Universität Bremen

Bachelor Thesis
zur Erlangung des akademischen Grades

Bachelor of Science

Model Checking eines Stellwerksalgorithmus mit FDR4

Felix Brüning

Matrikelnummer: *4348078*

Erstgutachter:
Prof. Dr. Jan Peleska

Zweitgutachter:
Prof. Dr. Rolf Drechsler

Bremen, den 13. Februar 2020

Inhaltsverzeichnis

Abbildungsverzeichnis **III**

Tabellenverzeichnis **IV**

Listingverzeichnis **V**

1 Einführung **1**
1.1 Praktischer Hintergrund . . . 1
1.2 Verwandte Arbeiten . . . 2
1.3 Zielsetzung . . . 2

2 Routen-basiertes, autonomes Zugbeeinflussungssystem **4**
2.1 Gleisnetz und Ortung . . . 5
2.2 Routen und Stellwerk . . . 6
2.2.1 Routen . . . 6
2.2.2 Stellwerk . . . 7

3 Model Checking mit FDR4 **14**
3.1 CSP - Syntax und Semantik . . . 14
3.1.1 Channels . . . 15
3.1.2 Prozesse . . . 16
3.1.3 Assertions . . . 17
3.1.4 Hiding . . . 18
3.2 FDR4 . . . 18
3.2.1 Übersetzung von CSP . . . 19
3.2.2 Refinement Checking . . . 19

4 Test und Verifikation **22**
4.1 Modellierung der Streckenabschnitte . . . 23
4.2 Model-in-the-Loop (MiL) Tests . . . 26
4.2.1 Kleines Gleisnetz . . . 27
4.2.2 Tests für das gesamte Gleisnetz . . . 31
4.3 Modell Validierung des gesamten Systems . . . 35
4.3.1 Sub-Controller Tests . . . 35
4.3.2 System-Tests . . . 38
4.3.3 Ergebnis der Tests . . . 43

5 Evaluation **44**
5.1 Testkonfiguration und Ergebnisse . . . 44
5.2 Genutzte Rechnerressourcen . . . 45

6 Fazit und Ausblick **47**

6.1 Vergleich FDR4 und nuXmv 47
6.2 Ausblick 49

Literatur **51**

Anhang **56**

Inhalt des Speichermediums **56**

Eidesstattliche Erklärung **57**

Abbildungsverzeichnis

1 Verhalten des Zugcontrollers . . . 2

2 Internal Block Diagramm vom System . . . 5
3 Gleisnetz der Märklin Eisenbahn . . . 5
4 Beispiel Interlocking-Table . . . 7
5 Architektur des Stellwerkes. [PHH16] . . . 8
6 Train-Controller Zustandsautomat . . . 9
7 Zustandsautomaten des Sub-Controllers . . . 11
8 Zustandsautomat des Safety-Monitors . . . 12

9 Benutzeroberfläche von FDR4 . . . 18

10 Schametische Darstellung einer Kreuzweiche . . . 23
11 Zustandsautomat einer Kreuzweiche . . . 24
12 Weichen . . . 25
13 Zustandsautomaten einer Weiche . . . 25
14 Zustandsautomat eines Track-Elementes . . . 26
15 Kleines Gleisnetz . . . 27
16 Fehler bei einer Weiche . . . 30
17 Test 1: Züge befahren die Route . . . 32
18 Testaufbau bei der Verifikation der Sub-Controller . . . 36
19 Testaufbau für Tests des gesamten Systems . . . 39

Tabellenverzeichnis

1 Benchmarkergebnisse 46

Listings

1 Beispiel Pattern-Matching . . . 16
2 Synchronisationsbeispiel . . . 16
3 Kommunizierende Prozesse . . . 17
4 Beispiel Code Refinement-Checking . . . 19
5 Ausgabe von FDR4 bezüglich des Bespiels . . . 21
6 Synchronisation der Streckenabschnitte . . . 27
7 Marked- und Locking-Zustand einer Weiche . . . 28
8 Init- und Train_from_left-Zustand . . . 29
9 Erster MiL Test . . . 29
10 Testergebnis vom kleinen Gleisnetz . . . 30
11 Testergebnis bei fehlerhafter Implementierung . . . 30
12 MiL-Test: Zwei Züge befahren das Gleisnetz . . . 32
13 Ergebnis MiL-Test . . . 33
14 MiL-Test: Zwei Züge fahren auf eine Weiche und kollidieren . . . 33
15 Ergebnis Test . . . 34
16 Falsches Abbiegen eines Zuges . . . 34
17 Ergebnis falsches Abbiegen . . . 35
18 Sub-Controller Test . . . 37
19 Ergebnis Subcontroller-Test . . . 37
20 Fehler im LOCKING Zustand . . . 38
21 Lösung des Fehlers . . . 38
22 Ergebnis mit neuem Modell . . . 38
23 CSP-Codeausschnitt aus den Tests . . . 40
24 Fehlgeschlagender Test . . . 41
25 Test Konfliktrouten . . . 41
26 Test mit Nicht-Konfliktrouten . . . 42
27 CSP_M Beispiel . . . 48
28 nuXmv Beispiel . . . 49
29 Weichen Implementierung . . . 53
30 Track-Element Implementierung . . . 54

Kapitel 1

Einführung

1.1 Praktischer Hintergrund

In der heutigen Zeit ist Software nicht mehr aus dem Alltag wegzudenken. Nicht nur im Bereich von Unterhaltungsmedien und mobilen Endgeräten nimmt die Digitalisierung zu, sondern auch beim Steuern und Überwachen von Anlagen und Maschinen. Wo früher Maschinen aufwendig von Menschenhand gesteuert und überwacht wurden, übernehmen heute Computer diese Arbeit. Sogenannte eingebettete Systeme, die zum Steuern und Überwachen von Maschinen und Geräten eingesetzt werden, finden sich heutzutage in vielen Bereichen der Industrie. In den heutigen Eisenbahnen, sowie Flugzeugen und Autos findet man viele von diesen Systemen. Diese müssen dabei eine hohe Sicherheit garantieren, um das Leben von Menschen nicht zu gefährden.

Doch die Anzahl der eingebetteten Systeme nimmt nicht nur zu, sondern führen auch immer komplexere Software aus, die aus bis zu mehreren Millionen Zeilen Programmcode bestehen kann. Um die immer komplexer werdende Software zu testen, bedarf es an immer mehr Rechenleistung. Hinzu kommt, dass komplexer werdende Software immer mehr Fehler enthält und dabei einen hohen Grad an Korrektheit und Sicherheit aufweisen muss.

Bei Systemen, wie beispielsweise aus der Eisenbahndomäne, folgt man beim Entwickeln eines Systems eine strenge Vorgehensweise: Zunächst muss die zu entwickelnde Software modelliert werden. Dazu wird die Systembeschreibungssprache SysML (System Modelling Language) [Gro17] genutzt. Beim Modellieren können schon gravierende Fehler entstehen, die die Sicherheit im realen Betrieb stark beeinträchtigen können. Um diese frühzeitig zu entdecken und somit den weiteren Entwicklungszyklus kosteneffizienter zu gestalten, wird das sogenannte Model Checking betrieben. Beim Model Checking wird das gegebene Modell auf Einhaltung der Sicherheitsspezifikationen geprüft, sodass die Sicherheit des modellierten Systems gewährleistet ist.

1.2 Verwandte Arbeiten

Model Checking in der Eisenbahndomäne haben sich viele Arbeitsgruppen als Forschungsthema gesetzt. So forschen Jan Peleska und Wen-Ling Huang der Arbeitsgruppe Betriebssysteme und verteilte Systeme (AGBS) der Universität Bremen in diesem Bereich. Das Paper mit dem Titel "Formal modelling and verification of interlocking systems featuring sequential release " zeigt dies ([HHP17]).

Hierbei wird untersucht, wie man für Stellwerke, die Sequencial Release zur Streckenfreigabe nutzen, Model Checking betreiben kann. Dabei wird das Verhalten beschrieben, wie Strecken allokiert, gesperrt und befahren werden, wie in Abbildung 1 zu sehen.

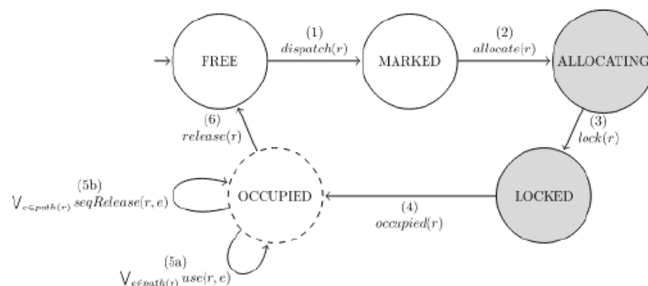


Abbildung 1: Verhalten des Zugcontrollers

Ebenso beschäftigen sich die Arbeitsgruppe aus Dänemark von Anne E. Haxthausen im Bereich der Eisenbahndomäne, wie in [HP15] zu sehen.
In diesem Paper wird gezeigt, wie Stellwerke mit variabler Konfiguration mithilfe von Model Checking geprüft werden können. Dabei wird der Fokus darauf gelegt, wie man das mathematisch umsetzt, sowie hinterher dadurch Model-Based-Testing (MBT) durchführen kann.

1.3 Zielsetzung

Im studentischen Projekt TEAMOD (Test- and Model Checking) vom Wintersemester 2018/19 bis Sommersemester 2019, wurde ein autonomes Zugbeeinflussungssystem unter der Leitung von Prof. Dr. Jan Peleska und Prof. Dr. Wen-Ling Huang entwickelt. [FB19] Dieses hat die zentralen Bestandteile:

- Ein zentrales autonom-entscheidendes Stellwerk,
- je ein Zugcomputer pro Zug zur Steuerung und Überwachung des Zuges mit autonomer Zielfestlegung,
- ein Zugortungssystem

- und ein Kommunikationssystem zur Datenübertragung via UDP.

Um die Sicherheit des Systems zu garantieren, wird in dieser Arbeit das zuvor spezifizierte und modellierte System mittels Modellprüfung verifiziert.
Dabei werden die entworfenen Modelle in der Prozessalgebra CSP_M modelliert und mittels FDR4 verifiziert. Zunächst wird das bestehende Gleisnetz modelliert und durch Model-in-the-Loop Tests verifiziert. Das soll sicherstellen, dass das modellierte Gleisnetz dem realen entspricht. Für den Test befahren Züge jedes Gleis und jede Weiche des Gleisnetzes.
Zunächst soll gezeigt werden, dass die Routen-(Sub-)Controller hinsichtlich Entgleisung und Kollisionen sicher sind. Durch geeignete Refinement-Checks wird dieses bewiesen.
Anschließend soll gezeigt werden, dass die generierte Interlocking-Table sicher ist. Dabei werden ebenfalls Model-in-the-Loop Tests verwendet, um die Sicherheit vor Entgleisung und Kollisionen zu prüfen. Um eine vollständige Testabdeckung zu erzielen, werden Konfliktrouten parallel befahren. Ebenso wird gezeigt, dass alle nicht im Konflikt stehende Routen keine Sicherheitsdefizite aufweisen.
Das Ziel ist es zu zeigen, dass das sichere Befahren des Gleisnetzes durch autonome Züge garantiert werden kann. Dabei wird FDR4 als Refinement Checker genutzt, um die Modelle gegen die Spezifikation zu testen.

Kapitel 2

Routen-basiertes, autonomes Zugbeeinflussungssystem

Im Wintersemester 2018/19, sowie im Sommersemester 2019 wurde im Rahmen des studentischen Projekts TEAMOD ein autonomes Bahnsystem entwickelt. Dabei ist es möglich, dass Züge autonom das Gleisnetz befahren können. Allgemein besteht ein Bahnsystem aus vier Komponenten: Die zentrale Komponente ist das Stellwerk. Dieses ist für das Stellen der Weichen, sowie das Schalten der Signale und für das Freigeben und Sperren von bestimmten Gleisabschnitten zuständig. Ebenso hat das Stellwerk die Aufgabe, die aktuell fahrenden Züge zu überwachen und bei einem Falschabbiegen oder einer drohenden Kollision den Nothalt dieser Züge auszulösen. Zudem hat es die Macht über alle Entscheidungen und greift im Notfall in das laufende System ein.
Jeder Zug hat dabei einen eigenen sogenannten Train-Control-Computer (kurz: TCC), der den Zug autonom steuert. Dieser sucht sich dabei autonom aus eine Menge von möglichen Zielen Ziele und fragt diese am Stellwerk an. Dieses reagiert darauf und der TCC regelt dabei die Zuggeschwindigkeit in Bezug zur verbleibenden Routen-Abschnitte. Ebenfalls kann dieser in einer Notsituation den Zug vollautomatisch abbremsen und so ein Entgleisen verhindern.
Als drittes System gibt es das Ortungssystem, welches permanent die aktuelle Position der Züge ermittelt. Diese Positionsdaten werden dann vom Zug an das Stellwerk gesendet, welches die Daten zur Kontrolle des Gleisnetzes benötigt.
Die vierte Komponente ist für die Kommunikation zuständig. Der sogenannte Router ist für das Verschicken und Empfangen von Datenpaketen über das UDP-Protokoll zuständig. Ebenso ist diese Schnittstelle eine zentrale Anlaufstelle für das Erreichen der Märklin Station. Die Märklin-Spezifischen UDP-Telegramme werden vom Router erzeugt, wodurch andere Komponenten lediglich Funktionen des Routers aufrufen müssen (Programmbibliothek). Das SysML Internal-Block-Diagram (kurz: IBD) aus Abbildung 2 zeigt diesen Aufbau.

