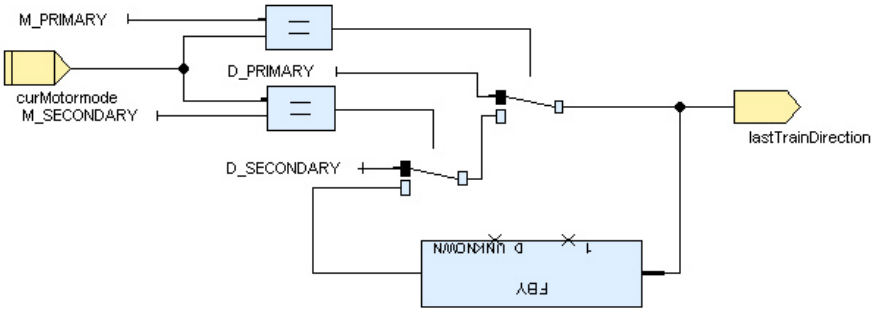


Abbildung 2.11.: Zugfahrtrichtung

Fehlererkennung

Die Erkennung von Fehlerzuständen ist wiederum in ein eigenes Diagramm ausgelagert. Die Arten möglicher Fehler wurden innerhalb des Modellbahnpraktikums [7] in einer Fehlerbaumanalyse untersucht. Folgende Fehlerzustände werden von einem Streckenelement erkannt und dann in die globalen Daten eingefügt:

TRACK_DOUBLE_ENTRY : Diese Fehlermeldung wird genau dann ausgelöst, wenn im selben Tick zwei Züge gleichzeitig den Streckenabschnitt befahren möchten, d.h. von beiden Seiten Züge übergeben werden. Auf der Modellbahnanlage führte dies zu einem Zusammenstoß der Züge.

E_TRACK_OCCUPIED : Ist bereits ein Zug übergeben worden und befindet er sich innerhalb des Streckenabschnitts, so wird dieser Fehler dadurch ausgelöst, dass ein weiterer Zug von beliebiger Seite her übergeben wird, bevor der aktuelle Zug den Streckenabschnitt verlassen hat. In der Realität könnte hier nicht mehr zwischen beiden Zügen unterschieden werden, schlimmstenfalls wäre sogar erneut ein Zusammenstoß der Züge denkbar.

E_MOTORMODE_MISMATCH : Die Auslösung dieses Fehlers erfolgt, wenn bei der Übernahme eines Zuges die Fahrtrichtung der beiden beteiligten Streckenabschnitte nicht übereinstimmt. Dies würde auf der realen Anlage zu einem Kurzschluss führen.

E_SPEED_MISMATCH : Auch unterschiedliche Geschwindigkeiten sind bei der Übernahme eines Zuges unerwünscht, da hier in der Realität unerwünscht Strom fließen würden, falls die Geschwindigkeit über die Spannung reguliert wird.

E_OK : Ist keiner der obigen Fehlerzustände aktiv, so wird statt einer Fehlermeldung dieser Platzhalter für den Gleisabschnitt in das globale Fehlerfeld geschrieben.

Sobald einer der obigen Fehlerzustände erkannt wurde, erfolgt ein Festhalten dieses in einer Fangschleife, um zu verhindern, dass der Programmierer des Controllers

diesen übersehen oder gar ignorieren kann. Bereits nur ein solcher Fehler könnte auf der Modellbahnanlage später die Hardware u.U. schwer beschädigen.

2.4.6. Switchpoint

In der Simulation gibt es drei verschiedene Weichenoperatoren:

1. `SwitchPoint1`
2. `SwitchPoint2`
3. `SwitchPointCross`

Die ersten beiden *Einfach-Weichentypen* können auf ihrer einen Seite mit zwei und auf ihrer anderen Seite mit einem `Track`-Operator verbunden werden. Der dritte Operator ist ein komplexer Weichentyp, welcher an jeder Seite den Anschluss zweier `Track`-Operatoren zulässt (Kreuzungs-Weichentyp).

Einfach-Weichentypen

Zunächst werden die ersten beiden Weichentypen genauer untersucht. In der Abbildung 2.12 sind die Funktionsweisen schematisch dargestellt.

Dabei entsprechen die lokalen Daten den schwarzen Pfeilen, wobei die Pfeilrichtung mit der Datenflussrichtung übereinstimmt. Die roten Pfeile stellen die globalen Daten dar. Da sie im Modell mit den lokalen Daten der Hauptfahrtrichtung (in einer Struktur) gekoppelt sind, wird dies in der Grafik durch die örtliche Nähe der Pfeile berücksichtigt.

Auf der Ebene der lokalen Daten unterscheiden sich die beiden Weichentypen nicht. Hier werden die Daten der angeschlossenen Streckenelemente entsprechend der Weichenstellung durchgereicht. Nur für die globalen Daten, die unabhängig von der Stellung der Weichen weitergeleitet werden müssen, existieren hier Unterschiede.

Der komplexere linke Weichentyp hat dabei die Aufgabe, beide globalen Datenströme zusammenzuführen. Bei diesem Vorgang wird es keine Überschneidungen definierter Feldelemente geben, denn jeder Streckenabschnitt hat seinen eigenen Index und damit sein eigenes Element in den Datenstrukturfeldern. Eine Zusammenführung ist aber überaus wichtig, da andernfalls beim Controller nur noch partielle Informationen und Sensorwerte als `controllerFeedbackSensor` ankommen würden. Das Zusammenführen wird mit Hilfe der drei Operatoren `MergeGlobalData`, `MergeSingleData` und `MergeSingleDataArray` durchgeführt.

Die Implementierung des rechten Weichentyps ist einfacher, da ihm lediglich die Aufgabe zukommt, die globalen Datenströme an beide angeschlossenen Streckenelemente weiterzuleiten.

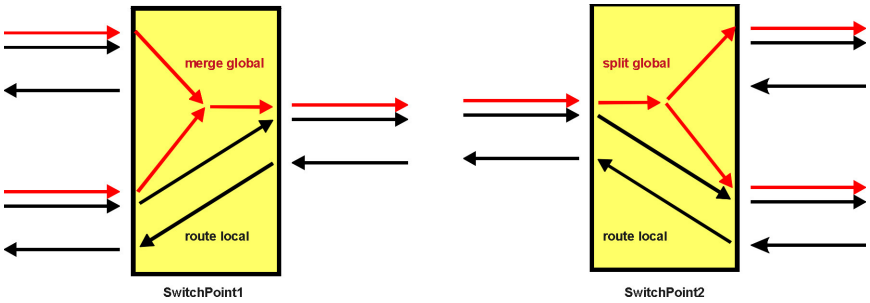


Abbildung 2.12.: Beide Einfach-Weichentypen (schematisch)

Kreuzungs-Weichentyp

Die Kreuzungsweichen werden in der Modellbahnanlage nur an zwei Stellen verwendet, und zwar an beiden Enden des Outer Circle Bahnhofs, an welchen es sowohl Übergänge in den Kicking Horse Pass, als auch in den Inner Circle gibt. Eine genaue Betrachtung, wie diese Weichen in der Bahnanlage verschaltet sind, findet sich auf der Homepage der Modellbahn [19] und in der Diplomarbeit von Stephan Hörmann [20, Kapitel 2.5].

In der Simulation besteht dieser Weichentyp funktionell aus zwei zusammengeschalteten *Einfach-Weichentypen*, mit einigen Besonderheiten, weshalb hier ein separater Operator gewählt wurde. An den Schnittstellen-Enden zu den jeweils anderen Kreisen muss hier zusätzlich darauf geachtet werden, dass nur lokale Daten übertragen werden. Eine Weiterleitung der globalen Daten innerhalb des Outer Circle, unabhängig von den Weichenstellungen, ist ebenso erforderlich. Außerdem existiert jeweils noch eine verbotene Weichenstellkombination, welche durch die fest vorgegebene Hauptfahrtrichtung im Inner Circle und Outer Circle bedingt ist. Für eine detaillierte Betrachtung dieser Kombinationen sei erneut auf die Modellbahnhomepage [19] verwiesen.

Fehlererkennung

Auch bei den Weichen war die Grundlage für eine Fehlererkennung die im Rahmen des Modellbahnpraktikums [7] durchgeführte Fehlerbaumanalyse. Die folgenden Fehler werden von einem Weichenoperator erkannt und dann in die globalen Daten eingefügt:

E_WRONG_SWITCHPOINT_DIRECTION : Dieser Fehlerzustand wird dadurch erreicht, dass ein Zug in das offene, nicht geschaltete Ende eine Weiche hinein fährt. Auf der realen Bahnanlage würde ein solcher Zug entgleisen.

E_FORBIDDEN_SWITCHCROSS_SETTING : Die Kreuzungsweiche ist falsch beschaltet worden. Diese verbotene Weichenstellkombination kann und darf nicht verwendet werden, denn sie führt in der Realität zu einem Kurzschluss.

E_MOTORMODE_MISMATCH : Die an die Weiche angeschlossenen und durch die Weichenstellung verbundenen Streckenabschnitte müssen mit der gleichen Fahrtrichtung beschaltet sein, damit es zu keinem Kurzschluss bei der Bahnhardware kommt.

E_SPEED_MISMATCH : Das gleiche gilt auch für die Geschwindigkeiten der angeschlossenen und durch die Weiche verbundenen Streckenabschnitte.

E_OK : Ist keiner der obigen Fehlerzustände aktiv, so wird statt einer Fehlermeldung dieser Platzhalter in das globale Fehlerfeld geschrieben.

Auch hier wird, wie bei der Fehlererkennung des `Track`-Operators, sobald einer der obigen Fehlerzustände erkannt wurde, dieser in einer Fangschleife festgehalten, damit der Programmierer des Controllers diesen nicht übersehen oder ignorieren kann.

2.4.7. PrepareDisplayData

Der Operator `PrepareDisplayData` (s. Abbildung 2.13) bereitet Daten aus der Simulation für die Ausgabe auf einem entsprechenden GUI auf. Hier kann z.B. das ModelGUI-Projekt verwendet werden, welches das nächste Kapitel thematisiert.

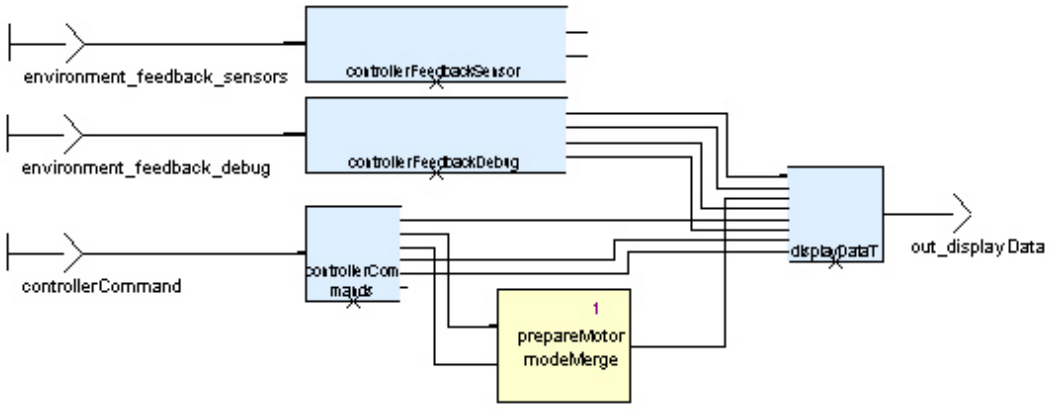


Abbildung 2.13.: Der Operator `PrepareDisplayData`

In der Tabelle 2.1 sind alle Datenstrukturfelder mit ihrer Größe aufgelistet, aus denen sich die Struktur `displayData` in genau dieser Reihenfolge zusammensetzt.

Name	Beschreibung	Grösse
startPosition	Positionen der Zuganfänge (Loks)	48
endPosition	Positionen der Zugenden	48
motormodes	Fahrtrichtungen	48
blockErrors	Fehlerzustände für Streckenelemente	48
points	Weichenstellungen	48
pointErrors	Fehlerzustände für Weichen	48
signals0	Jeweils erste Signale in Hauptfahrtrichtung	49
signals1	Jeweils zweite Signale in Hauptfahrtrichtung	49

Tabelle 2.1.: Datenstrukturfelder von `displayData`

Innerhalb der Datenfelder ist die Zuordnung bei den Streckenelementen (startPosition, endPosition, motormodes, blockErrors, signals0, signals1) durch ihre alphabetische Reihenfolge gegeben, bei den Weichen (points, pointErrors) durch ihre fortlaufende Nummer gemäß des Streckenplans.

Dabei haben die beiden letzten Arrays als Besonderheit 49 Einträge, wobei der jeweils letzte der Ansteuerung des Bahnübergangs mit seinen beiden Signalen gilt. Außerdem enthält das vorletzte Array über alle ersten Signale in Hauptfahrtrichtung immer genau dort nicht genutzte und später zu ignorierende Elemente, wo das zugehörige Streckenelement nur über ein Signal verfügt, wie es z.B. im Inner Circle und Outer Circle der Fall ist. Der Operator `prepareMotormodeMerge` sorgt lediglich dafür, dass falls als Geschwindigkeit `0.0` eingestellt, dann auch der `motormode` entsprechend auf `OFF` gesetzt ist. Die Sensorwerte werden z.Zt. nicht weiter verwendet, könnten aber bei einer möglichen Erweiterung des GUI interessant werden.

2.5. SCADE-Simulationslauf

SCADE bietet eine integrierte Simulationsumgebung an, mit der Modelle und auch einzelne Operatoren simuliert werden können. Dafür muss zuvor entsprechend C-Code generiert und kompiliert werden. Die Generierung des C-Codes für die gesamte Simulation erfolgt wie im Anhang A beschrieben. Nachdem der Quellcode durch einen Compiler in eine ausführbare DLL überführt wurde, kann aus der SCADE-IDE heraus die Simulationsumgebung (s. Abbildung 2.14) gestartet werden.

In dieser Umgebung lässt sich das Modell der Bahnsimulation testen. Als Eingabemöglichkeiten stehen hier die `controllerCommands` zur Verfügung, die in der Schnittstelle des Operators `Simulation` als *input* definiert sind. Als Ausgaben können alle Werte der `displayData` und der `controllerFeedbackCommand` betrachtet und auf ihre Richtigkeit hin überprüft werden. Diese sind ebenfalls in der obigen Schnittstelle definiert, allerdings als *output*.

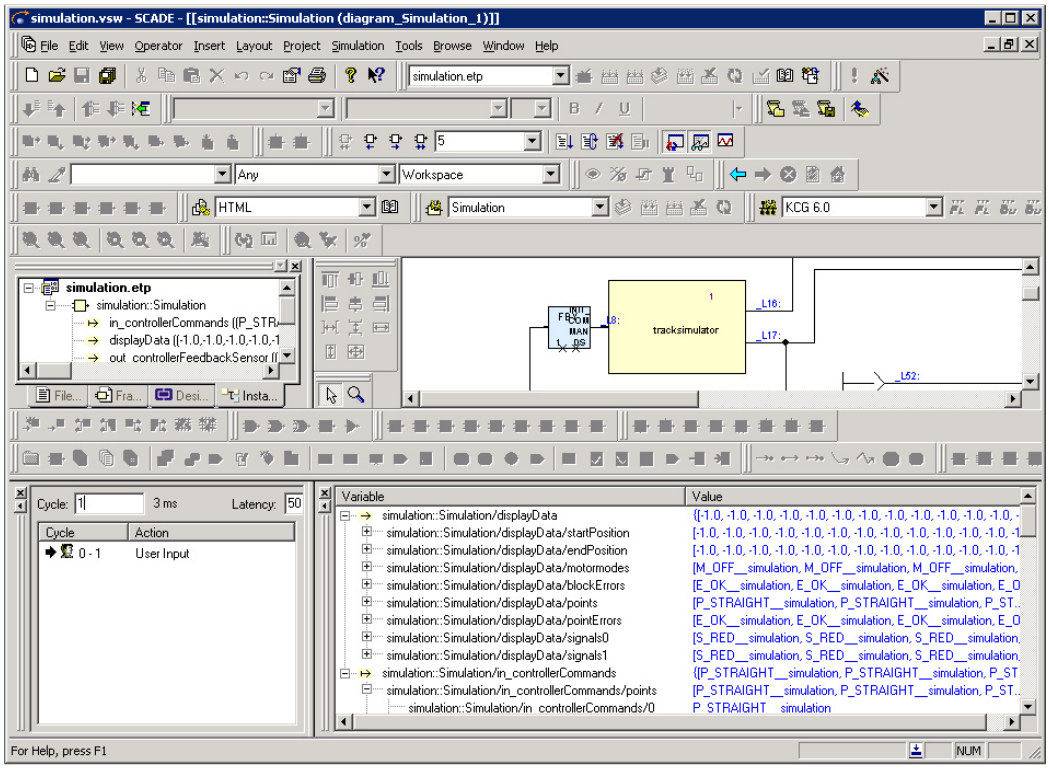


Abbildung 2.14.: Die SCADE-Simulationsumgebung

Für umfangreichere Tests, insbesondere nach größeren Erweiterungen, ist es zweckmäßig, einen kleinen Dummy-Controller in SCADE zu erstellen und diesen im Operator `Simulation` direkt mit den `controllerCommands` und den `controllerFeedbackCommand` zu verbinden. Hier werden, zur Modifizierung der in den `controllerCommands` definierten Unterstrukturfeldern, weitere C-Operatoren benötigt, für die bereits der passende C-Quellcode bereitgestellt wird. Nach dem Muster z.B. des `ContactPairModifier`-Operators können damit entsprechend weitere sog. *imported C operator*s erstellt werden. Diese Operatoren dürften mit zukünftigen SCADE-Versionen überflüssig sein, da erwartet wird, dass die Manipulation von Datenfeldern zukünftig (wieder) mit reinen SCADE-Operatoren möglich ist. Bei der Implementierung und Verwendung eines SCADE-Controllers ist schließlich darauf zu achten, die Zyklenfreiheit im Operator `Simulation` durch einen entsprechenden Verzögerungsoperator sicherzustellen.