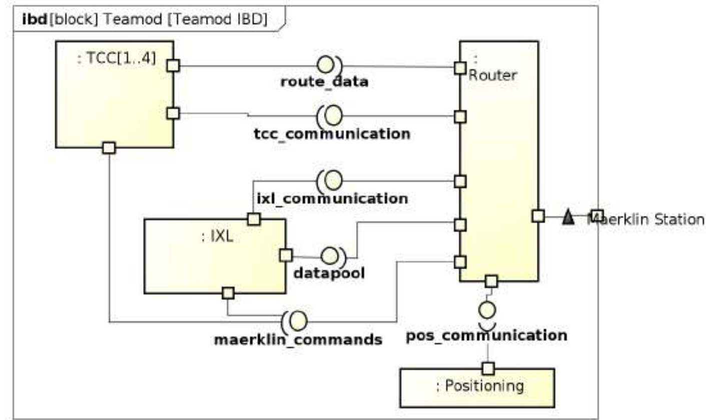


Abbildung 2: Internal Block Diagramm vom System

2.1 Gleisnetz und Ortung

Von der Arbeitsgruppe Betriebssysteme, verteilte Systeme von Jan Peleska und Wen-Ling Huang wurde ein vollständiges Märklin Gleisnetz zur Verfügung gestellt. Dieses war als Rundkurs mit Abstellgleisen konzipiert. Die folgende Abbildung zeigt das Gleisnetz schematisch:

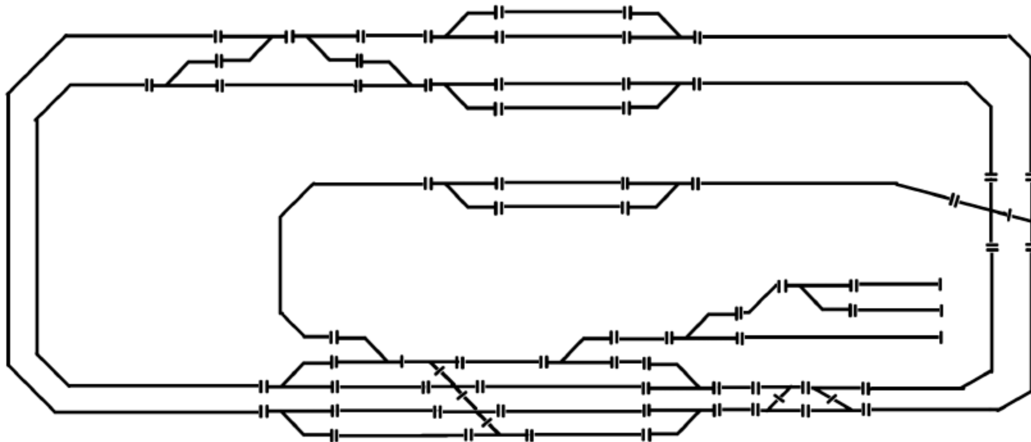


Abbildung 3: Gleisnetz der Märklin Eisenbahn

Die Züge auf dem Gleis werden von einer Märklin-Station (CS2) gesteuert, die den Zügen Befehle über die Schienen übermittelt. Diese dienen dabei ebenso als Stromquelle für die Züge. Da auf beiden Schienen der gleiche logische Strom fließt, wurde diese Eigenschaft genutzt, um daraus ein Zugortungssystem zu entwickeln. Dafür wurden an einigen Stellen Gleisstücke einseitig abisoliert, sodass der Strom nur noch über eine

Schiene eines Teilstückes fließt.
Befährt nun ein Zug dieses manipulierte Gleisstück, so leitet dieser, durch seine durchgängige Eisenachse, den Strom von der nicht-manipulierten Schiene zur manipulierten. Jede manipulierte Schiene ist dabei mit einem Raspberry Pi über ein Spannungsteiler Board verbunden.
Jeder der vier Raspberry Pis prüft dabei kontinuierlich den Wert, der an der angeschlossenen Schiene anliegt. Über einen gewissen Algorithmus wird der Durchschnittswert der anliegenden Spannung pro Gleisstück in einem bestimmten Zeitintervall gemessen. Dieser Algorithmus soll das auftretende Rauschen an dem Gleisstück mindern, oder sogar ganz entfernen, sodass immer, wenn ein Zug ein Gleisstück befährt, die aktuelle Zugposition präzise gemessen werden kann. Ist es nun der Fall, dass tatsächlich ein Zug detektiert wurde, so wird diese Information als Datenpaket vom Programm per Broadcast ins lokale Netzwerk (LAN) versendet. Dieses Datenpaket besteht dabei aus allen Identifikationsnummern der angeschlossenen Gleise und deren Belegungszustand. Das Versenden via Broadcast hat den Vorteil, dass sowohl das Stellwerk, als auch alle Zugsteuerungscomputer diese Ortungsinformation erhalten. Dadurch ist das separate Versenden an jeden Teilnehmer überflüssig und beschleunigt das Erfassen der Belegungszustände der Gleise. [FB19]

2.2 Routen und Stellwerk

Um einen sicheren, wie auch geordneten Eisenbahnbetrieb zu erreichen, wurde ein Routen-basierte Stellwerk entworfen. Das Stellwerk arbeitet nach der Art und Weise, dass das Gleisnetz in etwa identisch große Teilrouten unterteilt wird. Somit erlaubt dieses einem Zug zwischen zwei Punkten zu fahren und muss, wenn keine weitere Freigabe zur Weiterfahrt erteilt wurde, am Ende dieser Route zum Stehen kommen. Dabei ist es grundsätzlich auch möglich, dass sich Routen überschneiden.

2.2.1 Routen

Routen sind dabei Pfade auf dem Gleisnetz, die auf einem Track-Element beginnen und auf einem enden. Ein Track-Element ist dabei eine Schiene, die wie in Kapitel 2.1 beschrieben, die Position eines Zuges erfassen kann. Das hat den Vorteil, dass das Stellwerk den initialen Startpunkt des Zuges, sowie das Erreichen des Endpunktes konkret feststellen kann. Jedoch speichert das Stellwerk noch weitere Parameter einer Route, die für das sichere befahren der Route nötig sind. Dazu gehört zum einen die nötigen Weichenstellungen für die in der Route befindlichen Weichen. Diese werden mit POSITIVE (für Ausrichtung in gerader Richtung), sowie NEGATIVE (für Ausrichtung in abbiegender Richtung) gespeichert.

Da Routen sich teilweise überschneiden können, beinhaltet jede Route die Informationen über Routen, die mit dieser in Konflikt stehen. Das bedeutet, dass Konfliktrouten keinesfalls von anderen Zügen befahren werden dürfen, wenn die eigentliche Route gesperrt ist. Ist dies trotzdem der Fall, so kann es zu einer Kollision der Züge kommen. Damit das verhindert wird, speichert jede Route diese Konfliktrouten, die dann für die Strecke gesperrt werden. Ist dies jedoch doch der Fall, dass eine Konfliktroute befahren wird, so wird sofort ein Nothalt ausgelöst, woraufhin der Zug auf der legalen Route, wie auch alle Züge auf Konfliktrouten zum Stehen kommen.
Gerade diese Eigenschaft ist bedeutend für eine Kollisionsfreie Fahrt.

ID	SRC	DST	PATH	POINTS	LENGTH	MAX SPEED	CONFLICTS	DIRECTION
0	214	220	212,208,202,220	2:p,3:p,5:m,7:p,11:p,12:m	100	50	1,2,3,4,5,6,7,8,9,37,38,39,40,41,42,43,44,45,46,52,53,54,55,56,57,	0
1	214	203	212,208,203	2:p,3:p,5:m,7:p,11:p,12:p	100	50	0,2,3,4,5,6,7,8,9,37,38,39,40,41,42,43,44,45,46,52,53,54,55,56,57,	0
2	214	204	212,209,204	2:p,3:p,5:p,10:p	100	50	0,1,3,4,5,6,7,8,9,10,47,48,49,52,53,54,55,56,57,61,62,66,67,74,75,	0

Abbildung 4: Beispiel Interlocking-Table

Das Stellwerk erlaubt es, dass Züge in beiden Richtungen über das Gleisnetz fahren dürfen. Dazu ist es nötig, dass Routen die erlaubte Fahrrichtung speichern. Diese Eigenschaft ist zudem dafür nützlich, dass die Pfadberechnung vom Startpunkt eines zum Wunschziel eines Zuges vereinfacht wird.
Ein weiterer Sicherheitsaspekt ist der eigentliche Weg, den der Zug innerhalb der Route fährt. So wird gespeichert, über welche Track-Elemente die Route führt. Diese Informationen werden zur Verfolgung des Zuges benötigt. So wird der Zug streng überwacht, dass dieser die für ihn freigegebene Route vollständig befährt und keine Abweichungen zur Route nimmt. Dadurch können Weichenfehler schnell ausfindig gemacht werden, um mögliche Kollisionen mit anderen Zügen zu verhindern.
Für eine realistischere Darstellung wird die Routenlänge, sowie die
Höchstgeschwindigkeit für Züge gespeichert, die jedoch noch Standardwerte sind.

2.2.2 Stellwerk

Das Stellwerk stellt im Kontext das Herzstück des Systems dar und ist somit einer sehr hohen Sicherheitsbetrachtung unterlegen (SIL4). Daher wurde eine Architektur gewählt, die die wichtigen Steuerungsalgorithmen in einzelne Controller kapselt und Daten strikt trennt. Das hat den Vorteil, dass eine hohe Sicherheit der einzelnen Softwarekomponenten gewährleistet wird. Ebenso ist dadurch eine starke Parallelisierung der Controller möglich.
Zentral ist das Stellwerk in drei groben Komponenten unterteilt (Abbildung 5):
Die unterste Komponente ist der sogenannte "Hardware Abstraction Layer". Diese bildet sich bei dem Stellwerk aus einem Linux-Betriebssystem, sowie die Routing-Software, die die ankommenden Daten dekodiert und in den Datenpool einpflegt. Ebenso dient

sie als Schnittstelle zum Netzwerk, indem sie es ermöglicht, dass Stellwerksdaten, wie freigegebene Routen oder Weichensignale von den Controllern im Stellwerk zu den entsprechenden Komponenten (Märklin-Station) versendet werden können. Diese Schnittstelle wird ausschließlich vom Datenpool und vom Sub-Controller bedient, alle anderen Komponenten des Stellwerkes beziehen und versenden ihre Daten direkt über den Datenpool.

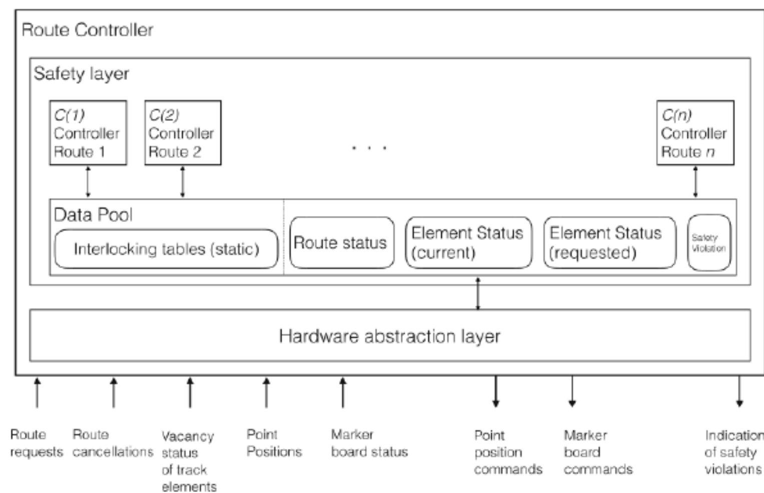


Abbildung 5: Architektur des Stellwerkes. [PHH16]

Eine weitere Schicht ist die Sicherheitsschicht (Safety-Layer). Diese beinhaltet zum einen den sensiblen Datenpool, mit Zustandsinformationen über Weichen, Züge und Track-Elementen. Ebenso beinhaltet diese weitere Controller, die zum Betrieb des Stellwerkes nötig sind. Diese Schicht wird besonders vom Netzwerk isoliert, da alle beinhalteten Komponenten den Zügen entweder Movement Authorities geben (Signal, ob ein Zug die Erlaubnis hat zu fahren) oder Weichen stellen.
In dieser Schicht gibt es eine Vielzahl an Controllern, die im Folgenden genauer beschrieben werden.