Der in Abbildung 2.15 dargestellte einfache Beispielcontroller zeigt, wie leicht eine solche `controllerCommands`-Struktur vom Controller mit Hilfe bereits vordefinierter, durch entsprechende C-Modifier veränderbare, Konstantenfelder erstellt werden kann.

In diesem Controller wird das Feedback der Simulation nicht verwendet, weshalb der Eingang terminiert ist. Mit dem Konstantenfeld `INIT_POINTS` werden alle Wei-

chen auf STRAIGHT gestellt. Die Fahrtrichtung ist auf allen Streckenabschnitten OFF, mit Ausnahme von IC_ST_1, wo sie auf PRIMARY gesetzt ist. Auch die Geschwindigkeit wird, bis auf das Gleiselement IC_ST_1, überall mit 0 initialisiert. Dieses ist mit der Geschwindigkeit 40 beschaltet. Die Signale werden alle mit GRÜN belegt.

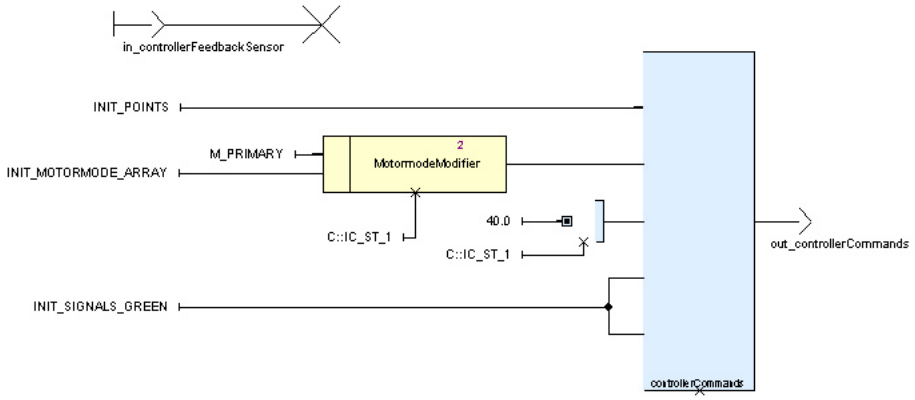


Abbildung 2.15.: Ein einfacher SCADE-Controller

3. Visualisierung mit der ModelGUI

3.1. Motivation

Im vorangegangenen Kapitel wurde in Abschnitt 2.5 beschrieben, wie das Simulationsprojekt in SCADE simuliert werden kann. Hierbei bietet SCADE dem Benutzer lediglich eine rein textuelle Schnittstelle an, in welcher sich alle im Interface des Operators `Simulation` definierten Datenstrukturen ansehen und dort als *input* definierte Datenstrukturen auch während der Simulation verändern lassen.

Unter einer Visualisierung oder Veranschaulichung wird allgemein die Aufbereitung von abstrakten Daten verstanden, die dann in eine visuell leichter erfassbare Form gebracht werden. Unter Umständen werden in einer solchen Aufbereitung auch für die Visualisierung vernachlässigbare Details der ursprünglichen Daten heraus gefiltert.

Da die Simulation der Modellbahnanlage, wie einführend erläutert wurde, dazu dienen soll, dem Entwickler des Controllers ein möglichst realitätsbezogenes Zustandsbild der simulierten Bahnhardware zu vermitteln, ist eine rein textuelle Anzeige, wie sie SCADE liefert, für diese Zwecke ungünstig. Auch die Ausgabe der Simulationsdaten als Funktionsgraph stellt in diesem Zusammenhang nur ein unzufriedenstellendes Hilfsmittel dar. Eine grafische Aufbereitung der textuellen Daten in einem **G**raphical **U**ser **I**nterface (GUI), und damit eine Visualisierung, erscheint daher angebracht.

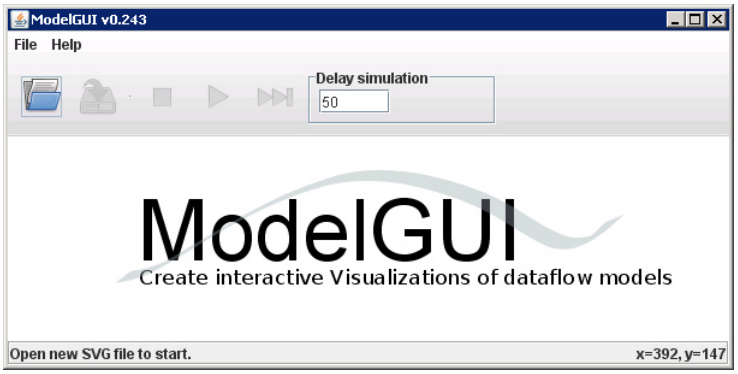


Abbildung 3.1.: Das Model GUI Projekt

Mit der ModelGUI [6], die an der Universität Kiel entwickelt wird, existiert bereits ein Projekt, das eine unkomplizierte Veranschaulichung von Datenflussmodellen ermöglichen soll. Die nächsten Abschnitte widmen sich der Vorstellung diese Projekts

und deren Anbindung an SCADE. Im Speziellen wird dabei auf die Visualisierung der simulierten Modellbahn eingegangen.

3.2. Das ModelGUI Projekt

Bei der ModelGUI handelt es sich um ein unabhängiges, auf JAVA basierendes Programm, welches Datenflusssimulationen verschiedener Programmierumgebungen visualisieren können soll. Das Programm entsteht an der Christian-Albrechts-Universität Kiel am Lehrstuhl für Echtzeitsysteme und Eingebettete Systeme. Obwohl sich die ModelGUI derzeitig noch in ihrer Entwicklung befindet, existieren schon voll lauffähige Versionen. Mit diesen ist es bereits möglich, sich über ein entsprechendes integriertes Interface mit den Modellierungswerkzeugen SCADE und Matlab/Simulink [5] zu verbinden. Die Abbildung 3.1 zeigt die Oberfläche des Programms nach dem Starten.

3.3. Anbindung an SCADE

Die Kommunikation der ModelGUI mit den Daten bereitstellenden Programmierumgebungen erfolgt über die TCP/IP-Schnittstelle. Um die ModelGUI während der Simulation in SCADE mit letzterem zu verbinden, muss SCADE auf einen TCP/IP-Port lauschen und damit Serverfunktionalität bereitstellen.

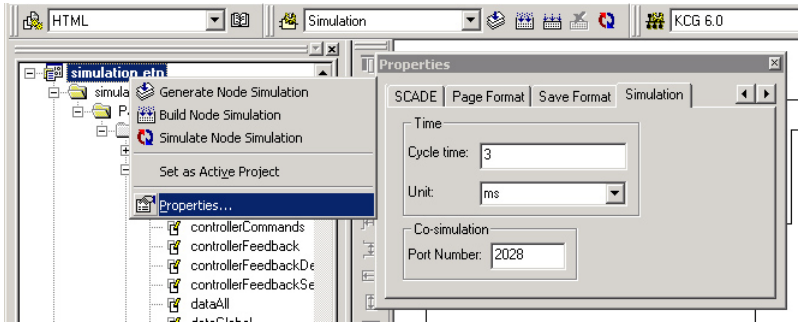


Abbildung 3.2.: Verbindungseinstellungen in SCADE

Die Abbildung 3.2 zeigt, wo sich diese Einstellungen in der aktuellen SCADE-Version befinden. Es ist darauf zu achten, keine bereits belegten Ports zu verwenden. In der ModelGUI muss unbedingt der gleiche Port im Einstellungs-Dialog angegeben werden. Außerdem ist SCADE aus der Liste der Modellierungswerkzeuge auszuwählen.

Nach dem Laden einer zur Modellsimulation gehörenden SVG-Datei, die im weiteren Verlauf noch näher erläutert wird, kann die Simulation von der ModelGUI aus

über den *Play*-Button gestartet und auch gesteuert werden. Die ModelGUI kontrolliert SCADE dabei im sog. *slave mode* und beeinflusst so die Geschwindigkeit der von SCADE ausgeführten Simulationsschritte.

Zu jeder mit der ModelGUI visualisierten Simulation eines Modells gehören die folgenden beiden Dateien:

1. **SVG-Datei :** In dieser Datei ist eine Vektorgrafik hinterlegt, die grafische Objekte enthält. Sie sollen von der ModelGUI bei der Visualisierung angezeigt werden können.

2. **MAP-Datei :** In der MAP-Datei findet eine Verknüpfung der Eigenschaften eines solchen grafischen Objektes mit einem festgelegten Simulationsdatum statt, das von dem mit der ModelGUI verbundenen Programm bereitgestellt wird.

Die folgenden beiden Abschnitte beschäftigen sich mit diesen Dateien für das Modellbahnsimulationsprojekt und erläutern deren genauen Aufbau und ihre Bearbeitung.

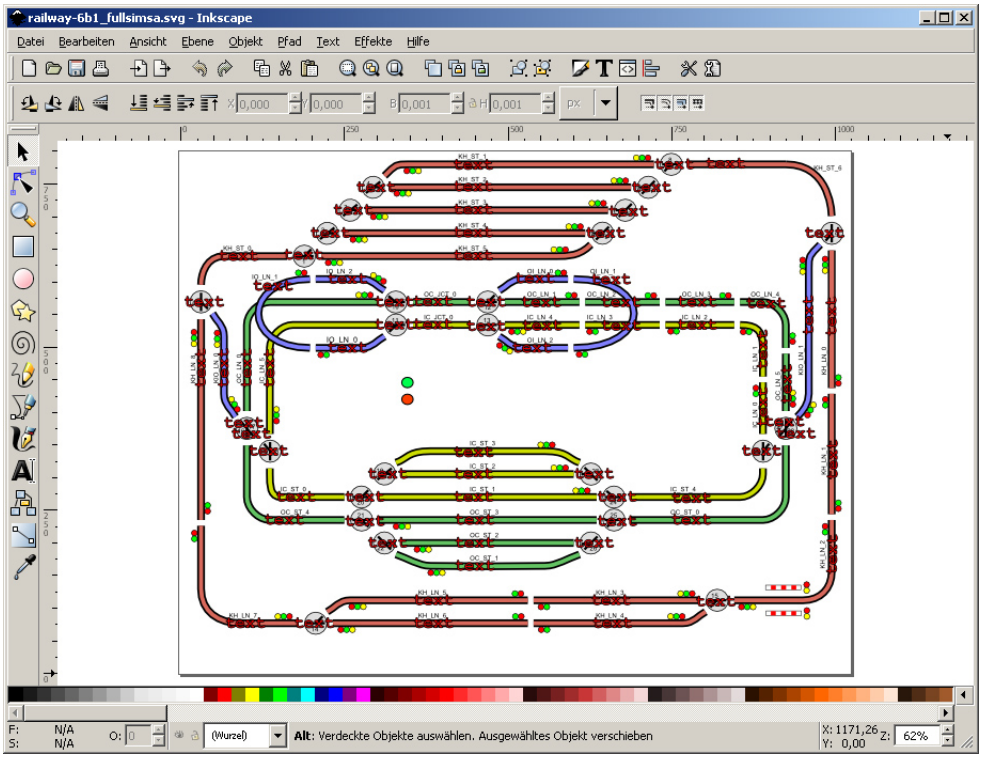


Abbildung 3.3.: Modellbahnsimulations-SVG-Datei im Inkscape Programm

3.3.1. Die SVG-Datei

SVG steht für **S**calable **V**ector **G**raphic und ist ein Dateiformat in welchem beliebig skalierbare Vektorgrafiken gespeichert sind. Die SVG-Datei, in welcher die

Vektorgrafik für die Modellbahnsimulation enthalten ist, kann einem beliebigen SVG-Zeichenprogramm z.B. mit dem Programm *Inkscape* [3] geöffnet und modifiziert werden. Die Abbildung 3.3 zeigt die Oberfläche des Programms mit der geöffneten, zur Simulation gehörenden Verktorgrafikdatei.

In dieser Grafik lässt sich das vereinfachte Streckenlayout des Gleissystems der Modellbahnanlage wieder erkennen. Dabei ist das Streckensystem hier in die selben Streckenelemente unterteilt wie das echte Vorbild. Die Elemente besitzen zudem einen, ihrer realen Pendants entsprechenden, gleichen Namen[11]. Auch die Weichennummerierung wurde aus der Realität in die Vektorgrafik übernommen. Dabei bestehen Weichen aus zwei durch entsprechende Namen identifizierte Objekte; einmal eines für die Linie bei der Stellung für Geradeausfahrt und ein zweites für die Linie bei der Stellung auf Abzweig (s. Abbildung 3.5). Je nachdem, ob die Weiche auf `STRAIGHT` oder `TURN` steht, soll schließlich nur eine der beiden Linien sichtbar gemacht und die jeweils andere versteckt werden. Letzteres ist die Aufgabe der im nächsten Abschnitt untersuchten MAP-Datei. Die Streckenelemente verfügen teilweise weiterhin über mehrere Signale. Dabei haben alle Signallampen eigene Bezeichnungen, wobei auch hier, konsistenterweise, die ersten Signale in Hauptfahrtrichtung mit `0` und die zweiten Signale in Hauptfahrtrichtung mit `1` bezeichnet sind.

Objekt	**Bezeichnung**	**Beispiel**
Streckenabschnitt	<trackname>	IC_ST_1
Weichen (geradeaus)	point_straight_<pointnumber>	point_straight_8
Weichen (abzweig)	point_turn_<pointnumber>	point_turn_8
Signale Rot	R<0|1>_<trackname>	R1_IC_ST_1
Signale Grün	G<0|1>_<trackname>	G1_IC_ST_1
Signale Gelb	Y<0|1>_<trackname>	Y1_IC_ST_1
Fehler Strecken	t_<trackname>	t_IC_ST_1
Fehler Weichen	t_point_<pointnumber>	t_point_8
Bahnübergang[12]	bar<1,...,8>	bar5
Zuganfang	engine	engine
Zugende	trailer	trailer

Tabelle 3.1.: Objektbezeichnungen in der SVG-Datei

[11] Die Festlegung der Namen erfolgt in den Objekteigenschaften.

[12] Der Bahnübergang besteht aus mehreren Einzelobjekten.

[13] Zusammengehörende SVG- und MAP-Dateien müssen vor, vom Suffix abgesehen, den gleichen Namen besitzen, damit die ModelGUI beim Öffnen der Vektorgrafik die dazugehörige Zuordnungsdatei finden kann.

[14] SCADE sendet bei Aufzählungstypen nicht den entsprechenden `Integer`-Wert, sondern den Namen der Aufzählungskonstanten mit einem angehängten „_simulation“ für den Operator, welcher simuliert wird.