Datenpool Der Datenpool bildet im Stellwerk die Schicht zwischen der Hardware Abstraction Layer und der Schicht der Controller. Dieser dient als zentraler Datenspeicher für Routen, Weichen und Züge. Alle Daten, die entweder vom Positioning-System oder von einem TCC empfangen werden, werden in diesen gespeichert. Gleichzeitig werden die im Datenpool gespeicherten Daten von den Controllern verarbeitet und bearbeitet.

Main-Controller Der Main-Controller hat die Funktion, den Datenpool, sowie die Kommunikationsschnittstellen zu initialisieren und nach dem erfolgreichen Start des Stell-

werkes auf Zugregistrierungen zu hören. Ebenso liest der Main-Controller die vordefinierte Interlocking-Table, die alle Routendefinitionen beinhaltet, ein und speichert diese Informationen in dem Datenpool. Weiter initialisiert dieser die Router-Funktionen, die die Kommunikation mit anderen Modulen ermöglicht. Bei der Initialisierung wird jede aktuelle Zugposition im zuvor initialisierten Safety-Monitor eingetragen. Das verhindert das sofortige Auslösen des Safety-Monitors. Dieser würde auslösen, da sich Züge auf nicht-allokierte Routen befinden. Ebenfalls muss die Verbindung zu dem Ortungssystem gesichert werden, wodurch ein bestimmter Health-Controller initialisiert wird.
Nach erfolgreicher Initialisierung aller Komponenten, ist der Main-Controller für Zug-Registrierungen zuständig. Falls sich dann ein Zug registriert, startet dieser einen für den Zug zuständigen Train-Controller.

Train-Controller Jeder Zug wird separat vom Stellwerk verwaltet. Um Züge besser zu verwalten, wird jedem registrierten Zug einen Train-Controller zugeordnet, der diverse Aufgaben übernimmt. Sobald sich ein Zug beim Stellwerk registriert hat, erstellt der Main-Controller für diesen einen Train-Controller und übergibt diesen eine eindeutige Identifikationsnummer.
Dieser wartet nun, dass der Zug ein Ziel anfragt. Wurde ein Wunschziel angefragt, so wird im Streckennetz über Routen ein Pfad berechnet, der vom Zug angegebenen Startpunkt bis hin zum Wunschziel führt. Wurde kein Pfad gefunden, so wird dem Zug ein Signal geschickt, der sich dann ein neues Ziel suchen muss.

Die folgende Abbildung 6 zeigt dabei schematisch den Zustandsautomaten des Train-Controllers:

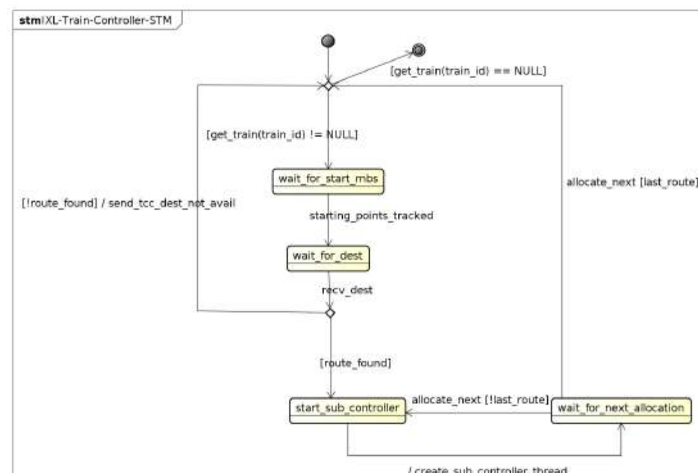


Abbildung 6: Train-Controller Zustandsautomat

Nach erfolgreicher Berechnung der Route, vom Start zum Ziel, wird der erste Sub-Controller der ersten Route gestartet. Ab dann übernimmt dieser die Verantwortung

für den Zug und führt diesen sicher durch die erste Teilroute.

Routenfindung Zur Routenfindung wird die Implementierung eines Algorithmus von Edsger W. Dijkstra, den Dijkstra-Algorithmus zum Finden des kürzesten Pfades in einem Graphen genutzt. Dieser sucht per Brute-Force von einem gegebenen Startknoten aus den kürzesten Weg über gewichtete Kanten zum Zielknoten. Dabei fährt dieser schrittweise jede ausgehende Kante von jeden Knoten ab und berechnet, anhand der Gewichtung (entspricht der Länge der Kante), den kürzesten Pfad [Dij20].

Von jeden Punkt aus existieren mehrere Teilrouten (siehe Interlocking-Table in Abbildung 4), die zu einem weiteren Punkt (Track-Element) führen. Man kann sich alle Routen zusammen als einen Graphen vorstellen. Jede Route hat einen Start, sowie Endpunkt (siehe Kapitel 2.2.1). Der Startpunkt jeder Route ist der Endpunkt einer anderen Route. Durch diese Eigenschaft lässt sich aus allen Routen einen Graphen erstellen. Dabei hat jede Route ebenso eine bestimmte Länge (Gewichtung der Kante). Der Dijkstra-Algorithmus kann somit in diesem Fall perfekt angewandt werden.

Deadlock - Erkennung Prinzipiell ist es möglich, dass in dem Gleisnetz ein Deadlock zwischen drei oder mehreren Zügen auftreten kann. Ein Deadlock tritt genau dann auf, wenn jeder der drei Züge den Zielpunkt als Startpunkt des vorausfahrenden Zuges gewählt hat oder das Ziel auf dem Pfad eines anderen Zuges liegt. Aus Sicherheitsgründen gibt das Stellwerk aber keinen Zug eine Fahrtfreigabe, da es möglich ist, dass ein Zug mit dem anderen kollidiert.
Als Lösung dafür, wird jede angefragte Route in eine Liste gespeichert. Ein Algorithmus berechnet dann, ob diese Routen zusammen im Gleisnetz einen Kreis ergeben. Wenn das der Fall ist, wird ein Deadlock erkannt. Zum Lösen des Konfliktes wird dem letzten Zug, der eine Route angefragt hat, ein Signal gesendet, dass dieser sich ein neues Ziel auswählen soll. Dieser Zug fragt dann eine neue Route an, die, wenn diese mit den Routen der anderen Züge keinen Deadlock bildet, für den Zug allokiert und zum Fahren freigegeben wird. Ebenfalls erhalten daraufhin die anderen wartenden Züge, wenn es keinen Konflikt gibt, eine Movement Authority, die es denen erlaubt, schließlich weiterzufahren.
Wird jedoch mit der Wahl der neuen Route erneut eine Deadlock-Situation ausgelöst, so wiederholt sich dieser Vorgang, bis der Deadlock aufgehoben werden kann.

Sub-Controller Der Sub-Controller hat als Stellwerkskomponente die zentrale Aufgabe eine Teilroute der Gesamtroute eines Zuges zu verwalten und zu überwachen. Nachdem eine passende Route für einen Zug gefunden wurde, wird der Sub-Controller für die erste Teilroute gestartet, der dafür die Konfliktrouten sperrt, sowie die Weichen stellt.

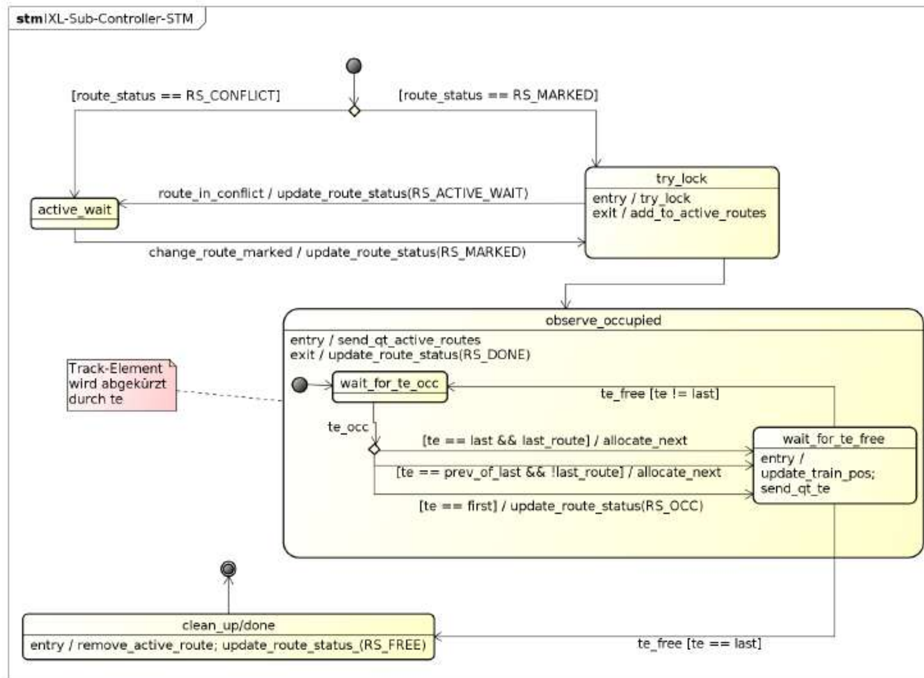


Abbildung 7: Zustandsautomaten des Sub-Controllers

Der Sub-Controller ist als Zustandsautomat konzipiert, wobei sich die Zustände an den Zuständen der Route im Datenpool orientieren. Der Zustand jeder Route wird im Datenpool zentral gespeichert. Zu Beginn ist kein Sub-Controller gestartet. Wenn nun ein Sub-Controller für eine Route gestartet wird, wird zunächst der Zustand aus dem Datenpool gelesen und danach gestartet.
Dabei kann sich dieser im Zustand RS_CONFLICT befinden. Das bedeutet, dass diese Route gerade im Konfliktzustand gesperrt ist. Dabei wartet der Sub-Controller (active_wait), bis dieser wieder daraus gelöst wird. Ein Sub-Controller ist genau dann in diesem Zustand, wenn eine Route gerade befahren wird, zu der diese Route in Konflikt steht.
Ebenfalls kann die Route im Zustand RS_MARKED sein, wenn der Sub-Controller erfolgreich gestartet werden konnte und vorher im Zustand RS_FREE war. Falls eine Route zuerst von einem Sub-Controller markiert wird (in den Zustand RS_MARKED versetzt) und dann ein Sub-Controller gestartet wird, mit der diese in Konflikt steht, wird diese durch RS_ACTIVE_WAIT in den Wartemodus versetzt. Diese wird erst dann daraus entfernt, wenn die zu der in Konflikt stehende Route beendet wird.
Falls noch kein Sub-Controller für diese Route gestartet wurde, wechselt diese Route zu RS_FREE, sonst zum Zustand RS_MARKED.
Im Zustand RS_MARKED führt der Sub-Controller das Sperren der Route durch. Dabei werden zunächst alle auf der Teilroute befindlichen Weichen in die richtige Position

versetzt, wie diese in der Interlocking-Table deklariert wurden. Ebenfalls werden hier alle im Konflikt stehenden Routen in den Zustand RS_CONFLICT versetzt, damit diese nicht von anderen Zügen gestartet werden können. Danach wird dem Safety-Monitor mitgeteilt, dass diese Route nun aktiv befahren wird, damit dieser diese überwacht. Schließlich wechselt diese in den Zustand RS_LOCKED, womit feststeht, dass die Route vollständig für das Befahren gesperrt wurde und gesichert ist. Nun wird dem Zug eine Moving Authority mitgeteilt, die dem Zug somit die Fahrterlaubnis erteilt.
Der Sub-Controller wartet, bis der Zug das erste Track-Element befährt und wechselt dann in den Zustand RS_OCCUPIED (RS_OCC). In diesem Zustand prüft der Sub-Controller, dass der Zug jedes Track-Element sequentiell befährt, indem dieser die befahrenden Track-Elemente nacheinander zählt. Dabei wird kontinuierlich die Position des Zuges im Datenpool aktualisiert. Wenn das zweitletzte Track-Element der Teilroute vom Zug befahren wird, wird dem Train-Controller übermittelt, dass dieser den nächsten Sub-Controller startet, der dann mit dem Sperrprozess der nächsten Teilroute beginnt. Wenn der Zug alle, wie in der Interlocking-Table spezifizierten Track-Elemente befahren hat, beendet dieser sich.
Hat der Zug das letzte Track-Element der Teilroute verlassen, beginnt der Sub-Controller mit dem Clean-Up-Verfahren, bei dem die Teilroute wieder freigegeben wird und aus den aktiven Routen im Safety-Monitor entfernt wird. Nach wechseln in den Zustand RS_DONE beendet dieser sich dann selbst und ist für andere Züge verfügbar.