Im Fehlerfall, während eines Simulationslaufs, sollen die roten Textlabels an den jeweiligen Streckenabschnitten und Weichen, akute Fehlermeldungen zur Anzeige bringen. Der Bahnübergang besteht aus den Objekten `bar1,...,bar8` für die Schranken und hat die Streckenbezeichnug `KH_X` für die Signale. Der grüne und rote Punkt markieren in der laufenden Simulation den Anfang und das Ende eines jeden simulierten Zuges.

In Tabelle 3.1 sind nochmals alle verwendeten Bezeichnungen für die Objekte der Vektorgrafik aufgelistet.

3.3.2. Die MAP-Datei

Bei der gleichnamigen[13] MAP-Datei, in welcher das *Mapping* (Zuordnung) für die zugehörige SVG-Datei der Modellbahnsimulation enthalten ist, handelt es sich um eine reine Text-Datei in XML-Notation. Sie lässt sich mit jedem beliebigen Texteditor bearbeiten.

Aufgabe der Datei ist die Verknüpfung der über TCP/IP eingehenden Simulationsdaten mit (benannten) Objekten aus der SVG-Datei. Die bei der Simulation von SCADE an die ModelGUI übermittelten[14] Daten sind für das Modellbahnsimulationsprojekt in der Struktur `displayData` verpackt, wie dies in Abschnitt 2.4.7 beschrieben ist. Die Tabelle 2.1 listet alle Datenstrukturfelder in der Reihenfolge auf, in welcher sie in der Struktur `displayData` abgelegt sind und in der sie auch an die ModelGUI übertragen werden. Jedes Datum innerhalb von `displayData` wird, aus Sicht der ModelGUI, nur noch über einen Index angesprochen. Dabei können sich die Arrays innerhalb von `displayData` einfach als aneinander gereiht vorgestellt werden, wobei sich der Index dann auf dieses eine, zusammengesetzte, große Datenfeld bezieht.

In der MAP-Datei werden diese Indizes mit *Port* bezeichnet. Für jeden solchen Port lässt sich hier genau festlegen, welche Objekte aus der SVG-Datei bei welchem Wert dieses Datums von der ModelGUI wie angezeigt werden sollen. Eine detaillierte Beschreibung dieser Möglichkeiten findet sich in der Wiki-Dokumentation [6] zur ModelGUI.

Im Folgenden soll nun anhand von Beispielen ein kurzer Überblick über die verwendeten Konstrukte gegeben werden.

Streckenabschnitte

```
1 <display port="104">
2    <colorize color="#ffffff, #00ff00, #ff0000, #000000"
3     color-property="stroke" id="b_IC_ST_1" input="M_OFF__simulation,
4 M_PRIMARY__simulation, M_SECONDARY__simulation, M_BRAKE__simulation"/>
5 </display>
```

Streckenabschnitte (s. Abbildung 3.4) sollen während der Simulation den Status der beschalteten Fahrtrichtung erkennen lassen. Dazu wird der Streckenabschnitt für jede Fahrtrichtung mit einer anderen Farbe umrahmt. Dies kann mit dem obigen

Code erreicht werden. Dabei entsprechen sich die aufgezählten Farben und die anschließend aufgezählten Fahrtrichtungen. Der Port 104 ist hier gleich dem Index des Datums `IC_ST_1` im `motormode`-Feld als Teil des Gesamtfeldes.

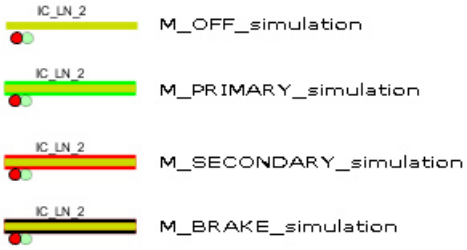


Abbildung 3.4.: Visualisierung von Streckenabschnitten

Weichen

```
<display port="200">
    <opaque id="point_straight_12"
       input="P_STRAIGHT__simulation, P_TURN__simulation" opacity="1, 0.3"/>
    <opaque id="point_turn_12"
       input="P_STRAIGHT__simulation, P_TURN__simulation" opacity="0.3, 1"/>
</display>
```

Die Weiche 12 (s. Abbildung 3.5) in der Verktorgrafik besteht, allen anderen Weichen entsprechend, aus zwei identifizierbaren Objekten mit dem Namen `point_straight_12` und `point_turn_12` für die jeweilige Weichenstellung. Für diese wird an dieser Stelle jeweils ein Opazitätsvektor[15] definitert. Je nachdem, ob der Wert des zum Port 200 gehörenden Datums `STRAIGHT` oder `TURN` ist, wird entweder das eine oder das andere Objekt sichtbar und damit die eingestellte Weichenstellung erkennbar gemacht.

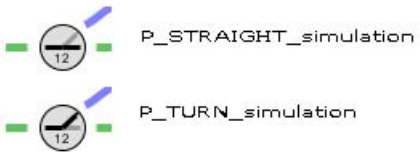


Abbildung 3.5.: Visualisierung von Weichen

[15] Der Opazitätsvektor definiert, je nach Eingabedatum, Werte für die *Durchsichtigkeit* des Objekts, zu welchem er gehört.

Signale

```
<display port="303">
   <opaque id="R0_KH_LN_0"
     input="S_RED__simulation,␣S_YELLOW__simulation,␣S_GREEN__simulation" opacity="1,␣0.3,␣0.3"/>
   <opaque id="Y0_KH_LN_0"
     input="S_RED__simulation,␣S_YELLOW__simulation,␣S_GREEN__simulation" opacity="0.3,␣1,␣0.3"/>
   <opaque id="G0_KH_LN_0"
     input="S_RED__simulation,␣S_YELLOW__simulation,␣S_GREEN__simulation" opacity="0.3,␣1,␣1"/>
</display>
```

Für Signale (s. Abbildung 3.6) wird dies ähnlich, und zwar für jede Farbe einzeln festgelegt. Da es außerdem drei mögliche Einstellungen für das Signaldatum, nämlich `S_RED`, `S_YELLOW` und `S_GREEN` gibt, besteht der Vektor hier aus drei Werten. Bei `S_RED` wird nur die rote Signallampe aufleuchten, bei `S_YELLOW` die grüne und gelbe zusammen und bei `S_GREEN` nur die grüne. Andere theoretisch denkbare Signalkombinationen lässt die aktuelle Simulation aus Plausibilitätsgründen nicht zu.

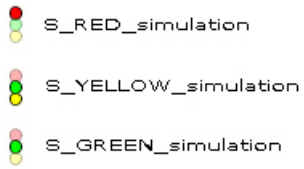


Abbildung 3.6.: Visualisierung von Signalen

Fehlermeldungen

```
<display port="152">
    <opaque id="t_IC_ST_1" input="E_OK__simulation" opacity="0"/>
    <textbox id="t_IC_ST_1"/>
</display>
```

Fehlermeldungen (s. Abbildung 3.7) werden direkt angezeigt. Nur die `E_OK`-Meldung wird durch eine Opazität von 0 versteckt, da sie einem Platzhalter für *kein Fehler* entspricht.

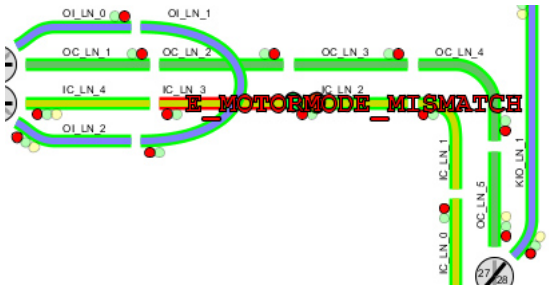


Abbildung 3.7.: Visualisierung von Fehlermeldungen

Züge

```
<display port="8">
    <move-path id="engine" input="1..100" path="IC_ST_1"/>
</display>
<display port="56">
    <move-path id="trailer" input="1..100" path="IC_ST_1"/>
</display>
```

Züge werden in der laufenden Simulation (s. Abbildung 3.8) als ein grüner und ein roter Punkt dargestellt, wobei der grüne dem Anfang und der rote dem Ende des Zuges entsprechen. Sowohl für den Anfang als auch für das Ende des Zuges liegen die Positionsangaben zwischen `1` und `100`, falls sich ein Zugende auf dem Gleisabschnitt befindet. Diese nehmen hingegen den Wert `0` an, wenn das betreffende Zugende nicht auf dem Gleisabschnitt liegt oder sich gar kein Zug auf diesem aufhält. Gibt es einen gültigen Wert zwischen `1` und `100`, so wird der entsprechende Punkt entlang des Pfades auf dem betrachteten Streckenabschnitt von der ModelGUI eingezeichnet.

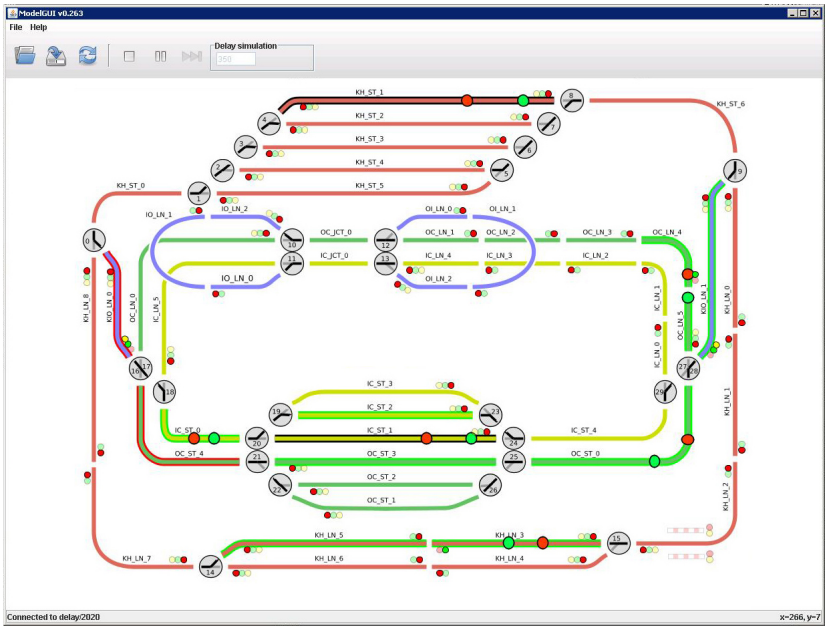


Abbildung 3.8.: Visualisierung der Modellbahn mit der ModelGUI

3.4. Anbindung zum Simulationsinterface

Das im nächsten Kapitel thematisierte Simulationsinterface entspricht in seiner Anbindung an die ModelGUI jener von SCADE. Dies bedeutet, dass es sich genau wie SCADE als TCP/IP-Server von der ModelGUI aus ansteuern lässt und die Simulationsdaten ebenso wie SCADE zurücksendet. Es muss dabei erneut darauf geachtet

werden, einen gültigen und freien Port zu wählen und diesen sowohl der ModelGUI als auch dem Interface mitzuteilen. Grundsätzlich sind die gleichen Einstellungen in der ModelGUI vorzunehmen (s. Abschnitt 3.3), die auch für eine Verbindung zu SCADE gemacht würden.

Das Simulationsinterface und wie entsprechende Einstellungen auf dieser Seite vorgenommen werden können, wird das nächste Kapitel erörtern.

4. Controller-Schnittstellen

4.1. Motivation

Die Modellbahnanlage verfügt bereits über eine C-Bibliothek (API[1]). Diese erlaubt es einem Controllerprogramm, das auf einem Hostrechner läuft, die Bahnhardware abstrakt anzusteuern. Dabei benutzt das Programm entsprechende *higher level* Funktionen aus dieser Bibliothek und kann so, ohne die Implementierungsdetails und Kommunikationswege der Bahnanlage zu kennen, u.a. Streckenabschnitte unter Strom setzen, Weichen stellen oder Kontakte abfragen.

Die Bibliothek unterstützt derweil nur den CAN und den Ethernet Bus. Das geschlossene System des TTP-Busses verwendet eine andere proprietäre Werkzeugkette und wurde bisher nur modellbasiert entwickelt. Für z.B. SCADE ist die Simulation hierfür direkt ausgelegt. Die C-Bibliothek wurde im Rahmen der Diplomarbeit von Stephan Hörmann [20] entwickelt. Implementierungsdetails dazu können selbiger Arbeit entnommen werden.

Nachdem bereits ein lauffähiges SCADE-Modell der Bahnhardware zur Verfügung steht, welches sich auch mit SCADE und der ModelGUI simulieren und visualisieren lässt, kann hingegen z.B. ein für die Bahn existierender, in C geschriebener Controller nicht ohne weiteres mit dieser zusammen getestet werden. Weiterhin benötigt die Simulation des Bahnmodells stets die SCADE Suite zu ihrer Ausführung.

Ziel des folgenden Teils dieser Arbeit ist es, die Modellbahnsimulation von der SCADE Suite zu lösen und eine Schnittstelle zu implementieren, die der bereits bestehenden für die reale Anlage ähnelt. Damit wäre die Simulation auch von Controllern ansteuerbar, die nicht in SCADE sondern in C entwickelt wurden.

Ist eine solche Schnittstelle implementiert, dann ermöglicht es diese, in C geschriebene Controller zunächst mit der Simulation zu testen und dann später ohne Änderungen damit die reale Modellbahnanlage zu steuern. Es ist jedoch noch nicht möglich, einen in einer anderen Programmiersprache (z.B. JAVA) abgefassten Controller in ähnlicher Weise mit der Simulation bzw. der Bahnanlage zu verbinden.

Eine zusätzliche Herausforderung ist es daher, über die beiden sich ähnelnden C-Schnittstellen, eine noch universellere Schnittstelle zu schaffen, die sowohl die Simulation als auch die Modellbahnanlage weiteren Programmiersprachen zugänglich

[1] API steht für **a**pplication **p**rogramming **i**nterface und ist eine Sammlung von Funktionen einer Schnittstellenbibliothek.

macht. Mit einer TCP/IP-Schnittstelle wird auch dieses Ziel erreicht und im zweiten Teil des Kapitels vorgestellt.

4.2. Die C-Schnittstelle

Um die Verwendbarkeit einer solchen C-Schnittstelle zu optimieren, werden die nachfolgend aufgezählte Bedingungen an diese Schnittstelle gestellt.

- Sie sollte sich einfach in den Programmcode einbinden lassen.
- Ein Controllerprogramm sollte sich nicht um die Details einer Verbindung zur ModelGUI kümmern müssen.
- Die aus der bestehenden C-Bibliothek bekannten Funktionen sollten in möglichst gleicher Weise funktionieren.
- Die Schnittstelle sollte möglichst plattformunabhängig sein.
- Eine Veränderung oder Erweiterung des SCADE-Modells sollte ohne größeren Aufwand möglich sein, d.h. ohne den generierten Code modifizieren zu müssen.

In den folgenden Abschnitten wird gezeigt, wie die Schnittstelle diesen Anforderungen gerecht wird.

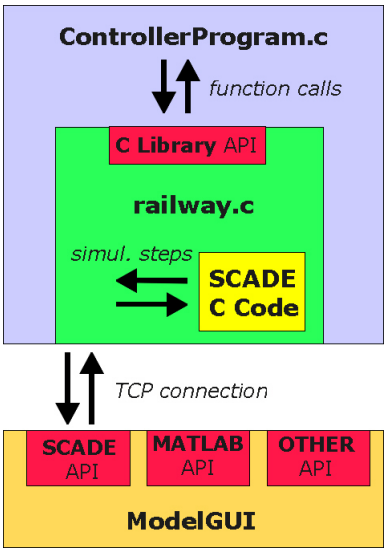


Abbildung 4.1.: Die C-Schnittstelle (schematisch)

Ein solches Interface operiert auf drei Ebenen. Zum einen ist hier das Verhalten der Funktionen aus der bestehenden C-Bibliothek zu nennen, welches es zu emulieren gilt. Auf der anderen Seite muss der von SCADE generierte C-Code für diese Emulation verwendet werden. Zusätzlich ist noch eine Verbindungsmöglichkeit für die

ModelGUI zu schaffen, welche später den Ablauf der Simulation steuern soll. Dieser Vorgang ist sich veranschaulicht vorzustellen, wie in Abbildung 4.1 dargestellt.

4.2.1. TCP/IP-Serverfunktionalität

Da die ModelGUI auf TCP/IP als Verbindung zu SCADE aufbaut und diese Verbindungsart auch für die Simulationsschnittstelle verwendet werden soll, muss ein TCP/IP-Server bereitgestellt werden, der Kommandos empfangen und mit Daten antworten kann. Für die Abstraktion von der asynchronen Sende- und Empfangs-Implementierung werden dafür die Funktionen `tcp_server_send()` und `tcp_server_receive()` verwendet. Die Initialisierung des Servers und das damit verbundene Öffnen eines, auf den übergebenen Port lauschenden Socket, erledigt die Funktion `tcp_server_init()`.

4.2.2. SCADE-Code

Der von SCADE generierte C-Code wird von der Simulation verwendet. Ein integrierter Build-Prozess hilft mit einem Make-Skript bei der automatischen Übernahme von neu generiertem Code (s. Abschnitt 4.2.9).

Von außerhalb gibt es in diesem Code nur zwei aufrufbare Funktionen:

1. `Simulation_reset_simulation()`
2. `Simulation_simulation()`

Erstere dient zur Initialisierung der von SCADE gelieferten Simulationsdaten. Dafür muss zuvor eine Struktur vom Typ `inC_Simulation_simulation` angelegt werden, die beim Funktionsaufruf (*by reference*) übergeben wird. Diese Struktur ermöglicht es später auf alle globalen Output-Daten zuzugreifen, für die ein entsprechender Eintrag in der Interface-Definition des Operators `Simulation` existiert; also zu den SCADE-Datenstrukturen `controllerFeedbackSensor` und `displayData`.

Die zweite Funktion führt einen Simulationsschritt aus. Ihr müssen neben der Datenstruktur für die Simulationsdaten auch die globalen Input-Daten übergeben werden. Dabei handelt es sich um eine Struktur vom Typ `outC_Simulation_simulation`, in welcher schließlich die `controllerCommands` enthalten sind. In ihr müssen alle Werte definiert sein, bevor sie der Funktion (*by reference*) übergeben wird. Als Ergebnis eines Simulationsschrittes lassen sich nach der Funktionsausführung die erstgenannten Simulationsdaten auswerten.

4.2.3. ModelGUI-Schnittstelle

Um der Anforderung gerecht zu werden, dass sich das spätere Controllerprogramm nicht um die Kommunikation mit der ModelGUI kümmern muss und dieses durch die Simulation vollständig gekapselt wird, ist es erforderlich, diesen Teil in einen eigenen Thread auszulagern. Dieser wird dann im Hintergrund arbeiten und so dem im

Vordergrund laufenden Controllerprogramm verborgen bleiben. Der Controllerthread kommuniziert mit jenem nur über die, an die bestehende C-Bibliothek der Modellbahn angelehnte Funktionsbibliothek.

Bei der Initialisierung der Simulation über die Funktion `initSimulation()` wird ein solcher Thread erzeugt, der die Funktion `SimulationServer()` abarbeitet. In letzterer ist die gesamte Kommunikation zur ModelGUI vom Verbindungsaufbau bis hin zur Bearbeitung von Datenanfragen gekapselt. Die Funktion `initSimulation()` wartet nicht auf das Ende des Threads (kein `pthread_join`) und kehrt sofort zurück. Diese Funktion muss aufgerufen werden, wenn die Simulation initialisiert werden soll. Außerdem muss bei ihrem Aufruf ein Port für die Kommunikation mit der ModelGUI übergeben werden. Im Controllerquellcode erfolgt dieser Aufruf implizit aus der Funktion `railway_initsystem()` heraus.

Damit am ModelGUI-Programm keine Änderungen vorgenommen werden müssen, emuliert die Schnittstelle für die ModelGUI-Kommunikation SCADE und lässt sich darüber hinaus in gleicher Weise von der ModelGUI aus steuern.

Die Kommunikation läuft nach dem folgenden Schema ab, welches auch in Abbildung 4.2 veranschaulicht wird:

1. Betreten einer Schleife, die nur durch `abortSimulation()` (oder `railway_stopcontrol()`) verlassen wird und andernfalls bei Verbindungsabbruch auf eine neue Verbindung wartet.

2. Lauschen auf dem geöffneten Port.

3. Verbindungsaufbau durch ModelGUI als Client.

4. Betreten einer weiteren Schleife, in der kontinuierlich auf Kommandos von der ModelGUI gewartet wird, um diese zu bearbeiten.

5. Kommando empfangen.

6. Falls ein Verbindungsabbruch erkannt wird, beende die innere Schleife.

7. Andernfalls werte das Kommando aus:

   Close : Reinitialisieren der Simulation und Verlassen der inneren Schleife.

   Step : Führe mit `Simulation_simulation()` einen Simulationsschritt aus und werte anschließend die Simulationsdaten aus.

   GetOutputValue : Verschicke die sich in den Simulationsdaten befindende Datenstruktur `displayData` so, wie dies auch SCADE tun würde, als einen großen Datenstrom, in dem Unterdatenstrukturen durch entsprechende Klammerung gekennzeichnet sind.

8. Gehe zu Punkt 5 (innere Schleife).

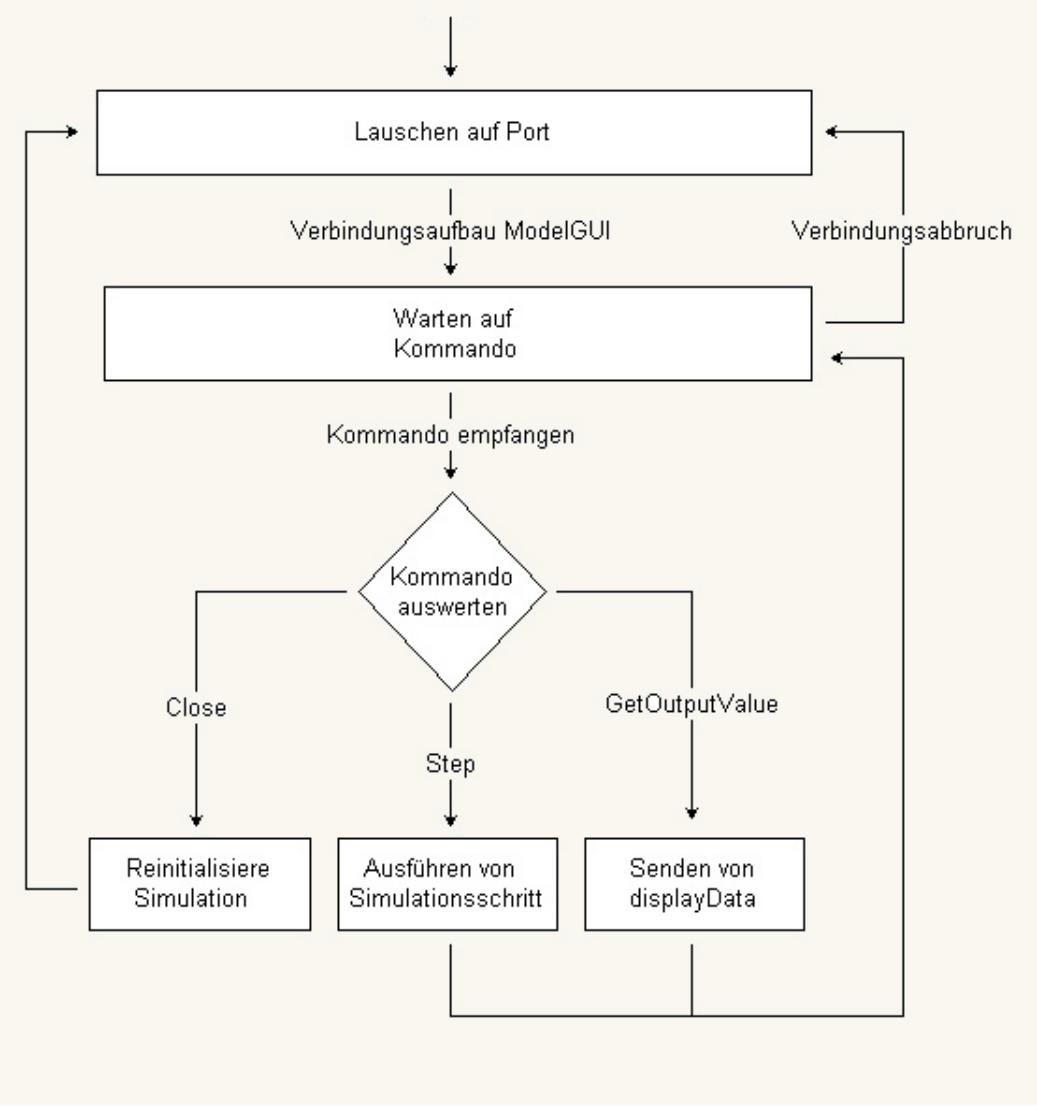


Abbildung 4.2.: Ablaufdiagramm für die Kommunikation mit der ModelGUI

Dadurch, dass ein Simulationsschritt nur durch ein Kommando von der ModelGUI ausgelöst werden kann, ist diese während des Simulationslaufs der Taktgeber, wie es auch bei SCADE im *slave mode* der Fall ist.

Um zu verhindern, dass der Kommunikationsthread nach Beenden des vordergründigen Controller-Threads noch im Hintergrund verbleibt und weiterhin auf einen Port lauscht, ist es nötig, diesen durch einen Aufruf von `abortSimulation()` abschließend zu beenden. Im Controllerquellcode übernimmt letzteres implizit die Funktion `railway_stopcontrol()`.

4.2.4. Kontaktauslösungen

Um die TTP-Kommunikation zu vereinfachen, wurde in der SCADE-Simulation vereinzelt abstrahiert. Zur Verallgemeinerung und Angleichung der API an die bestehende C-Bibliothek muss dies nun berücksichtigt werden.

Die Kontakte auf der realen Bahnanlage sind so beschaffen [20], dass sie beim Überfahren einer Lok und beim Überfahren des Zugendes auslösen. Dabei kommt eine Kondensatorschaltung zum Einsatz, in der nur, falls die Auslösung eine bestimmte Dauer überschreitet und damit der entsprechende Flankenwechsel eindeutig ist, das Ereignis als Kontaktauslösung gezählt wird. Die Bahnelektronik speichert dabei die Richtung und die Auslösung selbst. Mit der Funktion `getcontact()` aus der C-Bibliothek der Bahn kann diese Auslösung anschließend vom Bahncontroller abgefragt werden. Eine solche Abfrage sollte nicht zu selten stattfinden, da die Puffer für die Speicherung dieser Kontaktauslösungsereignisse nur eine begrenzte Größe besitzen.

Funktion	**Beschreibung**
`MEM_set_new_contact()`	Speichert ein neues Kontaktevent.
`MEM_istriggered_contact()`	Gibt zurück, ob ein bestimmter Kontakt ausgelöst wurde.
`MEM_iscurrent_event()`	Gibt zurück, ob dies das einzig für den Kontakt gespeicherte Ereignis ist.
`MEM_get_contact()`	Gibt die gespeicherte Richtung zurück oder UNI falls diese nicht mehr vorliegt.
`MEM_clear_contact()`	Entfernt das älteste Kontaktereignis.
`MEM_clearall_contact()`	Entfernt alle Events zu einem bestimmten Kontakt
`MEM_reset_contacts()`	Löscht den gesamten Puffer und alle gespeicherten Kontaktereignis.

Tabelle 4.1.: Pufferungsfunktionen für Kontaktereignisse

Die Kontakte in der Simulation werden dagegen auf jeden Fall beim simulierten Überfahren eines Zuges in gleicher Weise mit der Lok bzw. dem Zugende ausgelöst. Dies geschieht nach Abschnitt 2.4.5 im Operator `contactSetting`. Die darin verwendete Statemachine stellt sicher[16], dass eine Auslösung nur in einem einzigen Simulationsschritt stattfindet. Dies bedeutet, dass eine Kontaktauslösung in den Simulationsdaten nur für einen Tick lang sichtbar ist. Ein SCADE-Controller oder ein auf den TPP-Knoten laufender Controller würde auf dieses Ereignis aufgrund der Synchronizität ebenfalls in genau einem Ausführungsschritt reagieren können. Da

[16] Eine Emittierung des Ereignisses erfolgt nur bei Zustandsänderung über die Transitionsaktion.

die ModelGUI aber den Fortschritt der Simulation steuert, ist ohne eine zusätzliche Speicherung dieses Ereignisses eine solche Reaktion nicht gewährleistet. Die Gründe dafür liegen in den beiden unabhängig und nebenläufig ausgeführten Threads, zum einen für die Kommunikation zur ModelGUI und zum anderen für den Bahncontroller. Damit hängt es letztlich vom Scheduling des Betriebssystems ab, ob der Controller das Ereignis überhaupt zu sehen bekommt.

Um sicherzustellen, dass der Controller unabhängig vom Scheduling auf jeden Fall auf das Ereignis (früher oder später) reagieren kann, werden alle Kontaktauslösungen von der Schnittstelle in einem eigenen Puffer festgehalten, bis sie abgerufen wurden. Dafür stehen der Schnittstelle intern die in Tabelle 4.1 aufgelisteten Funktionen zur Verfügung.

Nach jedem von der ModelGUI initiierten Simulationsschritt werden stets alle Kontaktauslösungen als neue Kontaktereignisse in diesen Puffer übertragen.

4.2.5. Initiale Züge

Züge und ihre Position gehören auf der realen Modellbahnanlage zum Gesamtzustand des Systems. Hier hat der Benutzer die Möglichkeit Züge manuell auf andere Gleise zu setzen, umzudrehen, zu verschieben oder sie gänzlich zu entfernen.

In der Simulation wurde die Initialbelegung der Züge anfangs in einer Konstanten festgelegt. Dies erwies sich später als hinderlich, da für jede Änderung der Konstanten wieder neuer C-Code generiert und kompiliert werden musste. Aus diesem Grund wurde die `controllerCommands`-Struktur erweitert und ermöglicht so ein Einschleusen dieser Initialbelegung im ersten Simulationsschritt (s. Abschnitt 2.4.5).

```
1 ####################################
2 # initial train sample config file #
3 ####################################
4
5 # a comment line is ignored
6 not a valid line that will also be ignored (warning)
7 IC_ST_2
8 8
9 *KH_ST_2
```

Abbildung 4.3.: Setzen von initialen Zügen

Über die zusätzlichen Funktionen `setInitialTrain()`, `setInitialTrainEx()` und `resetInitialTrains()` hat ein Controllerprogramm die Möglichkeit, auf die Initialbelegung der Züge einzuwirken. Dabei sollten diese Funktionen an den Anfang gelegt werden, denn sie müssen für einen Einsatz des Controllers mit der realen Bahn wieder entfernt werden. Aus diesem Grund wird von einer solchen Vorgehensweise abgeraten und anstelle dessen die nachfolgend beschriebene empfohlen.

Es ist auch möglich, initiale Züge in einer eigenen Konfigurationsdatei (s. Abbildung 4.3 und Abschnitt 4.2.7) anzugeben, nach der bei Simulationsstart gesucht wird. Ist die Datei „railway.cnf“ im aktuellen Verzeichnis (oder im Suchpfad) vorhanden, so wird versucht die Initialbelegung anhand dieser Textdatei zu setzen. Mit `#`

beginnende oder leere Zeilen werde darin ignoriert. Jeder Streckenabschnittsname (oder auch dessen Nummer), auf der ein Zug initialisiert werden soll, steht in einer eigenen Zeile. Zeilen, die mit * gefolgt von einem Streckenabschnittsnamen (oder einer Nummer) beginnen, resultieren in der Initialisierung des entsprechenden Streckenelements mit Zügen, welche entgegen der Hauptfahrtrichtung stehen. Dies ist i.A. nur auf dem Kicking Horse Pass erwünscht.

In der Implementierung erfolgt das Einlesen der Konfigurationsdatei mittels der Funktion `ReadConfigFile()`. Findet diese keine solche Datei, so wird sie wahlweise mit der Ausgabe einer Warnung oder kommentarlos, ohne Züge zu setzen, zurückkehren. Auch mit fehlerhaften Zeilen in der Datei wird entsprechend verfahren.

4.2.6. An die C-Bibliothek angelehnte Funktionen

Bei der Implementierung der Schnittstelle wurde sehr viel Wert darauf gelegt, die Funktionsnamen und Aufrufe aus der existierenden C-Bibliothek wiederzuverwenden, um die Änderungen an Controllern (s. Abschnitt 4.2.7), die sowohl für die Simulation als auch für die reale Bahnanlage konzipiert sind, so klein wie möglich zu halten.

In der C-Bibliothek existieren mehrere Module, die den Zugriff auf die Bahnhardware auf einer bestimmten Ebene zulassen (siehe dazu auch [20]). Da im Bahnsimulationsmodell jedoch nicht die Kommunikation von Leistungselektroniken und PC104-Rechnern berücksichtigt wird, kommt hier nur das Modul `railway` in Frage, welches das in der Hierarchiestufe höchste ist und die abstrakteste Sicht auf die Bahnhardware zulässt.