Safety-Monitor Der Safety-Monitor ist ein Teil der Sicherheitsschicht und überwach permanent das gesamte Streckennetz. Dieser speichert alle aktiv befahrenen Routen und überprüft, ob jeder Zug nur die für ihn reserviert Strecke befährt. Wenn aus Gründen von Weichendefekten es zu einer Abweichung eines Zuges von seiner freigegebenen Route kommt, wird direkt ein Nothalt ausgelöst. Ebenso überwacht dieser die nicht-gesperrten Routen, ob sich auf ihnen kein Zug befindet oder sich möglicherweise verlorene Waggons befinden.

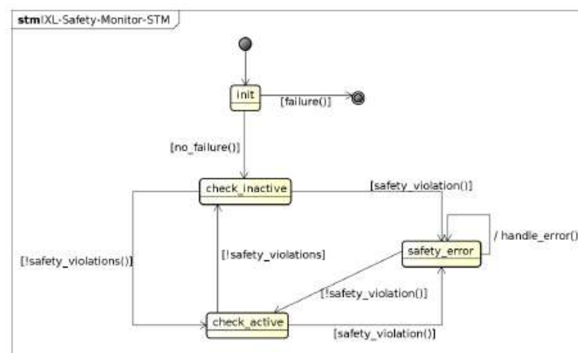


Abbildung 8: Zustandsautomat des Safety-Monitors

Auch dieser ist als Zustandsautomat implementiert, indem dieser im Zustand CHECK_ACTIV alle momentan aktiv befahrenden Zügen überprüft. Im Zustand CHECK_INACTIVE werden alle nicht aktiv befahrenden Routen kontrolliert, dass diese nicht befahren werden, also kein Track-Element belegt ist, was sonst einen Nothalt der Züge auslöst.

Health-Monitor Es muss zu jeden Zeitpunkt eine intakte Verbindung zu allen Geräten vorhanden sein. Daher wurde ein Health-Monitor implementiert, der in einem bestimmten Zeitzyklus von dem Ortungssystem und von den Zügen eine Health-Nachricht erhält, die die korrekte Funktionsweise des Systems bestätigt. Wenn von einem der Geräte die Health-Nachricht für einige Sekunden ausfällt, registriert das Stellwerk dies direkt als Systemabsturz und stoppt die Züge. Es werden nacheinander immer wieder Health-Nachrichten vom Zugsteuerungscomputer(TCC) und vom Ortungssystem (POS) geprüft, nachdem die jeweiligen Timer zum Abfrageintervall initialisiert wurden.

Kapitel 3

Model Checking mit FDR4

Beim Entwickeln von großen Hard- und Softwaresystemen wird heutzutage mehr Zeit für das Testen und Verifizieren dieser Systeme in Anspruch genommen, als in der eigentlichen Implementierung des Systems. [Gre19] Daher braucht es effiziente automatisierte Lösungen, Fehler so früh wie möglich zu entdecken und diese somit schnell beseitigen zu können. Andernfalls steigen durch spätere Änderungen die Kosten ungemein. Formale Methoden bieten da den richtigen Ansatz, effektive Verifikationstechniken bereitzustellen. Ein Ansatz hierbei bietet das Model Checking.

Model Checking ist eine Technik, das Modell einer sicherheitskritischen Software schon vor Beginn der Implementierung auf Einhaltung von Spezifikationen zu prüfen. Dabei wird das Modell in eine spezielle Semantik umgewandelt, die ein Model-Checker versteht. Eine Form dieser Semantik ist CSP, Communicating Sequential Processes, welche 1985 von C.A.R. Hoare spezifiziert und veröffentlicht wurde. [Hoa85] Thomas Gibson-Robinson et. al. entwarfen zu dieser Prozessalgebra einen sehr schnellen und effizienten Model-Checker, FDR (Failures-Divergences Refinement). [TGR13]
FDR in der neusten Version (FDR4), der in dieser Arbeit zum Model Checking verwendet wird, ist dabei ein sogenannter Refinement Checker, der das Modell auf Verfeinerung bezüglich einer Sicherheitsspezifikation prüft. Genaueres wird im Kapitel 3.2 beschreiben. FDR4 nutzt, sowie alle Vorgängerversionen, CSP_M, welches Erweiterungen im Vergleich zu CSP enthält. CSP_M ist dabei die Kombination aus einer funktionalen Programmiersprache und der von Hoare entworfenen Prozessalgebra CSP. Diese Prozessalgebra wird im Kapitel 3.1 weiter erläutern.

3.1 CSP - Syntax und Semantik

C.A.R. Hoare spezifizierte 1985 an der Oxford Universität erstmals die Prozessalgebra Communicating Sequential Processes [Hoa85] zum Modellieren von parallelen, untereinander kommunizierenden Prozessen. Eine Weiterentwicklung davon ist CSP_M, die

maschinenlesbare Version von CSP. CSP_M ist dabei eine Mischung aus einer funktionalen Programmiersprache im klassischen Sinne und der von C.A.R. Hoare entwickelten Prozessalgebra CSP. [TGR19a]
Herkömmliche Programmiersprachen wie C oder Java beschreiben das Verhalten und die Art und Weise, was und wie ein Computer eine Rechenoperation ausführen muss. Dabei gibt es häufig ein einziges Programm, welches prozedural Befehle ausführt oder Werte durch Funktionen berechnet. CSP hingegen beschreibt Programme auf die Art und Weise, wie diese mit anderen Prozessen kommunizieren. Dabei kommt es weniger auf die Berechnung an, als auf das Verhalten der Prozesse untereinander. So erzeugen Prozesse über sogenannte Channels (Kanäle) Events, welche von anderen Prozessen, die darauf hören, abgefangen werden. Dabei können pro Kanal mehrere Events gleichzeitig auftreten, die dann jedoch zeit-separiert auf einem Kanal abgebildet werden.[TGR19a]

Konventionen Allgemein verfolgt CSP einige Konventionen, die im Laufe dieser Arbeit auch eingehalten werden. Dabei unterscheidet man deutlich bei Groß- und Kleinschreibung, um beispielsweise Channel und Prozesse deutlich zu unterscheiden. Bezeichner von Prozessen werden hierbei üblich in Großbuchstaben geschrieben, wobei Bezeichner von Kanälen kleingeschrieben werden. Alle weiteren Bezeichner für datatypes sind ebenfalls in Kleinbuchstaben, Konstanten jedoch wieder in Großbuchstaben.

3.1.1 Channels

In CSP, sowie CSP_M nutzen Prozesse Kanäle (Channel), um miteinander zu kommunizieren und um Daten auszutauschen. Die Deklaration dieser ist ähnlich, wie bei der Deklaration von Datentypen.

$$\textbf{channel}\ c_1, ..., c_n : t_1.t_n$$

Sind Kanäle, wobei $c_1, ..., c_n$ die Bezeichner der Kanäle sind und $t_1.t_n$ sind die Typen der Kanäle.

Der Kanal

$$\textbf{channel}\ bool_test : \{0..2\}.Bool$$

kann dabei beispielsweise folgende Events erzeugen:

$$\alpha bool_test = \{0.true, 0.false, 1, true, 1.false, 2.true, 2.false\}$$

Um nun auf Kanälen Events zu erzeugen, gibt es verschiedene Techniken. Es ist möglich Events auf Kanälen zu erzeugen, zu empfangen und auf verschiedene Events

zu synchronisieren.

- $c?n$: Es wird auf jedes Event des Kanals c gehört und in die Variable n gespeichert.
- $c!1$: Über den Kanal c wird "1" als Event erzeugt.
- $c.1$: Es wird nur auf eine "1" auf dem Kanal c gehört. Dabei kann es ebenso auch eine "1" erzeugen. Der Prozess synchronisiert sich also auf das Event $c.1$ mit anderen Prozessen. Dieses wird noch bei der Synchronisation von Prozessen im Kapitel 3.1.2 deutlicher.

3.1.2 Prozesse

CSP-Prozesse definieren Prozeduren, die Events auf Kanäle erzeugen. Der einfachste aller Prozesse ist der Prozess STOP, der eine bestimmte Prozedur stoppt.
Prozesse reagieren auf die Umwelt. Das bedeutet, dass auf jedes Event, welches an Kanälen anliegt, auf das ein Prozess hört, von diesem verarbeitet wird. Ein Beispiel ist das Registrieren eines Zuges bei dem Stellwerk. Dabei führt das Stellwerk weiter andere Funktionen aus, reagiert jedoch, sobald ein Registrierungs-Event von einem Zug anliegt.

Beispiel $Q = a \rightarrow P$: Erst gilt a und dann verhält sich Q wie P.
Der Operator external Choice [ ] sagt aus, dass ein von zwei Events auftreten können, wie Pattern-Matching in funktionalen Sprachen:

```
channel a, b

P = a -> P
    []
    b -> STOP
```

Listing 1: Beispiel Pattern-Matching

Tritt somit entweder, wie in Listing 1, ein Event auf dem Kanal a oder b auf, reagiert der Prozess P dementsprechend. Es ist ebenso möglich Prozesse aufeinander zu synchronisieren.

```
channel a, b, c

Q = a -> Q
    []
    c -> STOP
    []
    b -> Q

S = Q [| {| a |} |] P
S' = Q [| { a.1 } |] P
```

Listing 2: Synchronisationsbeispiel

Sollen beispielsweise zwei Prozesse P (siehe Listing 1) und Q (siehe Listing 2) sich auf alle Events vom Kanal a synchronisieren, so beschreibt dies den Prozess S. Sobald dann ein Event auf dem Kanal a anliegt, reagieren sowohl Q, als auch P auf das Event und nehmen die Transition zum Folgezustand. S' hingegen synchronisiert die beiden Prozesse Q und P nur auf das Event 1 auf dem Kanal a. Auch dies ist möglich, da die Synchronisierung auf dem gesamten Alphabet eines Kanals in manchen Fällen falsch ist.
Die folgenden Prozesse sind Beispiele für CSP-Prozesse, die über ein Kanal kommunizieren.

```
channel a, b : {0..3}

P = a.0 -> STOP
    []
    b?x -> Q

Q = a!0 -> STOPs
```

Listing 3: Kommunizierende Prozesse

Sobald auf dem Kanal a eine "0" anliegt, terminiert Prozess P. Falls auf dem Kanal b ein Signal anliegt, wird der Prozess Q aufgerufen, der dann auf Kanal a eine 0 sendet und terminiert.

3.1.3 Assertions

Für den Refinement Check gibt es in CSP_M einige Typen von Assertions, die prüfen, ob eine bestimmte Spezifikation in einem bestimmten Verhalten erfüllt ist. Dazu gibt es drei grundsätzliche Typen:

```
1. assert Spec [T= Impl

2. assert Spec [F= Impl

3. assert Spec [FD= Impl
```

Dies bedeutet im Beispiel (1), dass der Trace, der von Impl erzeugt wird eine Teilmenge vom Trace von Spec sein muss. Detaillierter für alle Arten heißt das:

(1) Beschreibt, dass Spec Trace-Verfeinert wird durch Impl:
$Spec \sqsubseteq_{\mathrm{T}} Impl \Rightarrow trace(Impl) \subseteq trace(Spec)$

(2) Beschreibt, dass Spec Failure-Verfeinert wird durch Impl:
$Spec \sqsubseteq_{\mathrm{F}} Impl \Rightarrow failure(Impl) \subseteq failure(Spec)$

(2) Beschreibt, dass Spec Divergence-Verfeinert wird durch Impl:
$Spec \sqsubseteq_{\mathrm{FD}} Impl \Rightarrow divergence(Impl) \subseteq divergence(Spec)$

3.1.4 Hiding

Innerhalb eines Prozesses oder einer Assertion ist es möglich auftretende Events zu verbergen. Das hat den Grund, dass diese normal ausführen sollen, aber für den Refinement Check nicht beachtet werden sollen. Soll beispielsweise geprüft werden, ob ein Prozess jemals ein STOP-Event erzeugt und niemals ein Error-Event, so werden alle Events, die ein Prozess erzeugt verborgen, außer das Error-Event. Werden trotzdem alle erzeugt, so ist die Refinement-Bedingung verletzt. Intern werden dann die verborgenen Events in ein τ umgewandelt.