Zur Verwendung der Simulationsschnittstelle muss lediglich die „railway.h“ der Simulation eingebunden werden. Alle auch in der Simulation deklarierten Funktionen aus der C-Bibliothek, finden sich gegen Ende der Datei aufgelistet. Einige davon sind für die Simulation nicht interessant, andere wurden noch nicht implementiert und bieten so evtl. noch Spielraum für eine Weiterentwicklung. Meist ist anhand des Funktionsnamens bereits zu erkennen, welche Aufgabe die Funktion besitzt. Eine genaue Beschreibung ist jedoch der Diplomarbeit von Stephan Hörmann [20] zu entnehmen.

Im Folgenden soll kurz auf ausgewählte Funktionen eingegangen werden, denen in der Simulation und der existierenden C-Bibliothek eine leicht unterschiedliche Bedeutung zukommt.

`struct railway_system *railway_initsystem(struct railway_hardware *hardware)`

Diese Funktion liefert einen Dummy vom Typ `railway_system` zurück. Dieser Dummy hat für die Simulationsroutinen keine andere Verwendung. Für die echte

Bahn kann er nach diesem Funktionsaufruf jedoch auf ungleich `NULL` getestet werden und gibt so die erfolgreiche Initialisierung der Hardware an. Wichtiger für die Simulation ist, dass selbige in dieser Funktion initialisiert wird und zwar mit dem in der Initialisierungsdatei (s. Abschnitt 4.2.7) festgelegten Optionen. Die Funktion kehrt im Falle eines Wartens auf die ModelGUI erst zurück, wenn diese sich mit dem Controller verbunden hat.

`int railway_alive(struct railway_system *railway)`

Auf der Modellbahn kann mit dieser Funktion überprüft werden, ob die Steuerung noch normal arbeitet. In der Simulation wird hier auf eine positive Verbindung zur ModelGUI getestet.

`int contactexists(struct railway_system *railway, int block, int contact)`

Auf der realen Anlage verfügt nicht jeder Gleisabschnitt über Reed-Kontakte. Mit Hilfe dieser Funktion lässt sich dies für einen bestimmten Block abgefragen. Diese Funktion existiert zwar, ist jedoch *hart-codiert* und dementsprechend anzupassen, falls sich an der Bahnhardware diesbezüglich etwas ändern sollte.

`int railway_stopcontrol(struct railway_system *railway, int reset)`

Durch den Aufruf dieser Funktion wird `abortSimulation()` ausgeführt und damit der Simulationsthread, welcher von `initSimulation()` erzeugt wurde, beendet. Der Rückgabewert der Funktion ist stets `1`. Die Funktion kehrt im Falle eines Wartens auf die ModelGUI erst zurück, wenn diese die Verbindung zum Controller beendet hat (s. Abschnitt 4.2.7). Bei der realen Anlage führt der Aufruf dieser Funktion zum Herunterfahren der Steuerung und zum wahlweisen Zurücksetzen der Peripherie.

4.2.7. Controllermodifizierungen und die Initialisierungsdatei

Durch die Verwendung einer optionalen Initialisierungsdatei, die schon am Ende des Abschnitts 4.2.5 vorgestellt wurde, kann eine Änderung am Code des Controllers, der sowohl die Simulation als auch die reale Anlage steuern soll, gänzlich vermieden werden.

In dieser Datei kann zusätzlich der TCP/IP-Port festgelegt werden, an welchem die ModelGUI angebunden wird. Dies geschieht über die Option `GUIPORT`. Dieser wird ein freier Port, z.B. `2222` zugewiesen, wie dies der Auszug aus der Initialisierungsdatei in Abbildung 4.4 zeigt.

Außerdem ist es in der Datei mit der Option `WAITFORGUI` möglich, ein Warten des Controllerprogramms bei der Initialisierung der Simulation auf die ModelGUI festzulegen. Erst wenn die ModelGUI sich mit der Simulation verbunden hat, kehrt die Initialisierungsmethode `railway_initsystem()` zurück. Das Setzen dieser Option

hat ebenfalls ein solches Warten innerhalb der Funktion `railway_stopcontrol()` zur Folge. Bevor die Funktion die Verbindung abbricht und den Kommunikationsthread beendet, wartet sie darauf, dass die ModelGUI die Verbindung terminiert.

Eine weitere Option ist dafür gedacht, die Simulation und ModelGUI auf einem Windows2000-System lauffähig zu machen. Dazu ist die Option `WIN2KCOMPATIBLE` zu setzen. Dies sollte allerdings nur in Betracht gezogen werden, falls bei der Kommunikation zwischen Simulation und ModelGUI Probleme bestehen.

Ist die Option `QUIETMODE` aktiv, so werden keine Statusmeldungen vom Simulationsthread ausgegeben. Das sind z.B. Meldungen auf welchem Port gelauscht wird oder Meldungen über den Verbindungsstatus zur ModelGUI. Sollen derartige Hinweise unterdrückt werden, so ist diese Option einzuschalten.

Eine weitere Möglichkeit dafür bietet die letzte Option `LOGFILEOUTPUT`. Ist sie aktiviert, dann werden alle Meldungen in eine Log-Datei im aktuellen Verzeichnis umgeleitet. Dies betrifft dann auch Debug-Meldungen, die sich mit einer `#define`-Direktive im Quelltext aktivieren lassen.

Wird die Initialisierungsdatei nicht gefunden, oder sind Optionen darin nicht definiert, so wird standardmäßig der Port `2020` verwendet und nicht auf die ModelGUI gewartet. Ebenfalls ist die Kommunikation mit der ModelGUI standardmäßig nicht Windows2000-kompatibel, Nachrichten werden nicht in die Log-Datei umgeleitet und der Quiet-Modus ist nicht aktiv.

```
1  #set ModelGUI TCP port (optional)
2  GUIPORT = 2222
3
4  # Wait for ModelGUI during railway_initsystem() and railway_stopcontrol()
5  WAITFORGUI = 1
6
7  # Set ModelGUI communication for a Windows 2000 system
8  WIN2KCOMPATIBLE = 1
9
10 # Prevent the output of any messages from the simulation thread
11 QUIETMODE = 1
12
13 # Redirect all (debug) messages to a logfile named 'railway.log'
14 LOGFILEOUTPUT = 0
```

Abbildung 4.4.: Optionen in der Initialisierungsdatei

Im Controllerquelltext sollte außerdem evtl. die Pfadangaben der `#include`-Direktiven überprüft und angepasst werden.

Das Programm „apidemo.c“, welches ursprünglich die Funktionen der C-Bibliothek demonstrieren sollte und auch Teil der Diplomarbeit von Stephan Hörmann [20] ist, wurde zur Demonstrationszwecken für die Simulation ebenfalls in den Build-Prozess aufgenommen. Es kann als praktisches (Code-)Beispiel der soeben angeführten Bemerkungen herangezogen werden.

4.2.8. C-Beispielcontroller

Als einfaches Beispiel zur in diesem Kapitel thematisierten Funktionssammlung für die Simulation soll der nachfolgend besprochene Testcontroller aus Abbildung 4.5 vorgestellt werden.

Der Beispielcontroller hat die Aufgabe einen Zug, welcher sich initial auf dem Gleis IC_ST_1 befindet, einmal im Inner Circle fahren zu lassen. Sobald er auf dem Streckenabschnitt IC_LN_5 kurz vor dem Inner Circle Bahnhof ankommt, soll die Weiche 20 auf TURN gestellt und der Zug damit auf IC_ST_3 geleitet werden. Dabei ist seine Geschwindigkeit zu drosseln, so dass er vor dem roten Signal am Ende von IC_ST_3 zum Stehen kommt.

```
1  int main(int argc, char *argv[])
2  {
3     struct railway_system *railway;
4     railway = railway_initsystem(&kicking);
5     railway_openlinks_udp(railway,"node%02i","/dev/ttyS0");
6     railway_startcontrol(railway,0,0);
7
8     setpoint(railway, 17, BRANCH);
9     setpoint(railway, 16, BRANCH);
10    setpoint(railway, 0, BRANCH);
11    setpoint(railway, 9, BRANCH);
12    setpoint(railway, 1, BRANCH);
13
14    settrack(railway, -1, FWD, 40);
15    setsignal(railway, -1, 1, GREEN);
16
17    while ((getcontact(railway,IC_LN_5,1,1) == 0)&&(railway_alive(railway))) {usleep(100);}
18
19    setpoint(railway, 20, BRANCH);
20
21    while ((getcontact(railway,IC_ST_3,0,1) == 0)&&(railway_alive(railway))) {usleep(100);}
22
23    settrack(railway, -1, FWD, 5);
24    setsignal(railway, -1, 1, RED);
25
26    while ((getcontact(railway,IC_ST_3,1,1) == 0)&&(railway_alive(railway))) {usleep(100);}
27
28    settrack(railway, -1, OFF, 0);
29
30    railway_stopcontrol(railway,1);
31    railway_closelinks(railway);
32    railway_donesystem(railway);
33 }
```

Abbildung 4.5.: C-Beispielcontroller

Die Zeilen 3-6 dienen bei der realen Anlage der Initialisierung der `railway`-Struktur und dem Hochfahren der Steuerung. In der Simulation wird diese mit selbigem Aufruf initialisiert. Der entsprechende Kommunikationsthread zur ModelGUI lauscht dann standardmäßig auf den Port `2020` oder auf einen in der Initialisierungsdatei angegebenen, anderen Port. Die ModelGUI muss sich mit genau dem selben Port verbinden.

In den Zeilen 8 bis 12 werden einige Weichen gestellt. Die Zeile 14 und 15 zeigen, wie alle Gleisabschnitte mit der Geschwindigkeit `40` (in Hauptfahrtrichtung) beschaltet und außerdem alle Signale auf `grün` gestellt werden können.

Das Warten auf die Auslösung eines Kontakts, demonstrieren die Zeilen 17, 21 und 26. Hier wird in Verbindung mit dem Kontaktauslösungsereignis gleichzeitig

noch der Verbindungszustand zur ModelGUI abgefragt. Falls diese gestoppt wird, soll der Controller sich auch beenden und nicht weiter auf die Auslösung warten.

Schließlich muss am Ende in Zeile 30 noch der Simulationsthread beendet werden. Ist in der Konfigurationsdatei die Option `WAITFORGUI` gesetzt, so wird die Funktion erst zurückkehren, sobald die ModelGUI die Verbindung terminiert hat. Andernfalls käme es seitens der ModelGUI zur Fehlermeldung, dass SCADE keine oder unvollständige Daten schickte, denn der Kommunikationsthread würde mit seiner Terminierung auch die Kommunikationsverbindung abbrechen.

4.2.9. Neuen SCADE-Code verwenden

Soll das SCADE-Modell verändert oder erweitert werden, so ist zu beachten, dass sich auch der von SCADE generierte C-Code ändern wird. Als Konsequenz daraus muss dann auch jedes Programm, welches die Simulation benutzt, respektive die „railway.h“ der Simulation importiert, neu kompiliert werden.

Für den gesamten Build-Prozess wird eine Make-Datei bereit gestellt.

Make lässt sich u.a. mit dem Argument `COPYFILES` aufrufen. Dann kopiert das Skript allen C-Code mit Headerdateien in das „RailwaySimuSCADE“ Verzeichnis, in dem sich die „railway.c“ befindet. Dafür muss evtl. der Quellordner in der Make-Datei angepasst werden. Er hat dort die Bezeichnung `SCADEPROJECTFOLDER` und sollte auf den Ordner des SCADE-Projekts verweisen. Der dort bei der Codegenerierung erstellte Unterordner „Simulation“ wird in der Make-Datei durch `SCADEPROFILE` festgelegt und enthält den generierten C-Code, der zu kopieren ist.

Weiterhin muss bei einer Änderung oder Ergänzung der vom Modell benutzten *imported C operator*s darauf geachtet werden, dass das Make-Skript diese direkt aus dem Projektverzeichnis herauskopiert, denn SCADE dupliziert die Dateien nicht noch einmal in den Ordner „Simulation“. Für den Build-Prozess wird der Quellcode dieser Operatoren allerdings benötigt.

Wichtig ist zudem, dass sich keine unbenutzten C-Dateien oder H-Dateien im Projektverzeichnis oder im Ordner mit dem generierten C-Code befinden, da diese sonst den Build-Prozess störten. Alle ausführbaren Dateien finden sich schließlich im „Executables“ Verzeichnis wieder.

Als Compiler wird hier der GCC verwendet, der sowohl unter Linux als auch Windows (Cygwin [2]) zur Verfügung steht. Auch die Make-Datei funktioniert unter beiden Betriebssystemen und macht das Simulationsinterface und die Simulation damit plattformunabhängig.

4.3. Die TCP/IP-Schnittstelle

Durch die implementierte C-Schnittstelle wurde bereits erreicht, dass nun in C geschriebene Controller sowohl zusammen mit der Simulation als auch mit der echten

Bahn lauffähig sind.

Wie einführend diskutiert wurde, ist ein weiteres Ziel dieser Arbeit, sowohl die Simulation als auch den Zugriff auf die reale Anlage weiteren Programmiersprachen zu öffnen. Um dieses zu erreichen, wurde mit TCP/IP eine sehr universelle Schnittstelle gewählt, an welcher alle Programmiersprachen ansetzen können, die TCP/IP unterstützen.

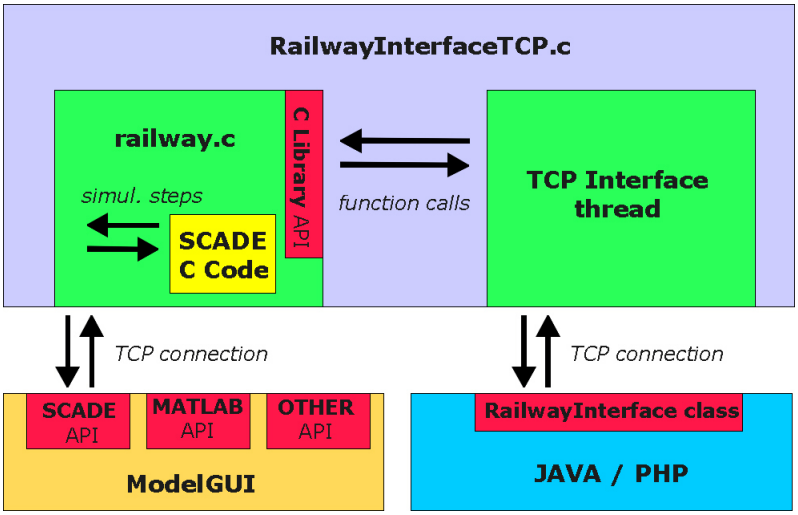


Abbildung 4.6.: Die TCP/IP-Schnittstelle (schematisch)

Dafür ist auf Seiten dieser Programmiersprachen nur noch eine entsprechende Funktionssammlung erforderlich, welche die Programmierung erleichtert und die TCP/IP-Kommunikation vor dem eigentlichen Controllerteil kapselt. Für JAVA und für PHP wurden entsprechende Klassen beispielhaft implementiert. Eine Erläuterung dieser findet in den noch folgenden Abschnitten 4.4.1 und 4.4.3 statt.

Zunächst soll hier die TCP/IP-Schnittstelle untersucht werden, die in der Abbildung 4.6 schematisch dargestellt ist. Vergleichend zu Abbildung 4.1 lässt sich erkennen, dass das Interfaceprogramm lediglich einen erweiterten C-Controller darstellt, welcher über eine zusätzliche TCP/IP-Schnittstelle verfügt.

Genau dies trifft auch zu und der offensichtliche Vorteil ist, dass hier genau die gleichen (minimalen) Änderungen erforderlich sind, um über dieses Interfaceprogramm zusätzlich auch die reale Bahnanlage anzusteuern, wie es der Abschnitt 4.3.3 beschreibt.

4.3.1. Interface Thread

Das Interfaceprogramm kommuniziert zum einen mit der ModelGUI über TCP/IP, wie dies auch jeder andere in C geschriebene Controller tun würde, der das Simulationsinterface benutzt. Zum anderen soll das Programm jedoch nicht selbst die

simulierte Modellbahn steuern, sondern einen weiteren TCP/IP-Zugang für einen externen Controller bereitstellen und dessen Kommandos bzw. die Antworten darauf lediglich entsprechend weiterleiten.