3.2 FDR4

FDR4 ist ein Refinement Checker. Es kann ein gegebenes Modell, modelliert in CSP_M, auf eine Verfeinerung durch eine gegebene Spezifikation prüfen. Dabei unterstützt das Tool nicht nur das Prüfen auf Trace-Refinement, sondern kann ebenfalls auf Fehler- und Fehlerdivergenz-Refinement, sowie auf Deadlock- oder Lifelock-Freiheit prüfen. FDR wurde erstmals 1991 von der Firma Formal System Ltd. und der Universität Oxford veröffentlicht. FDR steht hierbei für Failure Divergence Refinement.

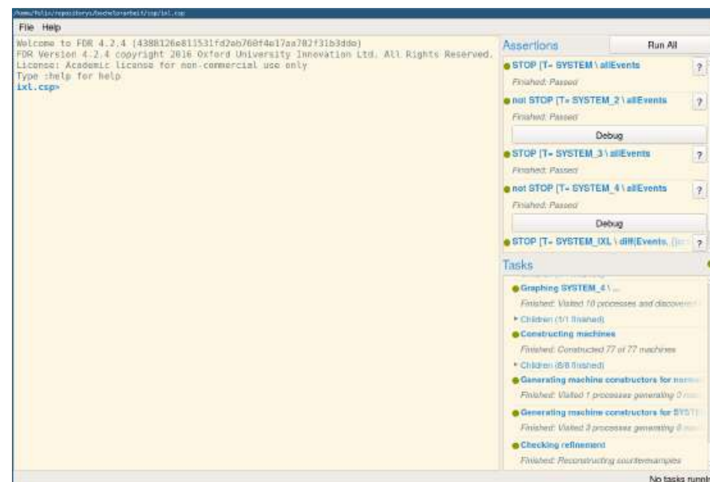


Abbildung 9: Benutzeroberfläche von FDR4

Es wandelt die CSP_M Spezifikation in eine interne Form um, wodurch, durch zentrale Algorithmen, auf eine Verfeinerung überprüft wird. Dabei ist FDR4 kein normaler Model-Checker, sondern nutzt im Vergleich zu anderen Model-Checkern, wie nuXmv keinen SAT-Solver zum Modell-Prüfen, sondern wendet eine andere Art und Weise an. [Kes14] [OU19]

3.2.1 Übersetzung von CSP

FDR4 hat eine besondere Art, zu prüfen, ob eine Spezifikation ein Modell verfeinert, also unter bestimmten Bedingungen im Modell gilt. Dazu muss der zuvor implementierte CSP_M-Code in eine andere Darstellung übersetzt werden. FDR4 verfolgt dabei verschiedene Schritte des Übersetzens.
Zunächst wird der CSP_M Code auf syntaktische Fehler überprüft. Wird dieser Vorgang erfolgreich abgeschlossen, so beginnt das Tool damit, den CSP_M-Code in reines CSP zu übersetzen. Dabei werden die funktionalen Eigenschaften von CSP_M eliminiert, was als Ergebnis die reine Prozessalgebra CSP entspricht. Da CSP Zustandsautomaten modelliert, wird der CSP-Code nun von FDR4 in ein Labelled-Transition-System, übersetzt. Dieser ist dann exakt äquivalent zur CSP-Darstellung, beschleunigt jedoch den Refinement Check erheblich.
Ebenso wie das Modell wird auch die zu überprüfende Spezifikation in einem Labelled-Transition-System umgewandelt, nach demselben Prinzip, wie das Modell. Diese Darstellung wird nun weiter genutzt, um den eigentlichen Refinement-Check auszuführen. [TGR13]

3.2.2 Refinement Checking

Um Model Checking auf das gegebene Modell anzuwenden, wird ein sogenanntes Refinement Checking angewendet. Dieses unterscheidet sich dahingegen vom konventionellen Model Checking, dass dieses Verfahren ein Modell wortwörtlich auf eine Verfeinerung untersucht.
A Trace-verfeinert B, genau dann, wenn die Folge von Events, die von A erzeugt werden, eine Teilmenge der von B erzeugten Events sind. Genauer heißt dies:

$$trace(A) \subseteq trace(B)$$

Beispiel Ein einfaches Beispiel zeigt Listing 4:

```
1   channel a, b, c
2
3   M0 = a -> M1
4   M1 = b -> M0
5        []
6        a -> M2
7   M2 = b -> M0
8
9   S0 = a -> S1
10  S1 = a -> S2
11  S2 = b -> S0
12
13  assert S0 [T= M0
```

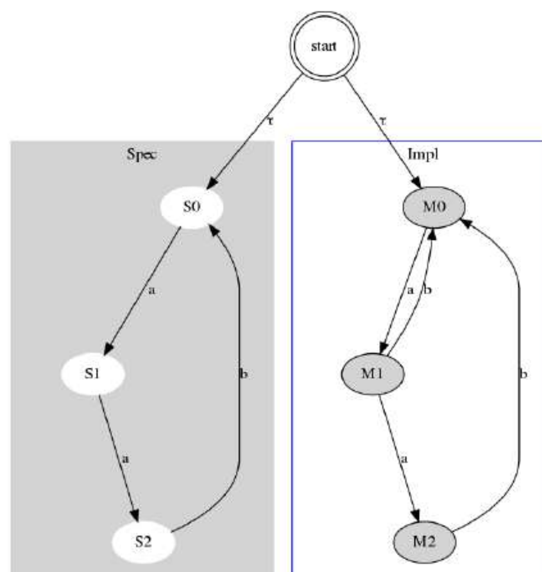
Listing 4: Beispiel Code Refinement-Checking

Durch die assert-Anweisung soll geprüft werden, ob die Spezifikation S0 vom Modell M0 Trace-verfeinert wird. Falls dieser Test mit PASS abschließen soll, so darf M0 nur den Trace, den S0 spezifiziert (oder eine Teilmenge), erzeugen.
S0 erzeugt dabei den Trace $T = \{aab\}$. Wie zu erkennen, erzeugt M0 zunächst ein a und wechselt dann zu M1. M1 erzeugt dann a, wechselt zu M2, welches ein b erzeugt und kehrt dann zum Startzustand M0 zurück. Dadurch erzeugt M0 bis jetzt den zu erzeugenden Trace. Doch, nach der Bedingung darf M0 nur den Trace von S0 erzeugen (Zeile 4-6 in Listing 4). Falls M0 noch einen weiteren Trace erzeugt, schlägt die Bedingung fehl und somit der gesamte Test.
Wie in Zeile 4 von Listing 4 zu erkennen, erzeugt M0 jedoch auch noch nach einem a ein b. Somit ergibt sich für M0 eine Menge der erzeugbaren Traces von $R = \{aab, ab\}$. Daraus ergibt sich:

$$\{aab, ab\} \nsubseteq \{aab\} \Rightarrow trace(M0) \nsubseteq trace(S0) \Rightarrow Spec \not\sqsubseteq_{\mathrm{T}} Impl \Rightarrow \text{Failed}$$

Die Bedingung ist somit falsch. Andersherum könnte man sagen, dass das Modell von der Spezifikation Trace-Verfeinert wird. Um den Fehler nun zu beheben, muss dieser Extra-Transition-Fault behoben werden, indem die Fehlerhafte Transition entfernt wird.

Refinement-Checking in FDR4 FDR4 hingegen implementiert dabei eine ähnliche Vorgehensweise, zu beweisen, dass eine Trace-Verfeinerung gilt. Nachdem das zu prüfende Modell in ein Labelled Transition System umgewandelt und die Spezifikation normalisiert wurde (jeder Prozess erzeugt ein Event und wechselt dann zu einem neuen Zustand [TGR13]), werden beide, als CSP-Prozess modellierten Modelle wie folgt durch Breitensuche untersucht:



Zunächst befindet sich der FDR4 Refinement-Checker im Zustand *start* und betrachtet die Zustandspaare $(S0, M0)$. Dabei untersucht FDR4 die erzeugbaren Events und den jeweiligen Folgezustand. S0 erzeugt dabei ein a und wechselt zu S1, M0 erzeugt ebenso ein a und wechselt zu M1. In diesem Falle sind diese äquivalent, es wurde keine Transition in M0 gefunden, die einen ungültigen Trace erzeugt. Im Folgeschritt betrachtet FDR4 nun das Zustandspaar $(S1, M1)$. S1 erzeugt ein weiteres a, wohingegen M1 dies auch tut. FDR4 erkennt eine weitere Transition, die von M1 ausgeht. Dadurch erzeugt M1 zusätzlich ein b, die es so nicht in $S1$ gibt. Diese Transition erkennt FDR4 als Gegenbeispiel zu der in Listing 4 genannten Spezifikation und erkennt einen Fehler. Anstatt ein b sollte M1 hier nur ein a erzeugen. [TGR13]

```
S0 [T= M0:
    Log:
        Result: Failed
        Visited States: 2
        Visited Transitions: 3
        Visited Plys: 1
        Estimated Total Storage: 268MB
        Counterexample (Trace Counterexample)
        Specification Debug:
        Trace: <a>
        Available Events: {a}
        Implementation Debug:
        M1 (Trace Behaviour):
        Trace: <a>
        Error Event: b
```

Listing 5: Ausgabe von FDR4 bezüglich des Bespiels

Dadurch, dass FDR4 wie hier mithilfe von Breitensuche nach Gegenbeispielen zu der Behauptung sucht, ergibt sich ein schnelles Finden von Fehlern, sowie einen minimalen Fehler-Trace.
Um den Vorgang des Refinement Checking zu beschleunigen, beherrscht FDR4 das parallele Refinement Checking auf mehreren CPU-Kernen. In jeder Schicht im Modell wird geprüft, ob ein bestimmter Trace von einem Zustand aus erzeugt werden kann. Eine Schicht ist dabei beispielsweise ein Zustandspaar im Modell und in der Spezifikation, wie zum Beispiel (M0,S0). Das Suchen nach einem Trace, ausgehend von einem Zustand, wird in FDR4 als quasi Wettlauf zwischen den CPU-Kernen implementiert. Der Kern, beziehungsweise der FDR-Thread, der zuerst einen möglichen Trace entdeckt, liefert eine Lösung für diese Schicht im Modell. Bei kleineren Modellen mag dieser Geschwindigkeitsvorteil zwar nicht direkt bemerkbar sein, bei sehr großen Modellen, wie beim gesamten Stellwerk ist dieser Vorteil durchaus bemerkbar. [TGR19b]

Kapitel 4

Test und Verifikation

Test und Verifikation nimmt bei hochkomplexen, sicherheitskritischen Systemen einen sehr hohen Stellenwert ein. Gerade bei einem Stellwerk muss der Fokus klar auf Sicherheit und Robustheit liegen. Da bei dem entwickelten autonomen Stellwerk kein Eingreifen von Menschenhand nötig ist, ist das Testen um so aufwendiger und notwendiger. Es muss zu jeder Zeit garantiert werden, dass das System immer richtige Entscheidungen trifft, um eine Gefährdung für Mensch und Maschine zu verhindern. Riskant wird es genau dann, wenn das Stellwerk Weichen für Züge falsch stellt, die dann mit anderen Zügen kollidieren. Gerade diese große Gefahr soll auf jeden Fall verhindert werden. Ein weiterer Punkt ist das Entgleisen eines Zuges. Auch das stellt eine vergleichbar große Gefahr dar, was lebensgefährliche Folgen hätte.
Um im voraus genau sagen zu können, dass auf jeden Fall das Modell des Systems diese Gefahren behandelt, soll dies durch Modellprüfung getestet werden. Durch geeignetes Model Checking der sicherheitskritischen Komponenten des Stellwerkes, soll bewiesen werden, dass das Stellwerk nach dem Modell Checking als durchaus sicher eingestuft werden kann. Dabei liegt der Fokus klar auf das Testen von Kollisions- und Entgleisungsvermeidung.
Zunächst wurde das Schienennetz, sowie das Modell des Sub-Controllers, des Train-Controllers und des Main-Controllers (siehe Abbildung 5) in CSP_M modelliert. Um zunächst sicherzustellen, dass das in CSP_M modellierte Schienennetz dem originalen entspricht, befahren Probeweise Züge dieses ohne existierendes Stellwerk. Das zeigt, dass Züge generell das Gleisnetz befahren können und, dass dieses für weitere Tests einsetzbar ist.
Anschließend wird eine Teststrategie vorgestellt, wie die Sicherheit des Stellwerkes gezeigt werden kann. Dabei liegt die Interlocking-Table, sowie Routen, Sub- und Train-Controller im Fokus der Modellprüfung.
So soll das abschließende Resümee die Sicherheit des Stellwerkes darlegen oder gar existierende Fehler in der Modellierung aufdecken.