Dies bedeutet, dass hier ein zusätzlicher Kommunikationsthread benötigt wird, mit welchem sich ein externer Controller verbinden kann. Auch dieser ist als ein TCP/IP-Server implementiert und benutzt dabei die gleichen Basisroutinen, die auch für die Schnittstelle zur ModelGUI zum Einsatz kamen (s. Abschnitte 4.2.1 und 4.2.3).

Der Thread wird beim Starten des Programms in der `main()`-Funktion über den Aufruf von `initInterface()` erzeugt. Hier muss entsprechend zur Initialisierung der Simulation ebenfalls ein (anderer) gültiger und freier TCP/IP-Port übergeben werden.

Der in Abbildung 4.7 veranschaulichte Kommunikationsablauf stellt sich nach dem Etablieren des Sockets wie folgt dar:

1. Betreten einer Schleife, die nur durch `abortInterface()` verlassen werden kann.

2. Lauschen auf dem geöffneten Port.

3. Verbindungsaufbau von einem beliebigen, dieses Interface unterstützenden Controller als TCP/IP-Client

4. Betreten einer weiteren Schleife, in der kontinuierlich auf Kommandos von diesem Controller gewartet wird, um diese zu bearbeiten.

5. Kommando oder Kommandos empfangen.

6. Betreten einer innersten Schleife die alle empfangenen Kommandos abarbeitet.

7. Abarbeiten des ersten Kommandos und Löschen dessen aus allen noch zu bearbeitenden Kommandos.

8. Verlassen der Schleife falls fertig, sonst gehe zu Punkt 7.

9. Gehe zu Punkt 5.

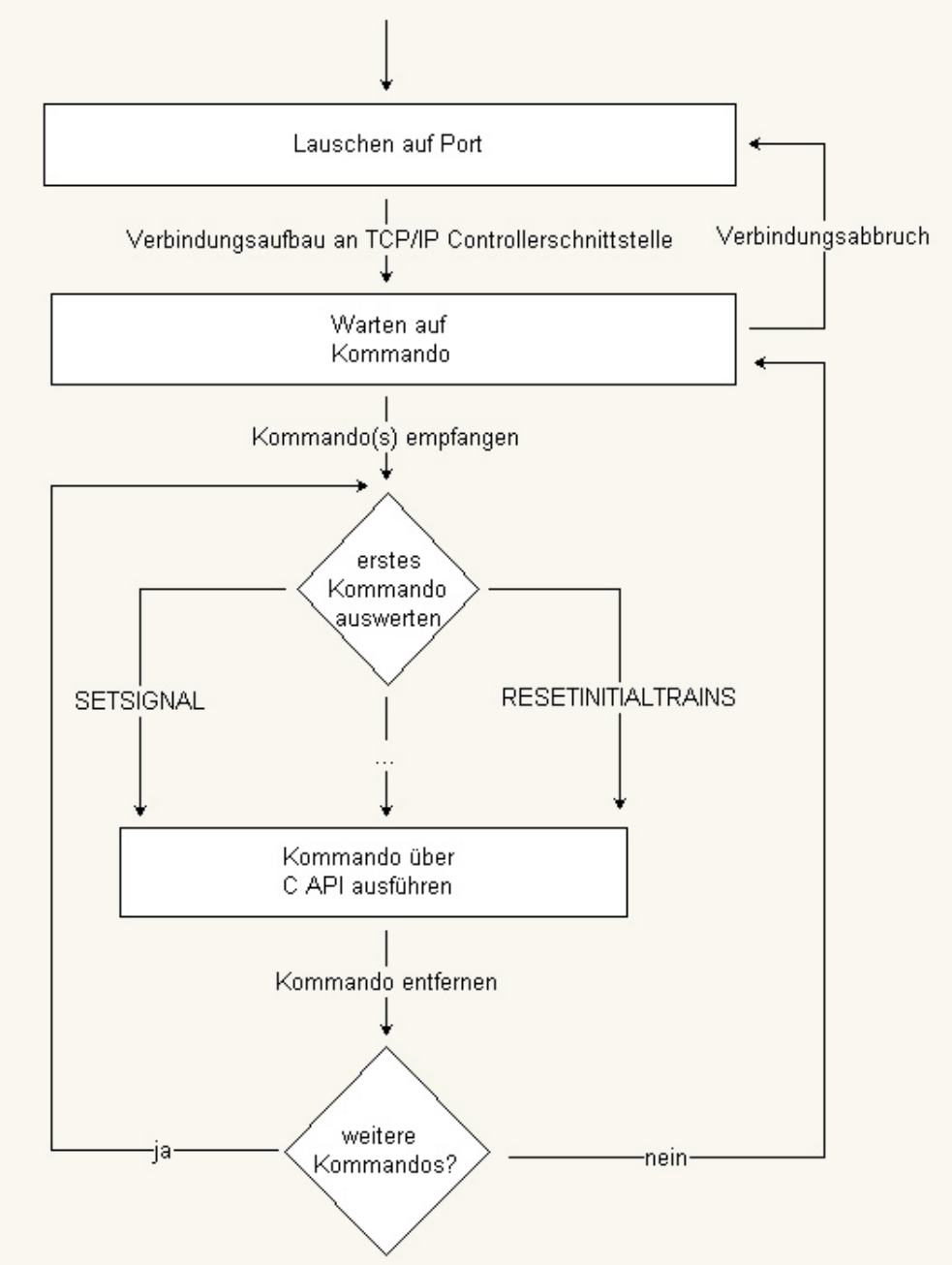


Abbildung 4.7.: Ablaufdiagramm für die Kommunikation mit TCP/IP-Controllern

Die TCP/IP-Verbindung wird solange gehalten, bis der Controller diese abbricht. Das führt aber noch nicht zur Beendigung des Threads. Dieser wartet vielmehr darauf, dass sich erneut ein Controller verbindet.

Das Hauptprogramm, bzw. der vordergründig laufende dritte Thread, bietet dem

Benutzer die Möglichkeit, beide hintergründig laufenden mit der Taste Enter abzubrechen. Dies beendet sowohl den Simulationsthread als auch den Interfacethread und schließlich das gesamte Interfaceprogramm.

```
<TCPcommand> := <command>\n

<command> := <SETSIGNALcmd> | <SETTRACKcmd> | <SETPOINTcmd> | <GETCONTACTcmd> | <TRACKUSEDcmd> |
        <SCANCONTACTcmd> | <GETSPEEDcmd> | <RESETINITIALTRAINScmd> | <RAILWAYALIVEcmd> |
             <RAILWAYSTOPCONTROLcmd> | <GETSIMULATIONTICKcmd> | <RESETSIMULATIONcmd> | <RESTARTSIMULATIONcmd> |
             <ABORTSIMULATIONcmd> | <ABORTINTERFACEcmd> | <SETINITIALTRAINcmd> | <SETINITIALTRAINEXcmd>

<SETSIGNALcmd> := SETSIGNAL<sep><para_block><sep><para_signal><sep><para_lights>
<SETTRACKcmd> := SETTRACK<sep><para_block><sep><para_mode><sep><para_target>
<SETPOINTcmd>  := SETPOINT<sep><para_point><sep><para_pointstate><sep>
<GETCONTACTcmd> := GETCONTACT<sep><para_block><sep><para_contact><sep><para_clearcontact>
<SCANCONTACTcmd> := SCANCONTACT<sep><para_block><sep><para_contact><sep><para_clearcontact>
<TRACKUSEDcmd> := TRACKUSED<sep><para_block><sep><sep>
<GETSPEEDcmd> := GETSPEED<sep><para_block><sep><sep>
<RAILWAYALIVEcmd> := RAILWAYALIVE<sep><sep><sep>
<RAILWAYSTOPCONTROLcmd> := RAILWAYSTOPCONTROL<sep><sep><sep>
<GETSIMULATIONTICKcmd> := GETSIMULATIONTICK<sep><sep><sep>
<RESETSIMULATIONcmd> := RESETSIMULATION<sep><sep><sep>
<RESTARTSIMULATIONcmd> := RESTARTSIMULATION<sep><sep><sep>
<ABORTSIMULATIONcmd> := ABORTSIMULATION<sep><sep><sep>
<ABORTINTERFACEcmd> := ABORTINTERFACE<sep><sep><sep>
<SETINITIALTRAINcmd> := SETINITIALTRAIN<sep><para_block><sep><sep>
<SETINITIALTRAINEXcmd> := SETINITIALTRAINEX<sep><para_block><sep><para_trainalign><sep>
<RESETINITIALTRAINScmd> := RESETINITIALTRAINS<sep><sep><sep>

<sep> := #

<para_signal> := <signalFIRST> | <signalSECOND>
<para_lights> := <signalRED> | <signalYELLOWGREEN> | <signalGREEN>
<para_block> := <blocks>
<para_mode> := <motorOFF> | <motorFWD> | <motorREV> | <motorBRAKE>
<para_target> := <numbers0to100>
<para_point> := <points>
<para_pointstate> := <pointSTRAIGHT> | <pointTURN>
<para_contact> := <contactFIRST> | <contactSECOND>
<para_clearcontact> := <clearNO> | <clearYES>
<para_trainalign> := <alignPRIMARY> | <alignSECONDARY>

<signalFIRST>       := 0
<signalSECOND>      := 1
<signalRED>         := 1
<signelYELLOWGREEN> := 2
<signalGREEN>       := 4
<motorOFF>   := 0
<motorFWD>   := 1
<motorREV>   := 2
<motorBRAKE> := 3
<pointSTRAIGHT> := 0
<pointTURN> := 1
<contactFIRST> := 0
<contactSECOND> := 1
<clearNO> := 0
<clearYES> := 1
<alignPRIMARY> := 0
<alignSECONDARY> := 1

<blocks> := 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
            19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 | 33 | 34 | 35 |
            36 | 37 | 38 | 39 | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47

<points> :=  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
            19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29

<numbers0to100> := 0 | <numberwithoutzero> [ <numberwithzero> ] | 100
<numberwithzero> := 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<numberwithoutzero> :=  1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Abbildung 4.8.: TCP/IP-Kommandos in EBNF-Notation

4.3.2. TCP/IP-Kommandos

Die von dem Controller zu sendenden Kommandos sind alle schematisch gleich aufgebaut: Sie bestehen aus einem Operationscode und darauf folgenden 3 Parametern, die nicht leer sein dürfen, jedoch nicht immer alle interpretiert werden. Das mit `#define TCPSEPARATOR '#'` definierte Zeichen trennt den Operationscode und die jeweiligen Parameter voneinander. Ein komplettes Kommando wird mit `'\n'` abgeschlossen. Die Funktion `SplitCommand()` wird dazu verwendet, aus mehreren Kommandos nur vom ersten die drei Parameter und den Operationscode zu extrahieren und anschließend dieses erste Kommando zu entfernen.

Im Folgenden seien alle derzeitig implementierten Kommandos vorgestellt, die in EBNF-Notation (Extended Backus-Naur Form) auch noch einmal in Abbildung 4.8 zu finden sind:

`SETSIGNAL#block#signal#lights\n` : Setzen von Signalen. Alle drei Parameter sind `Integer`-Werte. Der Parameter `block` bestimmt das Streckenelement, zu dem das Signal gehört; `signal` hat den Wert `0` oder `1`, je nachdem, ob es sich um das erste oder zweite Signal in Hauptfahrtrichtung handelt; `lights` bestimmt die Farben des Signals und ist ein `Integer`.

`SETTRACK#block#mode#target\n` : Setzen von Streckenabschnitten. Alle drei Parameter sind `Integer`-Werte. Der Parameter `block` bestimmt das zu setzende Streckenelement, `mode` bestimmt die Fahrtrichtung und `target` schließlich die Zielgeschwindigkeit.

`SETPOINT#point#state#\n` : Setzen von Weichen. Beide Parameter sind `Integer`-Werte, wobei `point` die Weiche bestimmt und `state` die Weichenstellung.

`GETCONTACT#block#contact#clear\n` : Auslesen von Kontaktereignissen. Alle Parameter sind erneut `Integer`-Werte. Der Parameter `block` bestimmt das Streckenelement, zu dem der Kontakt gehört; `contact` hat den Wert `0` oder `1`, je nachdem, ob es sich um den ersten oder zweiten Kontakt in Hauptfahrtrichtung handelt; `clear` gibt schließlich an, ob alle weiteren Kontaktereignisse für diesen Kontakt gelöscht werden sollen. Als Rückgabewert liefert dieses Kommando `NONE` für nicht ausgelöst, `FWD` für vorwärts ausgelöst, `REV` für rückwärts ausgelöst oder `UNI` für ausgelöst ohne feststellbare Fahrtrichtung.

`SCANCONTACT#block#contact#clear\n` : Entspricht dem vorherigen Kommando, mit dem Unterschied, dass der nächste passende Kontakt gesucht wird und zudem noch `block` und `contact` durch ein `#` getrennt zurückgegeben werden.

`TRACKUSED#block##\n` : Gibt an, ob ein Streckenabschnitt gerade von einer Lok befahren wird. Dabei ist `block` wieder ein `Integer`-Wert, der dem Streckenabschnitt entspricht. Rückgabewerte sind `0` für *frei* und `1` für *belegt*.

GETSPEED#block##\n : Gibt die gemessene Geschwindigkeit auf einem Streckenabschnitt zurück. Dabei ist `block` wieder ein `Integer`-Wert, der dem Streckenabschnitt entspricht.

RAILWAYALIVE###\n : Abfrage, ob das Modellbahnsystem bereit und funktionstüchtig ist bzw. bei der Simulation, ob die ModelGUI verbunden ist. Eine positive Antwort hat den Wert `1`, eine negative `0`.

RAILWAYSTOPCONTROL###\n : Beendet die Kommunikation zur Modellbahn bzw. bricht die Simulation ab.

GETSIMULATIONTICK###\n : Gibt den aktuellen Simulationsschritt zurück. Dieses Kommando funktioniert nur zusammen mit der Simulation und gibt andernfalls `-1` zurück.

RESETSIMULATION###\n : Setzt die Simulation zurück. Dieses Kommando funktioniert nur mit der Simulation zusammen.

RESTARTSIMULATION###\n : Startet die Simulation neu. Dieses Kommando funktioniert nur mit der Simulation zusammen.

ABORTSIMULATION###\n : Bricht die Simulation ab. Dieses Kommando funktioniert nur mit der Simulation zusammen.

ABORTINTERFACE###\n : Bricht die TCP/IP-Kommunikation zum Controller ab.

SETINITIALTRAIN#block##\n : Setzt einen initialen Zug auf dem Streckenabschnitt `block`. Dieses Kommando funktioniert nur mit der Simulation zusammen.

SETINITIALTRAINEX#block#alignment#\n : Setzt ebenfalls einen initialen Zug auf auf dem Streckenabschnitt `block` mit der Zugausrichtung (`alignment`) `0` für *in Hauptfahrtrichtung* und `1` für *in Gegenfahrtrichtung*. Dieses Kommando funktioniert nur mit der Simulation zusammen.

RESETINITIALTRAINS###\n : Setzt alle initialen Züge zurück. Dieses Kommando funktioniert nur mit der Simulation zusammen.

Diese einfach zu erweiternde Liste von Kommandos entspricht nicht allen, in der C-Bibliothek existierenden Funktionen, umfasst aber alle wichtigen Basisfunktionen, mit denen bereits komplexe Bahncontroller implementierbar werden.

4.3.3. Unterschiede zwischen Simulation und Modellbahn

Im vorangegangenen Abschnitt wurde bereits deutlich, dass es bei der Interpretation bzw. Ausführung der Kommandos Unterschiede zwischen der Simulation und der realen Anlage gibt. Diese Differenzen lassen sich teilweise bis zur C-Schnittstelle (vgl. dazu Abschnitt 4.2.6) zurückverfolgen. Dies betrifft weniger die Standardbefehle

zum Stellen von Weichen oder Festlegen von Fahrtgeschwindigkeiten sondern mehr diejenigen Befehle, welche zur Diagnose der Bahnanlage oder zur Verwaltung der Simulation (z.B. Setzen von Zügen) dienen.

Das Interfaceprogramm ist so konzipiert, dass es durch eine spezielle Präprozessordefinition sowohl für die Simulation als auch für die reale Modellbahn kompilierbar ist. Dazu muss lediglich die `#define SIMULATION` Anweisung modifiziert werden. Alle in Abschnitt 4.2.7 behandelten Änderungen werden bereits automatisch durch entsprechende Präprozessorregeln im Quelltext erledigt.

4.4. Verwendung von JAVA und PHP

Nachdem das TCP/IP-Schnittstellenprogramm genauer untersucht wurde, soll im Folgenden auf die Seite des Controllers eingegangen werden. Ein solcher Controller ist prinzipiell in jeder Programmiersprache denkbar, die TCP/IP unterstützt. Mit ihr lässt sich aus den in Abschnitt 4.3.2 vorgestellten Kommandos leicht eine eigene, in dieser Sprache abgefasste Funktionsbibliothek erstellen, die dann von einem solchen Controller verwendet werden kann.

In dieser Arbeit wurde eine solche Funktionssammlung beispielhaft für JAVA implementiert. Eine zusätzlich erstellte, eher experimentelle Bibliothek in der Skriptsprache PHP zeigt, dass sich das TCP/IP-Schnittstellenprogramm sehr universell verwenden lässt. Beide APIs und entsprechende Beispielcontroller werden im Folgenden vorgestellt.

4.4.1. Das JAVA-RailwayInterfacePackage

Für JAVA existiert dabei ein JAR-Paket, das sich wie gewohnt mit der `import`-Anweisung in das eigene JAVA-Projekt einbinden lässt.

In der Hauptklasse dieses Pakets werden zunächst einige Konstanten definiert, die die Lesbarkeit des Quelltextes implementierter Controller verbessern helfen sollen. Die Kommunikation wird über eine eigene Klasse `TAsyncCom` abgewickelt und im Konstruktor mit der übergebenen Hostadresse und dem Zielport hergestellt. Darauf folgt dann die Definition der Methoden, welche unter Benutzung der Kommunikationsinstanz, die entsprechenden Kommandos aus Abschnitt 4.3.2 an das Interfaceprogramm versenden und ggf. eine Antwort interpretieren und zurückgeben.

4.4.2. JAVA-Beispielcontroller

Obgleich die Verwendung des JAVA-Pakets relativ unkompliziert ist, soll an dieser Stelle ein, dem in C geschriebenen Beispielcontroller aus Abschnitt 4.2.8 entsprechendes JAVA-Programm (s. Abbildung 4.9) vorgestellt werden.

Dabei wird in Zeile 2 eine neue Instanz des JAVA-seitigen Simulationsinterfaces erstellt. Die Zeilen 4 bis 6 zeigen, wie auch vom Controller aus initiale Züge gesetzt und die Simulation beeinflusst werden kann. Würde das Programm statt der Simulation die reale Bahnanlage steuern, so ignorierte das Interfaceprogramm diese Kommandos. In den Zeilen 8 bis 12 werden dann erneut Weichen gesetzt. Daraufhin wird in Zeile 14 und 15 Fahrstrom auf alle Streckenabschnitte gegeben und alle Signale auf grün gestellt. Das Warten[17] auf eine Kontaktauslösung erfolgt beispielhaft in den Zeilen 17, 22 und 31. Auf eine verbundene oder nicht mehr verbundene ModelGUI muss an dieser Stelle nicht gewartet werden, da diese sich nur mit der Simulation und damit mit dem Interfaceprogramm verbindet. Es ist aber nötig, die Simulation nach dem Festlegen von neuen initialen Zügen zurückzusetzen, weil die Initialbelegung nur im ersten Simulationsschritt Berücksichtigung findet.

```
1  public static void main(String[] args) {
2     RailwayInterface RI = new RailwayInterface("localhost", 2020);
3
4     RI.ResetSimulation();
5     RI.SetInitialTrain(RI.IC_ST_1,false);
6     RI.RestartSimulation();
7
8     RI.SetPoint(17, RI.POINT_TURN);
9     RI.SetPoint(16, RI.POINT_TURN);
10    RI.SetPoint(0, RI.POINT_TURN);
11    RI.SetPoint(9, RI.POINT_TURN);
12    RI.SetPoint(1, RI.POINT_TURN);
13
14    RI.SetTrack(RI.ALL_TRACKS, RI.MOTORMODE_PRIMARY, 40);
15    RI.SetSignal(RI.ALL_TRACKS, RI.ALL_SIGNALS, RI.SIGNAL_GREEN);
16
17    while ((RI.GetContact(RI.IC_LN_5, 1, true) == 0)&&
18          (RI.RailwayAlive()));
19
20    RI.SetPoint(20, RI.POINT_TURN);
21
22    while ((RI.GetContact(RI.IC_ST_3, 0, true) == 0)&&;
23          (RI.RailwayAlive()));
24
25    RI.SetSignal(RI.ALL_TRACKS, RI.ALL_SIGNALS, RI.SIGNAL_RED);
26    RI.SetTrack(RI.ALL_TRACKS, RI.MOTORMODE_PRIMARY, 5);
27
28    while ((RI.GetContact(RI.IC_ST_3, 1, true) == 0)&&;
29          (RI.RailwayAlive()));
30
31    RI.SetTrack(RI.ALL_TRACKS, RI.MOTORMODE_OFF, 0);
32 }
```

Abbildung 4.9.: JAVA-Beispielcontroller

Vor dem Starten des Controllers muss auch das Interfaceprogramm aus Abschnitt 4.3 bereits ausgeführt werden. Eine Verbindung bei der Instantiierung ist andernfalls nicht möglich. Es ist weiterhin darauf zu achten, dass entsprechende Firewall-Einstellungen (des Betriebssystems) einen Verbindungsaufbau nicht verhindern.

[17] Ein Sleep-Befehl von 50ms, der die Prozessorauslastung in dieser *busy loop* ([10], S.225) begrenzen soll, ist bereits in der Klasse des Pakets implementiert und kann ggf. angepasst werden, falls dies nötig sein sollte.

4.4.3. PHP-RailwayInterfaceClass und PHP-Beispielcontroller

Die in PHP implementierte Klasse wird z.B. mit mit dem `require`-Kommando in ein PHP-Skript eingebunden. Die Klasse ist eher experimenteller Natur, da sie weniger praktische Relevanz hat, als dies beim JAVA-Paket der Fall ist. Der grundsätzliche Aufbau beider ist jedoch gleich.

Auch für diese Klasse existiert ein Beispielcontroller (s. Abbildung 4.10), der erneut die selben Aufgaben erledigt, wie der in C und JAVA besprochene Controllerquelltext.