4.1 Modellierung der Streckenabschnitte

Ein Gleisnetz, besteht aus einzelnen Teilkomponenten, die jeweils getrennt modelliert wurden. Diese bilden zusammen, wenn diese richtig miteinander verbunden sind, ein Gleisnetz, welches von mehreren Zügen befahren werden kann. Dabei gibt es

- Kreuzweichen,
- Weichen (links- und rechtsgerichtet),
- gerade Track-Elemente, die eine Zugposition detektieren.

Jede Weiche, sowie jedes Track-Element wurde als Zustandsautomat modelliert. Um eine Weiche oder ein Track-Element zu befahren, ist es nötig, diese zunächst anzufragen und dann in die richtige Position zu versetzen (Weichen). Das hat den Vorteil, dass es nur einem Zug gestattet ist diesen Streckenabschnitt zu befahren, um Kollisionen zu vermeiden. Befährt nach erfolgreichen Sperren eines Streckenabschnittes ein zweiter Zug diesen, so wird ein CRASH-Event ausgelöst. Im Folgenden werden die einzelnen Gleise beschrieben. Dazu wird das Verhalten jeder Gleisart durch ein SysML-Zustandsautomat dargestellt und erklärt. Das implementierte Gleisnetz besteht aus folgende Gleise:

Kreuzweiche Eine Kreuzweiche besteht aus vier Ein-, sowie Ausgängen, die ein Zug in unterschiedlichen Richtungen befahren kann. Es gibt einmal die Straight-Richtung, die von S_0 nach S_2, oder von S_1 nach S_3, sowie andersherum führt. Die Cross-Richtung hingegen führt von S_1 nach S_3, von S_1 nach S_2 oder andersherum. Das ist in Abbildung 10 zu sehen.

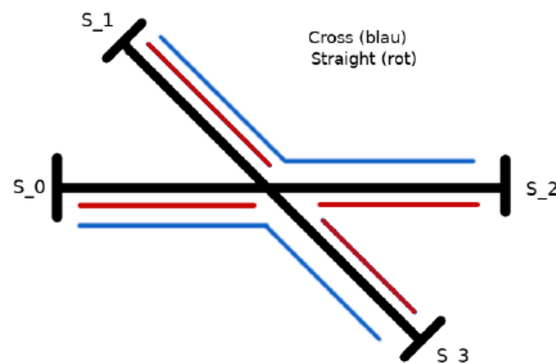


Abbildung 10: Schametische Darstellung einer Kreuzweiche

Wie im Zustandsautomaten in der Abbildung 11 zu sehen, verfolgt das Allokieren einer Kreuzweiche eine gewisse Strategie. So muss diese zunächst durch ein request angefragt werden. Anschließend muss diese über set in die richtige Position gesetzt werden, damit der Zug in die richtige Richtung fährt. Anschließend wird streng überwacht, wie der Zug fährt und nicht falsch abbiegt. Dazu werden verschiedene occupied-Zustände genommen, die den richtigen Fahrtverlauf vom falschen unterscheiden.
Wenn die Weiche in gerader Richtung (Straight) gesperrt ist kann dieser nur von s_0 nach s_2 oder von s_1 nach s_3 oder andersherum fahren. Kommt der Zug von s_0, wechselt der Zustandsautomat in den Zustand LOCKED_STR_OCC_L2R_UP, von s_2 kommend in den Zustand LOCKED_STR_OCC_R2L_DWN. Von s_1 kommend wechselt der Zustandsautomat in den Zustand LOCKED_STR_OCC_L2R_UP und von anderer Seite, aus Richtung s_3 in den Zustand LOCKED_STR_OCC_R2L_UP.

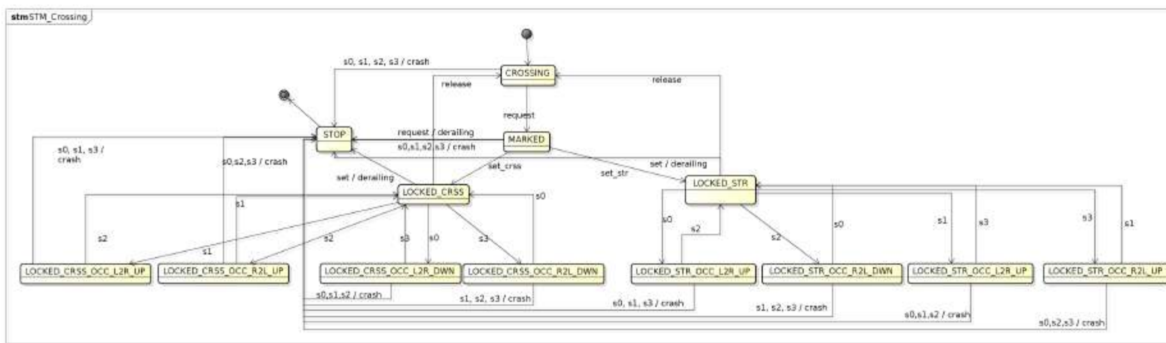
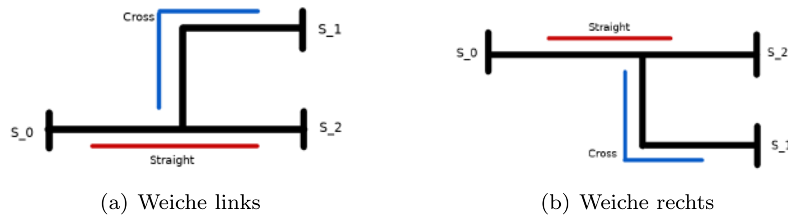


Abbildung 11: Zustandsautomat einer Kreuzweiche

Anders ist es nun, wenn die Weiche in Cross-Richtung steht. In dieser Position biegen Züge quasi seitlich ab, wie in Abbildung 10 zu sehen. Fährt ein Zug also von s_0 hinein, so wechselt der Zustandsautomat in den Zustand LOCKED_CRSS_OCC_L2R_DWN, woraufhin dieser dann bei s_3 hinausfährt. Wenn der Zug jedoch von s_3 in die Kreuzweiche hineinfährt, so wird in dem Zustand LOCKED_CRSS_OCC_R2L_DWN gewechselt.
Anders ist es, wenn der Zug von s_1 in die Kreuzweiche hineinfährt. Dabei wechselt der Zustandsautomat in den Zustand LOCKED_CRSS_OCC_L2R_UP und dann bei s_2 wieder hinaus. Kommt dieser von s_2 wird in dem Zustand LOCKED_CRSS_OCC_R2L_UP gewechselt, woraufhin der Zug bei s_1 wieder hinausfährt.
Die spezielle Kreuzweiche lässt den Zug in Straight-Richtung geradeaus durchfahren, wobei der Zug bei Weichenposition Cross seitlich abfährt, wie in Abbildung 10 zu sehen.

Weiche (links- und rechtsgerichtet) Es gibt zwei Arten von Weichen. Bei einer kann der Zug nach rechts abbiegen, bei der anderen nach links. Die beiden Weichen Arten sind im Folgenden zu sehen:



(a) Weiche links (b) Weiche rechts

Abbildung 12: Weichen

In dieser Arbeit wird jedoch nur eine Modellierung der Weiche verwendet, da die Implementierungen der beiden Weichenarten sich nicht im Modell unterscheidet. Der Zustandsautomat der einfachen Weiche ähnelt dabei dem der Kreuzweiche (siehe: Abbildung 11). Nach Anfragen der Route (request) und durch Sperren in, entweder Straight-Richtung oder in Cross-Richtung, werden die Ausgangsrichtungen durch verschiedene Occupied-Zustände überwacht.
Wurde nun in Straight-Richtung gesperrt, kann der Zug nur von S_0 oder s_2 hineinfahren (Abbildung 12). Wenn der Zug von S_0 hineinfährt wechselt der Zustandsautomat in den Zustand P_LOCKED_STR_OCC_FROM_LEFT, fährt der Zug von S_2 hinein- wechselt der Zustandsautomat in den Zustand
P_LOCKED_STR_OCC_FROM_RIGHT.

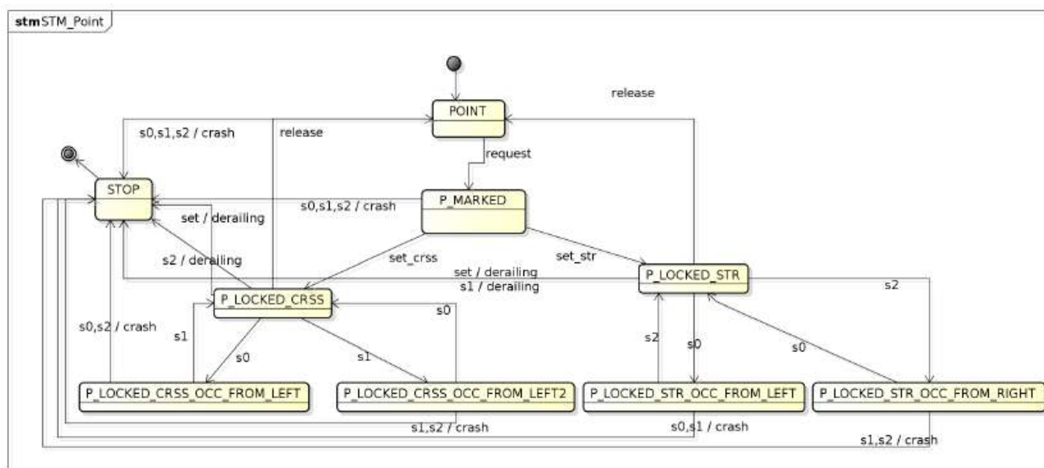


Abbildung 13: Zustandsautomaten einer Weiche

Ist die Weiche in Cross-Richtung gesperrt, so kann der Zug nur von S_0 oder S_1

hineinfahren. Beim Hineinfahren von S_0 kommend wechselt der Zustandsautomat in den Zustand P_LOCKED_CRSS_OCC_FROM_LEFT, von S_1 kommend in den Zustand P_LOCKED_CRSS_OCC_FROM_LEFT2. Fährt der Zug nun von S_0 kommend in Richtung S_2, so tritt eine Sicherheitsverletzung auf und es wird ein CRASH-Signal ausgelöst.

Track-Elements Track-Elemente sind in diesem Gleisnetz Streckenabschnitte, die einen Zug detektieren können.

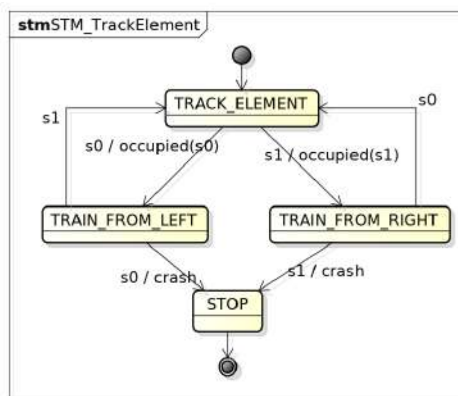


Abbildung 14: Zustandsautomat eines Track-Elementes

Befährt ein Zug dieses, wechselt der Zustandsautomat entweder in TRAIN_FROM_LEFT (wenn dieser von S_0 kommt) oder in den Zustand TRAIN_FROM_LEFT (wenn der Zug von S_1 kommt). Dabei löst dieser ein OCCUPIED-Signal für das jeweilige belegte Track-Element aus. Wenn jedoch ein Zug doppelt hineinfährt, wird ein CRASH-Signal ausgelöst, da dadurch ein Auffahrunfall auftritt.

4.2 Model-in-the-Loop (MiL) Tests

Model-in-the-Loop (MiL) Tests werden genutzt, um Modelle von sicherheitskritischen Systemen schon vor der eigentlichen Implementierung zu testen. Das hat den Vorteil, dass Fehler schon vor der Implementierungsphase erkannt werden und so hohe Kosten gespart werden. [Kne18]
Sicherheitskritische Systeme reagieren in Form von Sensordaten auf Eingangssignale aus der Umwelt. Diese werden zur Stimulierung der eigenen Software genutzt. Um das Echtzeitverhalten dieser Systeme zu testen, ist es nicht mehr nötig erst das implementierte System mit Testdaten zu stimulieren. Es ist möglich, schon in einem früheren Softwarestadium zu testen. So wird das Modell der Software, sowie die Sensordaten, die das Modell verarbeiten soll, in Echtzeit auf einer Echtzeitplattform simuliert.

Das hat zudem den weiteren Vorteil, dass damit für das Erzeugen von Sensordaten optimale Bedingungen existieren und dadurch ein realitätsnahes Testergebnis erzielt wird. [HWK+17]

Das Gleisnetz wird in den folgenden Abschnitten mit Hilfe eines MiL-Testverfahrens und dem Refinement Checker FDR4 getestet. Dazu wird zunächst beschrieben, wie Weichen, Kreuzweichen und Track-Elemente modelliert wurden (Kapitel 4.1). Im Anschluss soll das Verfahren anhand eines kleinen Gleisnetzes (Kapitel 4.2.1) beispielhaft gezeigt werden. Im Weiteren wird des großen Gleisnetzes (siehe Kapitel 4.2.2), wie in dem kleinen Beispiel gezeigt, getestet. Anschließend werden die Ergebnisse des Model-in-the-Loop-Tests ausgewertet.