```
1  $RI = new RailwayInterface("localhost","2020",false);
2
3  while(!($RI->RailwayAlive())){sleep(1);};
4
5  $RI->SetPoint(17, RailwayInterface::POINT_TURN);
6  $RI->SetPoint(16, RailwayInterface::POINT_TURN);
7  $RI->SetPoint(0, RailwayInterface::POINT_TURN);
8  $RI->SetPoint(9, RailwayInterface::POINT_TURN);
9  $RI->SetPoint(1, RailwayInterface::POINT_TURN);
10
11 $RI->SetTrack(RailwayInterface::ALL_TRACKS, RailwayInterface::MOTORMODE_PRIMARY, 40);
12 $RI->SetSignal(RailwayInterface::ALL_TRACKS, RailwayInterface::ALL_SIGNALS, RailwayInterface::SIGNAL_GREEN);
13
14 while (($RI->GetContact(RailwayInterface::IC_LN_5, 1, true) == 0)&&($RI->RailwayAlive()));;
15
16 $RI->SetPoint(20, RailwayInterface::POINT_TURN);
17
18 while (($RI->GetContact(RailwayInterface::IC_ST_3, 0, true) == 0)&&($RI->RailwayAlive()));
19
20 $RI->SetSignal(RailwayInterface::ALL_TRACKS, RailwayInterface::ALL_SIGNALS, RailwayInterface::SIGNAL_RED);
21 $RI->SetTrack(RailwayInterface::ALL_TRACKS, RailwayInterface::MOTORMODE_PRIMARY, 5);
22
23 while (($RI->GetContact(RailwayInterface::IC_ST_3, 1, true) == 0)&&($RI->RailwayAlive()));
24
25 $RI->SetTrack(RailwayInterface::ALL_TRACKS, RailwayInterface::MOTORMODE_PRIMARY, 0);
26
27 while($RI->RailwayAlive()){sleep(1);};
28
29 $RI->AbortInterface();
```

Abbildung 4.10.: PHP-Beispielcontroller

5. Ergebnisse

In dieser Arbeit kamen verschiedene Entwicklungswerkzeuge und Programmiersprachen zum Einsatz. Allen voran die SCADE Suite als grafische Modellierungssoftware nach dem synchronen Paradigma [12], ursprünglich basierend auf der textuellen und synchronen Datenflusssprache Lustre [21]. Es wurde weiterhin die Sprache C mit dem GCC als Compiler für den generierten C-Code als auch für das zusätzlich implementierte Interface verwendet und schließlich JAVA und PHP als Beispiel-Sprachen für die universelle TCP/IP-Controller-Schnittstelle.

Aus den in dieser Studienarbeit gesammelten Erfahrungen sollen in diesem Kapitel resümierend die wichtigsten festgehalten werden. Ein Vergleich zwischen den eingesetzten Werkzeugen und Sprachen wurde bereits anfangs in Abschnitt 2.2 angestellt. Dem schließt sich im Folgenden noch ein Ausblick auf mögliche Erweiterungen für die Simulation und ein für den Anwendungsbereich abschließendes Fazit an.

5.1. Erweiterungen

Neben der Entwicklung eines SCADE-Modells für die Simulation, sollte diese in der vorliegenden Arbeit auch für C und andere Programmiersprachen nutzbar gemacht werden. Dafür wurden entsprechende Schnittstellen und Interfaceprogramme implementiert.

Obwohl die entwickelte Simulation bereits alle wesentlichen Eigenschaften besitzt, um die in Abschnitt 2.1 dargestellten Vorzüge zu erhalten, lässt sich eine Simulation oft immer weiter verbessern. Die Möglichkeiten einer grundsätzlichen Erweiterbarkeit des Modells wurden in vorangegangenen Kapiteln bereits beschrieben. An dieser Stelle sollen einige konkrete Verbesserungsvorschläge exemplarisch aufgezeigt werden. Durch diese könnte die Simulation noch realistischer werden und so das Verhalten der echten Bahnanlage noch wirklichkeitsnäher modellieren.

Realistisches Anfahren und Abbremsen : Die Simulation berücksichtigt zwar die verschiedenen Fahrtrichtungen, jedoch wird ein Zug bei `STOP` oder `BRAKE` sofort angehalten. Ein Ausrollen oder langsames Abbremsen, wie es auf der realen Bahnanlage der Fall wäre, gibt es im Modell bisher nicht. Es sollte bei einer Erweiterung darauf geachtet werden, dass die Zugrichtung beim Rollen auch nach Gleiselement-Übergängen bei einer Kontaktauslösung korrekt ausgegeben wird. Hier können ggf. die lokalen Daten entsprechend erweitert werden.

Bahnübergang : Der Bahnübergang ist zwar bereits in der Simulation vorhanden, es fehlt aber bisher eine realistische Modellierung des Schrankenverhaltens mit

den dazugehörigen Kontakten. Eine solche Änderung muss dann gleichzeitig auch in der C-Schnittstelle Berücksichtigung finden.

Streckenabschnitte ohne Kontakte : Das Modell selbst unterscheidet bisher nicht zwischen Streckenabschnitten die Kontakte besitzen und jenen, auf denen keine installiert sind. Lediglich das Interface berücksichtigt diesen Punkt. Eine Erweiterung des Modells wäre hier für die Zukunft aber zweckmäßiger. Dort könnte bei einer Unterscheidung dann auch ein Ablehnen initialer Züge, die auf solchen Gleisabschnitten keinen Platz haben, stattfinden.

Variable Zuglängen : Die Länge der Züge ist in der Simulation bisher fix auf eine Maximallänge beschränkt. Bei einer Weiterentwicklung wäre erneut die lokale Datenstruktur, die zwischen zwei Streckenelementen ausgetauscht wird, zu erweitern.

Lampen : Die auf der echten Modellbahn installierten Lampen finden sich derzeitig noch nicht im Simulationsmodell wieder. Dies wäre eine Erweiterung, welche sich hauptsächlich auf die `controllerCommands`-Struktur und die ModelGUI-Dateien bezieht.

Erweiterte Funktionen : Die für die Modellbahn existierende C-Bibliothek stellt neben weiteren `Get()`-Funktionen, mit denen sich Zustände von Weichen u.ä. wieder abfragen lassen, auch noch zusätzliche Diagnosefunktionen bereit. Die C-Schnittstelle enthält für die meisten dafür bereits Dummyfunktionen, die, falls nötig, mit Leben gefüllt werden können. Hierbei müsste die Schnittstelle (oder auch das Modell) noch um ein *Gedächtnis* erweitert werden, das in der Lage ist, sich die erwähnten Zustände zu *merken*.

5.2. Fazit

In dieser Studienarbeit wurde eine komplette Simulation für die Modellbahnanlage der Informatik geschaffen. Diese ist durch den modellbasierten Ansatz und den modularen Aufbau des Simulationsmodells für zukünftige Modifizierungen oder Erweiterungen, auch an der realen Bahnanlage, bestens gerüstet.

Es ist außerdem eine Controller-Schnittstelle in Anlehnung an die existierende Modellbahn-Bibliothek in C entstanden, mit welcher ein Simulationslauf zusammen mit (auch bereits bestehenden) C-Controllern ohne Modifikationen am Code möglich wird. Es lassen sich damit in C geschriebene Controller zunächst mit der Simulation testen und später auf der realen Bahnanlage ausführen. So kann die Bahnhardware vor Beschädigungen geschützt werden und die Entwicklung und das Testen des Controllers ist örtlich nicht mehr an die Bahnanlage gebunden. Eine solche Code-Entwicklung kann nunmehr auch vielfach parallel stattfinden, wie dies z.B. in von der Universität angebotenen Softwarepraktika ist.

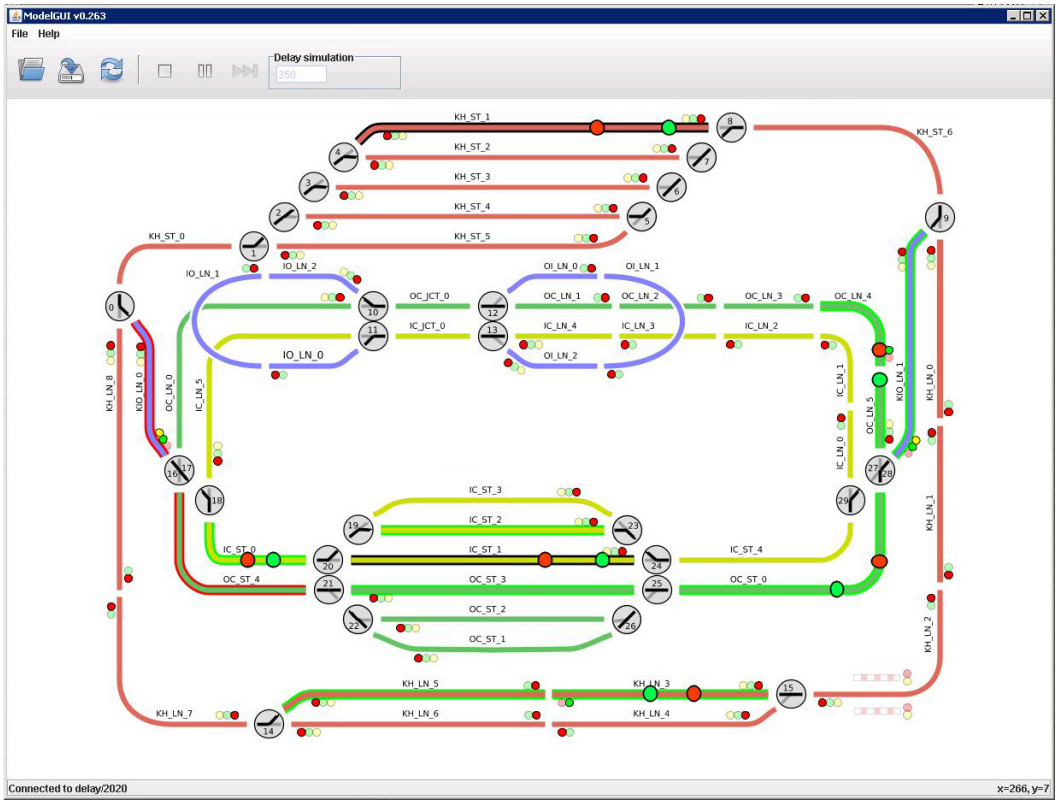


Abbildung 5.1.: Die laufende Simulation in der ModelGUI

Schließlich wurde noch einer TCP/IP-Schnittstelle entworfen, mit welcher sowohl die Simulation als auch die Modellbahnanlage weiteren Programmiersprachen zugänglich gemacht werden. Für diese Schnittstelle existiert bereits beispielhaft eine JAVA-Klasse, die die Steuerung der Simulation und der Modellbahn von einem in JAVA geschriebenen Controller aus ermöglicht.

Während des Entwicklungsprozesses mussten verschiedene Designentscheidungen getroffen werden, die sich gegen Ende jedoch alle als richtig herausstellten. Auch die zahlreichen Tests der erstellten Programme und Schnittstellen, sowie des generierten Simulationsmodells, verliefen durchweg erfolgreich. Die Arbeit mündet somit in einer realistischen, vielseitig einsetzbaren Simulation der Modellbahnanlage und versteht sich als gute Grundlage für weitere Projekte rund um diese herum.

A. Code-Generierung

Im Folgenden Abschnitt soll skizziert werden, wie von SCADE aus dem Modell der Bahnanlage kompilierbarer C-Code generiert werden kann.

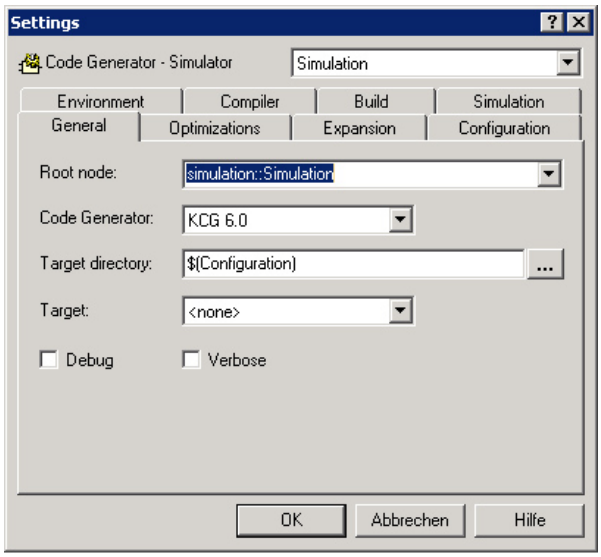


Abbildung A.1.: Die *Code Generator Settings* von SCADE

Dazu müssen im *Code Generator Settings*-Dialog, der in Abbildung A.1 zu sehen ist, einige Einstellungen vorgenommen werden:

1. Zunächst wird ganz oben rechts im Dialogfenster der Codegenerator „Simulation“ ausgewählt.
2. Unter der Karteikarte *General* muss dann der Root-Knoten auf „simulation::Simulation“ eingestellt werden.
3. Unter der Karteikarte *Expansion* sind schließlich verschiedene Operatoren auszuwählen, die SCADE bei der Überprüfung auf Zyklenfreiheit hin expandieren soll:
   - simulation::cycleBreaker
   - simulation::dataAll2Local
   - simulation::dataAll2LocalOnly
   - simulation::dataLocal2All

- simulation::innerCircle
- simulation::kickingHorsePass
- simulation::outerCircle

4. Als Compiler lässt sich unter der Karteikarte *Build* der GCC/GNU C einstellen. Da zunächst nur Code generiert werden soll, ist diese Einstellung optional und nur für einen evtl. folgenden Simulationslauf (s. Abschnitt 2.5) von Bedeutung.

Durch einem Klick auf „Generate“ kann letztlich C-Code aus dem Modell generiert werden, der sich anschließend im Unterverzeichnis „Simulation“ des Projektordners befindet.

B. Kurzreferenz

In den folgenden Abschnitten soll eine kompakte und praktisch orientierte Anleitung von der Erstellung bis hin zur Verwendung implementierter C- oder JAVA-Controller für die Modellbahn bzw. ihrer Simulation gegeben werden.

B.1. C-Controller

B.1.1. Implementierung

Die Implementierung eines C-Controllers erfolgt dem Beispiel des Controllers in Abschnitt 4.2.8 entsprechend. Zunächst muss das System der realen Anlage wie auch die Simulation initialisiert werden, und schließlich muss beides auch wieder heruntergefahren bzw. beendet werden. Die entsprechenden Codezeilen sind im oben genannten Abschnitt entsprechend kommentiert. Zur späteren Ansteuerung der realen Bahnanlage sind die Module „railway.h“ und „kicking.h“ mit der `#include`-Direktive einzubinden; bei der Simulation reicht hierbei die „railway.h“ aus.

B.1.2. Kompilieren

Soll der C-Controller die Simulation ansteuern, so wird er zusammen mit der Simulation kompiliert. Heraus kommt dann eine einzige, eigenständig ausführbare Programmdatei, die sowohl den vordergründigen Controllerthread als auch einen später hintergründig laufenden Simulationsthread enthält. Für den Build-Prozess existiert ein Make-Skript, welches z.B. entsprechend der Datei „apidemo.c“ um weitere C-Controller erweitert werden kann und im Abschnitt 4.2.9 vorgestellt wurde.

B.1.3. Verwendung

Um die Simulation auszuführen, muss die kompilierte Programmdatei des Controllers gestartet werden. Sie enthält einen zweiten Thread, der die Simulation darstellt und im Hintergrund von der ModelGUI aus gesteuert wird. Die Verbindung zwischen ModelGUI und Simulationsthread erfolgt über TCP/IP. Hier muss in beiden Programmen der gleiche Kommunikationsport angegeben werden. Bei der ModelGUI ist dies über einen Einstellungsdialog möglich. Der Simulationsthread versucht beim Starten eine Initialisierungsdatei im aktuellen Verzeichnis (oder aus dem Suchpfad) zu lesen, die, wie in Abschnitt 4.2.7 beschrieben, aufgebaut ist. Ist in dieser Datei kein Port festgelegt, oder eine solche Datei gar nicht auffindbar, so wird der Standardport `2020` verwendet. Es darauf zu achten, dass evtl. Firewalleinstellungen (Linux:

„System->Administration->Security Level and Firewall“, Windows: „Windows Security Center->Windows Firewall“) deaktiviert sind, oder eine Verbindung zumindest nicht verhindern. Die ModelGUI und die zu öffnende SVG-Datei befinden sich im Verzeichnis „Executables“. Nach dem Starten der ModelGUI und des Controllers, welcher auf die ModelGUI warten sollte (s. Abschnitt 4.2.7), kann, nach dem Öffnen der SVG-Datei in der ModelGUI, die Verbindung über den Button *Play* hergestellt werden. Jetzt sollte der Controller die durch die ModelGUI visualisierte Simulation steuern, als wäre es die echte Modellbahnanlage.

Es folgt noch ein Hinweis zum Beenden des Controllerthreads. Wird dieser Thread abgebrochen, ohne dass er den Simulationsthread bzw. das Programm beendet, so ist dieser anschließend noch hintergründig aktiv und muss über entsprechendes „kill“-Kommando des Betriebssystems manuell gestoppt werden. Wird dies versäumt, so kann der belegte Port nicht für weitere Instanzen des Controllerprogramms verwendet werden.

B.2. JAVA-Controller

B.2.1. Implementierung

Die Implementierung eines JAVA-Controllers erfolgt dem Beispiel des Controllers in Abschnitt 4.4.2 entsprechend. Als Unterschied zu dem zuvor betrachteten C-Controller, bleiben hierbei Controller und Simulation voneinander getrennt und kommunizieren zusätzlich über TCP/IP. Dazu muss ein entsprechender Port bei Erzeugung einer Railway-Schnittstellen-Klasseninstanz angegeben werden. Diese Klasse liegt sowohl als Quelltext als auch als JAR-Paket vor.

B.2.2. Kompilieren

Das Kompilieren der JAVA-Dateien des Controllers kann über einen einfachen Aufruf von „javac controller.java“ geschehen. Neben dem Controller muss jedoch auch das Schnittstellenprogramm, in welchem sich auch die eigentliche Simulation verbirgt, kompiliert vorliegen. Dieses ist in den automatischen Build-Prozess inkludiert (s. Abschnitt 4.2.9) und kann über den Aufruf von „make all“ im Verzeichnis „Executables“ neu erstellt werden.

B.2.3. Verwendung

Bei der Verwendung eines JAVA-Controllers muss zunächst immer das Schnittstellenprogramm, dass sich im „Executables“-Verzeichnis befindet, aufgerufen werden. Dieses erlaubt die Angabe eines TCP/IP-Ports für den JAVA-Controller und eine optionale Angabe eines weiteren Ports für die ModelGUI. Wird auf das zweite Argument verzichtet, dann übernimmt das Schnittstellenprogramm diesen erneut aus der in Abschnitt 4.2.7 vorgestellten Initialisierungsdatei für die Simulation. Auch alle übrigen Einstellungen, z.B. zur Windows2000-Kompatibilität, können hierin vor-

genommen werden. Diese Datei muss sich im aktuellen Verzeichnis, aus dem das Schnittstellenprogramm heraus aufgerufen wird, oder im Suchpfad befinden.

Nachdem das Schnittstellenprogramm gestartet ist und auf zwei Ports lauscht, sollte zunächst die ModelGUI und anschließend das JAVA-Controllerprogramm gestartet werden. Es muss bei Verbindungsproblemen darauf geachtet werden, dass die gewählten Ports frei sind und der richtige Hostname eingetragen ist. Bei *loopback*-Verbindungen auf dem gleichen Rechner ist dies entweder `127.0.0.1` oder `localhost`. Wird das Schnittstellenprogramm mehrfach kurz hintereinander mit den selben Ports neu gestartet, kann es dabei zu Problemen durch vom Betriebssystem vorgegebene Timeouts kommen.

Es existieren weiterhin drei kleine Stapelverarbeitungsprogramme zum Aufruf der ModelGUI („gui.bat“), zum Aufruf des Interfaceprogramms („interface.bat“) mit den Port-Optionen `2222` für das GUI und `2020` für das Java-Programm und schließlich noch für den eines Java-Beispiel-Controllers („java.bat“). Um diese alle von einer Konsole aus im Hintergrund zu starten, ist den Befehlen ein „&“ anzuhängen.

Das Schnittstellenprogramm unterstützt die Eingabe eines oder zweier Ports. Der erste ist unbedingt erforderlich für die Verbindung zum Controller. Der zweite wird für die ModelGUI-Verbindung benötigt, ist jedoch optional. Wird er nicht mit angegeben, so verwendet das Schnittstellenprogramm den in der Initialisierungsdatei angegebenen Port. Wird das Programm mit nur einem Port aufgerufen und ist die Option `WAITFORGUI` in dieser Datei aktiv, dann wartet das Interfaceprogramm während der Initialisierung der Simulation auf die Verbindung der ModelGUI. Vorher ist auch eine Verbindung zum Controller nicht möglich. Um dies zu verhindern, kann entweder die Option abgestellt werden, oder es werden beim Aufruf des Schnittstellenprogramms beide Ports mit angegeben.

C. Inhalt der CD-ROM

Name	Beschreibung	Verzeichnis
SCADE-Modell Projekt	Das mit der SCADE-Version FSCa kompatible Projekt, in welchem das Simulationsmodell inklusive Dokumentation gespeichert ist	// Programmcode / ScadeSource / simulationsa /
SCADE-Modell C-Code	Der von SCADE mit dem Profil *Simulation* generierte C-Code im gleichnamigen Unterverzeichnis	// Programmcode / ScadeSource / simulationsa / Simulation /
C-Schnittstelle	Die Dateien „railway.h“ und „railway.c“, die als Simulation zusammen mit dem generierten C-Code die gleiche Funktionalität bereitstellen, wie die bestehende C-Bibliothek der echten Modellbahn	// Programmcode / RailwaySimuSCADE /
C-Controller	Mit den Dateien „apidemo.c“, „RailwayControllerSample.c“ und „SampleController.c“ verschiedene Beispiel-Implementierungen von C-Controllern für die Simulation	// Programmcode / RailwayControllerC /
TCP-Schnittstelle	Das Interfaceprogramm zur Bereitstellung einer TCP/IP-Schnittstelle für die Simulation und die Modellbahnanlage mit der Quelltextdatei „RailwayInterfaceTCP.c“	// Programmcode / RailwayInterfaceTCP /
JAVA-Schnittstelle und Controller	Das JAR-Paket „railwayInterfacePackage.jar“ als einzubindende Schnittstelle und „SampleController.java“ als Beispiel-JAVA-Controller	// Programmcode / RailwayControllerJAVA /
PHP-Schnittstelle und Controller	Die einzubindende Schnittstellenklasse „RailwayInterfaceClass.php“ und „SampleController.php“ als Beispiel-PHP-Controller	// Programmcode / RailwayControllerPHP /
Start-Skripte	Die vier Stapelverarbeitungsprogramme „gui.bat“, „interface.bat“, „java.bat“ und „c.bat“	// Programmcode /

Tabelle C.1.: Inhalt der beiliegenden CD-ROM

Name	Beschreibung	Verzeichnis
ModelGUI	Die ausführbare JAR-Datei „modelgui-6b1.jar“ zur Visualisierung	// Programmcode / Executables /
SVG-Datei	Vektorgrafik der Simulation für die ModelGUI in der Datei „railway-6b1_fullsimsa.svg“	// Programmcode / Executables /
MAP-Datei	Mapping der Simulation für die ModelGUI in der Datei „railway-6b1_fullsimsa.map“	// Programmcode / Executables /
zusätzliche Modifier	Für einen SCADE-Controller benötigter C-Code	// Programmcode / SCADEAdditionalCModifiers /
Initialisierungsdatei	Die Datei „railway.cnf“ für die Optionen und initialen Züge der Simulation	// Programmcode /
Dokumentation	Die Datei „refman.pdf“ als Referenz zum Programmcode	// Documentation / latex /
Ausarbeitung	Diese Ausarbeitung in der Datei „Studienarbeit.pdf“	// Ausarbeitung /

Tabelle C.2.: Inhalt der beiliegenden CD-ROM (Fortsetzung)

D. Literaturverzeichnis

[1] Controller Area Network - CAN in Automation, . URL www.can-cia.org/.

[2] Cygwin Project, . URL www.cygwin.com/.

[3] Inkskape Project, . URL www.inkscape.org/.

[4] Kiel Integrated Environment for Layout, . URL www.informatik.uni-kiel.de/~rt-kiel/.

[5] Matlab/Simulink, . URL www.mathworks.com/.

[6] ModelGUI Project, 2007. URL rtsys.informatik.uni-kiel.de/wiki/index.php/ModelGUI.

[7] Model Railway 2007, 2007. URL rtsys.informatik.uni-kiel.de/cgi-bin/trac.cgi/wiki.

[8] Real-Time Workshop, . URL www.mathworks.com/products/rtw/.

[9] Charles André. Syncharts: a visual representation of reactive behaviors. Technical Report RR 95–52, rev. RR (96–56), I3S, Sophia-Antipolis, France, Rev. April 1996.

[10] Alan Burns and Andy Wellings. *Real-Time-Systems and Programming Languages*. Addison Wesley, 2001.

[11] Esterel Technologies. SCADE Suite. URL www.esterel-technologies.com/products/scade-suite/.

[12] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[13] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987. URL citeseer.ist.psu.edu/harel87statecharts.html.

[14] Chris Hills. Embedded c - traps and pitfalls. *ESS Conference, Olympia, London*, May 2000.

[15] Institute of Electrical and Electronics Engenieers. Ieee 802.3 csma/cd (ethernet). URL www.ieee802.org/3/.

[16] G. Kahn. The semantics of a simple language for parallel programming. August 1974.

[17] Hermann Kopetz. *Real-Time Systems*. Kluwer Academic Publishers, 1998.

[18] Edward A. Lee and Thomas M. Parks. Dataflow process networks. pages 773–799, May 1995. URL `citeseer.ist.psu.edu/lee95dataflow.html`.

[19] Stephan Hoermann. Homepage der Modellbahnanlage. URL `www.informatik.uni-kiel.de/~railway/`.

[20] Stephan Hoermann. Diplomarbeit zur Modellbahn, 2006. URL `www.informatik.uni-kiel.de/~railway/Downloads/hoehrmann2.pdf`.

[21] The Synchronous Group. Lustre. URL `www-verimag.imag.fr/SYNCHRONE/`.

[22] TTTech. Time-Triggered Technology. URL `www.tttech.com/`.

Index

E

F

G

H

I

P

Q

R

S

T

U

V

W

X

Z