4.2.1 Kleines Gleisnetz

Zur Veranschaulichung des MIL-Tests des Gleisnetzes wird zunächst das Vorgehen und die Implementierung anhand eines kleinen Gleisnetzes erklärt. Das kleine Gleisnetz wird im Folgenden auf alle Sicherheitsspezifikationen überprüft. Dabei werden Model-in-the-Loop-Tests angewendet.
Das kleine Gleisnetz hat die Form

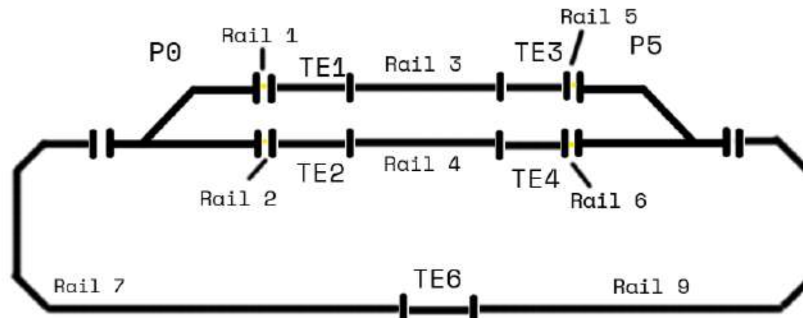


Abbildung 15: Kleines Gleisnetz

Um dieses Gleisnetz in CSP zu modellieren, wurden die Weichen und Track-Elements (kurz: TE) wie folgt aufeinander synchronisiert:

```
NETWORK = (( ((POINT(0,7,1,2) [|{rail.1}|] TRACK_ELEMENT(1,1,3))
    [|{rail.2}|] TRACK_ELEMENT(2,2,4))
    [|{ rail.3, rail.4 }|]
    ((POINT(5,9,5,6)
    [|{rail.5}|] TRACK_ELEMENT(3,3,5))
    [|{rail.6}|] TRACK_ELEMENT(4,4,6)) )
    [|{rail.7, rail.9}|] TRACK_ELEMENT(6,7,9))
```

Listing 6: Synchronisation der Streckenabschnitte

Dabei wird jeder Streckenabschnitt (Weiche oder Track-Element) über Verbindungen (hier: rail) verbunden. Abbildung 15 zeigt das, was in CSP_M modelliert wurde. Die rails in der Implementierung entsprechen denen aus der Abbildung 15.

Eine Weiche ist so implementiert, dass ein crash-Event dann ausgelöst wird, wenn die Weiche noch nicht korrekt gestellt wurde und trotzdem befahren wird. Es kann nun sein, dass die entsprechende Weiche für den Zug nicht freigegeben wurde, somit kommt es zum Crash. Anders, wenn die Weiche für die angeforderte Route gestellt wurde, dieser jedoch falsch hinauffährt, kommt es zum Entgleisen (derailing) des Zuges (siehe Abbildung 13). Listing 7 zeigt hierbei den MARKED und LOCKED_STR Zustand einer Weiche. Der gesamte CSP_M-Code befindet sich dabei im Anhang (siehe Listing 29).

```
MARKED_P(id, s0, s1, s2) =
    set.id.str -> P_LOCKED_STR(id, s0, s1, s2)
    []
    set.id.crss -> P_LOCKED_CRSS(id, s0, s1, s2)
    []
    ([] x : {s0, s1, s2} @ rail.x -> crash.id -> STOP)
    []
    reqsec.id -> derailing.id -> STOP

P_LOCKED_STR(id, s0, s1, s2) =
    rail.s2 -> P_LOCKED_STR_OCC_FROM_RIGHT(id, s0, s1, s2)
    []
    rail.s0 -> P_LOCKED_STR_OCC_FROM_LEFT(id, s0, s1, s2)
    []
    rail.s1 -> derailing.id -> STOP
    []
    release.id -> POINT(id, s0, s1, s2)
    []
    reqsec.id -> derailing.id -> STOP
    []
    set.id.str -> derailing.id -> STOP
    []
    set.id.crss -> derailing.id -> STOP
```

Listing 7: Marked- und Locking-Zustand einer Weiche

Im MARKED Zustand ist zunächst zu erkennen, dass durch "set.id.str/set.id.crss" die Weiche gestellt wird. Danach wird in den Zustand P_LOCKED_STR oder P_LOCKED_CRSS gewechselt, die dann jeweils unterscheiden, aus welcher Richtung der Zug einfahren darf und löst bei falscher Einfahrt einen Fehler (derailing/crash) aus. Wie im Zustand P_LOCKED_STR zu erkennen, ist es ausschließlich erlaubt, dass der Zug von rail.s2 oder rail.s0 einfährt. Fährt der Zug jedoch von der falschen Seite der Weiche hinein (Cross-Richtung rail.s1), so wird das entsprechende derailing-Event ausgelöst. Das derailing-Event wird in diesem Zustand ebenfalls ausgelöst, falls im geradeaus-gesperrten Zustand P_LOCKED_STR die Weiche noch einmal angefragt oder gestellt wird.

Track-Elemente sind hingegen so implementiert, dass sich zur selben Zeit nur ein Zug

befindet. Dabei wird das Event "occupied" erzeugt, welches den Belegungszustand des Track-Elements zeigt. Listing 8 zeigt einen Ausschnitt aus dem CSP_M-Code des Track-Elements. Der gesamte Code befindet sich im Anhang (siehe Listing 30).

```
TRACK_ELEMENT(id, s0, s1) =
    rail.s0 -> occupied.id -> TRAIN_FROM_LEFT(id, s0, s1)
    []
    rail.s1 -> occupied.id -> TRAIN_FROM_RIGHT(id, s0, s1)
    []
    free.id -> TRACK_ELEMENT(id, s0, s1)

TRAIN_FROM_LEFT(id, s0, s1) =
    rail.s1 -> TRACK_ELEMENT(id, s0, s1)   -- drive through
    []
    rail.s0 -> crash.id -> STOP
    []
    release.id -> free.id -> TRACK_ELEMENT(id, s0, s1)
```

Listing 8: Init- und Train_from_left-Zustand

Jedes Track-Element kann im Vergleich zu einer Weiche ohne Anfragen befahren werden. Dementsprechend ist der Zustandsautomat vergleichbar klein.
Befährt nun ein Zug das Track-Element, so wird ein occupied-Event ausgelöst und im Folgezustand gewechselt. Wie im Zustand TRAIN_FROM_LEFT zu sehen, darf ein Zug, wenn dieser von links auf das Track-Element fährt, nur rechts wieder hinausfahren. Andernfalls kann es sein, dass ein anderer Zug von hinten in den Zug fährt.
Auf diesem kleinen Gleisnetz soll nun detailliert der Model-in-the-Loop Test gezeigt werden. Dieses Gleisnetz soll mit Hilfe von zwei Zügen befahren werden. Ein Zug befährt das Netz über Rail 1, 3 und 5, der andere über Rail 2, 4 und 8.
Dazu wurden zunächst zwei Züge deklariert, die auf eine movement authority warten und dann die vordefinierte Strecke über die jeweiligen Gleise fahren. Das Stellwerk sperrt die korrespondierenden Abschnitt und erteilt nach erfolgreichen Sperren der Route eine movement authority (movement_auth), welche dem Zug die Fahrterlaubnis erteilt.

```
TRAIN1 = movement_auth.0 -> rail.7 -> rail.2 -> rail.4 -> rail.6 -> rail.9 ->
  STOP
TRAIN2 = movement_auth.1 -> rail.7 -> rail.1 -> rail.3 -> rail.5 -> rail.9 ->
  STOP

IXL1 = reqsec.0 -> set.0.str -> reqsec.5 -> set.5.str -> movement_auth.0 -> STOP
IXL2 = reqsec.0 -> set.0.crss -> reqsec.5 -> set.5.crss -> movement_auth.1 ->
  STOP

TEST1 = TRAIN1 [|{| movement_auth |}|] IXL1
TEST2 = TRAIN2 [|{| movement_auth |}|] IXL2

SYSTEM1 = NETWORK [|{|reqsec, set, rail, release|}|] TEST1
SYSTEM2 = NETWORK [|{|reqsec, set, rail, release|}|] TEST2

assert STOP [T= SYSTEM1 \ diff(Events, {| crash, derailing |})
assert STOP [T= SYSTEM2 \ diff(Events, {| crash, derailing |})
```

Listing 9: Erster MiL Test

Die beiden Assertions sollen nun prüfen, ob beim Befahren des Gleisnetzes kein crash- oder derailing-Event erzeugt wird. Durch die diff-Anweisung wird erreicht, dass alle auftretbaren Events, außer crash und derailing, verborgen werden. Dadurch wird mit der Assertion wert darauf gelegt, dass der Test (SYSTEM1/SYSTEM2) stoppt (es wird kein Event erzeugt), wobei crash- und derailing-Events erzeugt werden dürfen. Sobald eines der beiden Events auftritt, schlägt der Test fehl. Es wird erwartet, dass der Test mit PASS abschließt, da der Test einer validen Fahrt auf dem realen Gleisnetz entspricht. Die beiden Tests schließen dabei mit folgendem Ergebnis ab:

```
STOP [T= SYSTEM1 \ diff(Events, {| crash, derailing |}): Passed
STOP [T= SYSTEM2 \ diff(Events, {| crash, derailing |}): Passed
```

Listing 10: Testergebnis vom kleinen Gleisnetz

Um nun wiederum zu zeigen, dass der Test bei einem fehlerhaft in CSP_M implementierten Modell fehlschlägt, wird eine Weiche falsch eingebaut.

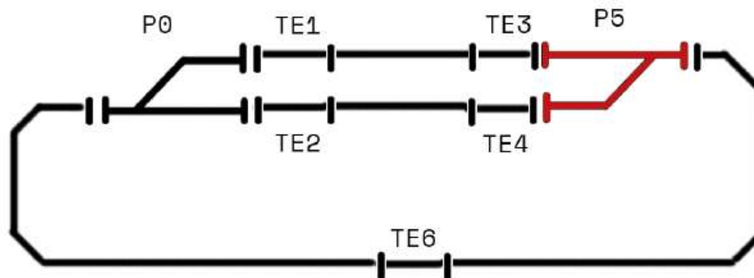


Abbildung 16: Fehler bei einer Weiche

Wie in Abbildung 16 zu sehen wird nun eine Weiche absichtlich falsch eingebaut, um zu zeigen, dass nun der MiL-Test fehlschlägt. Dadurch, dass nun die Weiche P5 falsch gestellt wird, kann erwartet werden, dass der Test fehlschlägt und ein derailing-Event an der Weiche P5 erzeugt wird.
Das Ergebnis des Tests ist Folgendes:

```
STOP [T= SYSTEM1 \ diff(Events, {|crash, derailing|}):
    Log:
        Result: Failed
        Visited States: 17
        Visited Transitions: 89
        Visited Plys: 11
        Estimated Total Storage: 268MB
        Counterexample (Trace Counterexample)
        Specification Debug:
        Trace: <>
        Available Events: {}
        Implementation Debug:
        (Unnamed) (Trace Behaviour):
        Trace: <...>
        Error Event: derailing.5
```

```
STOP [T= SYSTEM2 \ diff(Events, {|crash, derailing|}):
    Log:
        Result: Failed
        Visited States: 17
        Visited Transitions: 89
        Visited Plys: 11
        Estimated Total Storage: 268MB
        Counterexample (Trace Counterexample)
        Specification Debug:
        Trace: <>
        Available Events: {}
        Implementation Debug:
        (Unnamed) (Trace Behaviour):
        Trace: <...>
        Error Event: derailing.5
```

Listing 11: Testergebnis bei fehlerhafter Implementierung

Wie erwartet, wird ein derailing-Event an der Weiche P5 erzeugt (derailing.5). Durch den fehlgeschlagenen Tests ist gezeigt, dass diese bei einer Fehlimplementierung dies auch erkennen. Dieses Ergebnis zeigt nun notwendigerweise, dass der Test korrekt ist und die CSP_M-Implementierung dem realen Gleisnetz entspricht.

4.2.2 Tests für das gesamte Gleisnetz

Nun soll das gesamte große Gleisnetz durch Model-in-the-Loop Tests verifiziert werden. Dadurch soll sichergestellt werden, dass das in CSP_M modellierte Gleisnetz dem realen Märklin-Gleisnetz (Abbildung 3) entspricht. Ebenso soll dadurch garantiert werden, dass das Schienennetz sicher ist und für den sicheren Bahnverkehr geeignet ist. Die Ergebnisse der jeweiligen Tests belegen dann die zuvor aufgestellte Behauptung.
Das Gleisnetz, wie in Abbildung 3 zu sehen, wird in CSP_M durch Weichen, Kreuzweichen, Track-Elements und Verbindungen (rail) modelliert. Die Darstellung der CSP_M Implementierung befindet sich dabei im Anhang.
Es gibt unzählige Möglichkeiten, das Gleisnetz zu befahren, dabei werden jedoch nur auf drei verschiedene Arten von Tests durchgeführt. Dafür werden immer jeweils unterschiedliche Routen befahren, ähnlich wie sie in der Interlocking-Table definiert wurden. Dabei werden so viele Routen befahren, dass jede Strecke einmal befahren wird. Es soll getestet werden:

- Zwei Züge befahren je eine Route des Gleisnetzes ohne zu kollidieren,
- Zwei Züge kollidieren, weil sie auf die gleiche Weiche fahren,
- Ein Zug biegt falsch ab (Es soll ein crash-Event ausgelöst werden).

Dabei wird das Gleisnetz ähnlich wie am Beispiel des kleinen Gleisnetzes getestet.

Zwei Züge befahren konfliktlos das Gleisnetz Bei diesem Test befahren zwei Züge das Gleisnetz. Zug 1 befährt die rote, Zug 2 befährt die grüne Route (siehe Abbildung 17). Dabei werden zunächst alle Weichen in richtiger Position gestellt. Wenn diese erfolgreich gesperrt wurden, wird dem Zug, der die Route befahren soll, eine movement authority erteilt, wodurch dieser losfährt. Die CSP_M Implementierung ist dabei folgende:

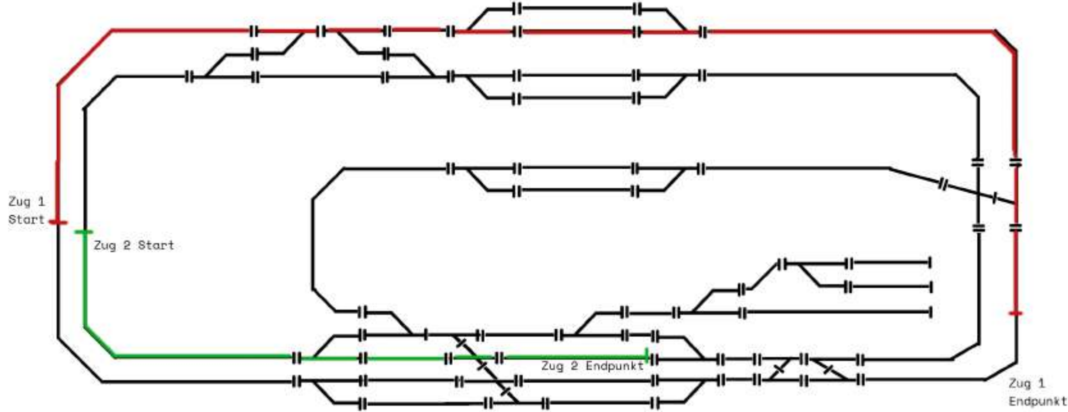


Abbildung 17: Test 1: Züge befahren die Route

```
TRAIN1 = movement_auth.0 -> rail.1 -> rail.78 ->
            rail.3 -> rail.7 ->
            rail.10 -> rail.14 ->
            rail.18 -> rail.21 ->
            rail.23 -> rail.27 -> STOP

TRAIN2 = movement_auth.1 -> rail.76 -> rail.73 ->
            rail.57 -> rail.49 -> STOP

IXL1 = reqsec.16 -> reqsec.17 -> reqsec.20 ->
            reqsec.21 -> reqsec.24 -> reqsec.13 ->
            reqsec.10 -> set.16.str -> set.17.str ->
            set.20.str -> set.21.str -> set.24.str ->
            set.13.str -> set.10.str ->
            movement_auth.0 -> movement_auth.1 -> STOP

TEST1 = (TRAIN1 ||| TRAIN2) [|{ movement_auth.0, movement_auth.1 }|] IXL1

SYSTEM_VALID = RAILWAY_NETWORK [|{| rail, set, reqsec, release |}|] TEST1

assert STOP [T= SYSTEM_VALID \ diff(Events, {|crash|})
```

Listing 12: MiL-Test: Zwei Züge befahren das Gleisnetz

Jeder Zug (TRAIN1 und TRAIN2), sowie das Stellwerk (IXL1) stellen dabei drei unabhängige Prozesse dar, die wie im realen System auf unterschiedlicher Hardware ausgeführt werden. Der Testfall TEST1 führt die beiden Züge in parallel aus (Operator |||) und synchronisiert diese mit dem Stellwerk (IXL1) auf die Events des Kanals movement_auth. Das bedeutet, dass die Züge auf die jeweiligen Events des Kanals movement_auth vom Stellwerk hören und darauf reagieren.

Beim SYSTEM_VALID wird schließlich der Testfall TEST1 mit dem Gleisnetz über rail, set, reqsec, release verbunden. Das sind die Events, die vom Stellwerk und den Zügen ausgelöst werden, worauf das Gleisnetz dann reagieren soll. Die Events des Kanals rail sind dabei die Gleise, die vom Zug befahren werden (siehe Abbildung 17).
Zu guter Letzt wird mit Hilfe der Assertion geprüft, ob das gegebene System ein STOP-Event erzeugt. Dabei werden alle Events, außer crash, ausgeblendet. Das hat die Aufgabe, dass in diesem Fall alle anderen Events für diesen Test irrelevant sind. Diese werden zwar erzeugt (indem sie intern zu τ-Events umgewandelt werden), jedoch für den Test nicht beachtet.
Dieser Test sollte kein crash-Event auslösen, wenn das Gleisnetz korrekt ist. Somit kann ein STOP-Event erwartet werden. Wird jedoch doch ein crash-Event erzeugt, ist die CSP_M-Implementierung falsch.

Das Ergebnis ist:

```
STOP [T= SYSTEM_VALID \ diff(Events, {|crash|}): Passed
```

Listing 13: Ergebnis MiL-Test

Dieser Test war erfolgreich, was bedeutet, dass die Züge das Gleisnetz erfolgreich befahren konnten. Das Gleisnetz entspricht dabei dem realen.

Zwei Züge kollidieren, weil sie auf dieselbe Weiche fahren Hierbei werden die Züge ähnlich wie im vorherigen Test deklariert, jedoch enden diese fälschlicherweise auf dieselbe Weiche. Das darf auf keinen Fall passieren, da dadurch diese Züge kollidieren. Hierbei soll auf jeden Fall ein crash-Event erzeugt werden.
Die CSP_M-Modellierung ist dabei folgende:

```
TRAIN3 = movement_auth.2 ->
    rail.69 -> rail.68 -> rail.67 -> rail.65 ->
    rail.63 -> rail.61 -> rail.60 -> rail.26 ->
    STOP

IXL2 = reqsec.16 -> reqsec.17 -> reqsec.20 -> reqsec.21 -> reqsec.24 ->
    set.16.str -> set.17.str -> set.20.str -> set.21.str -> set.24.str ->
    reqsec.25 -> reqsec.26 -> reqsec.23 ->
    set.25.crss -> set.26.crss -> set.23.str ->
    movement_auth.0 -> movement_auth.2 -> STOP

TEST2 = (TRAIN1 ||| TRAIN3) [|{| movement_auth |}|] IXL2

SYSTEM_CRASH = NETWORK [|{| rail, set, reqsec, release |}|] TEST2

SPEC1 = crash.24 -> STOP

assert not STOP [T= SYSTEM_CRASH \ diff(Events, {|crash|})

assert SPEC1 [T= SYSTEM_CRASH \ diff(Events, {|crash|})
```

Listing 14: MiL-Test: Zwei Züge fahren auf eine Weiche und kollidieren

Hierbei ist nur die verkürzte Darstellung angegeben. Beide Züge enden auf der Weiche 24. Daher wird das Event crash.24 erwartet. SPEC1 zeigt dabei diese Spezifikation. Zunächst wird getestet, dass das System (SYSTEM_CRASH) nicht stoppt. Da, wie im letzten Testfall, alle Events außer crash ausblendet werden, wird bei "not STOP" ein crash-Event ausgelöst. Das belegt die aufgestellte Spezifikation hinreichend. Die Spezifikation SPEC1 liefert hingegen die notwendige Bedingung, dass auch tatsächlich an der Weiche das crash-Event erzeugt wird.
Wenn beide Behauptungen erfolgreich abschließen, ist garantiert, dass die Kollision der beiden Züge überhaupt und am richtigen Punkt erkannt werden.
Das Ergebnis

```
not STOP [T= SYSTEM_CRASH \ diff(Events, {|crash|}): Passed
SPEC1 [T= SYSTEM_CRASH \ diff(Events, {|crash|}): Passed
```

Listing 15: Ergebnis Test

zeigt, dass die Spezifikation gilt und, dass das Gleisnetz im Sinne des Testfalls sicher ist.

Falsches Abbiegen eines Zuges Ebenso muss geprüft werden, ob das falsche Abbiegen eines Zuges registriert wird. Diese Eigenschaft ist zentral für die Sicherheit des Gleisnetzes, wodurch ebenfalls Weichenfehler erkannt werden können. Wie im echten System wird beim falschen Abbiegen ein Notstopp für alle Züge ausgelöst, welches durch ein crash-Event ausgelöst wird.
Um den Test zu modellieren, fährt ein Zug auf eine in Cross-Richtung (quer-Richtung) eingestellte Weiche(Weiche 14), fährt diese jedoch in Straight-Richtung (geradeaus-Richtung) heraus. Dabei wird ein crash.14-Event erwartet. Die CSP_M-Modellierung ist dabei:

```
TRAIN4 = movement_auth.3 -> rail.77 -> rail.74 -> STOP

IXL3 = reqsec.14 -> reqsec.8 -> reqsec.6 ->
    set.14.crss -> set.8.str -> set.6.crss ->
    movement_auth.3 -> STOP

TEST3 = TRAIN4 [|{| movement_auth |}|] IXL3

SYSTEM_WRONG_TURN = NETWORK [|{| rail, set, reqsec, release |}|] TEST3

SPEC2 = crash.14 -> STOP

assert not STOP [T= SYSTEM_WRONG_TURN \ diff(Events, {|crash|})

assert SPEC2 [T= SYSTEM_WRONG_TURN \ diff(Events, {|crash|})
```

Listing 16: Falsches Abbiegen eines Zuges

Zunächst wird wieder behauptet, dass das System nicht stoppt, also ein crash-Event auslöst. In der zweiten Behauptung wird schließlich untersucht, ob das System ein

crash.14-Event erzeugt.
Gelten beide Behauptungen, bestehen also beide mit **Passed**, so ist hinsichtlich des Tests das Gleisnetz sicher.

Das Ergebnis ist:

```
not STOP [T= SYSTEM_WRONG_TURN \ diff(Events, {|crash|}): Passed
SPEC2 [T= SYSTEM_WRONG_TURN \ diff(Events, {|crash|}): Passed
```

Listing 17: Ergebnis falsches Abbiegen

zeigt, dass das System hinsichtlich des Tests sicher ist.

4.3 Modell Validierung des gesamten Systems

Im Folgenden sollen nun Modelle von sicherheitskritische Systemkomponenten, wie das Modell des Sub-Controllers und der generelle Ablauf im laufenden Betrieb, getestet werden. Dabei soll gezeigt werden, dass das System in jedem Zustand sicher ist und den Zug immer richtig durch die jeweilige von ihm angefragte Route leitet. Ebenso soll gezeigt werden, dass die jeweils in Ausführungszeit parallel laufenden Controller keine Sicherheitsverletzung erzeugen.
Zum Testen des Systems war es nötig, vorher einen geeigneten Testplan zu erstellen mit begleitender Teststrategie. Die entworfene Teststrategie umfasst dabei nicht das Model Checking des gesamten Systems mit allen Controllern des Stellwerkes (siehe Abbildung 5), sondern nur einen bestimmten Teil. Für das sogenannte Sub-Model Checking werden nur die für das Fahren eines Zuges benötigte Komponenten verifiziert. Der Health-Controller, der auf Lebenszeichen von Zügen und vom Positionssystem hört wurde hier ausgelassen, da dieser für die Fahrt eines Zuges im Detail zunächst irrelevant ist.
Stattdessen wird der Sub-Controller, der Train-Controller, sowie das Gleisnetz und das Verhalten von Zügen zum Stellwerkssystem genauer betrachtet.

4.3.1 Sub-Controller Tests

Jeder Sub-Controller steuert genau eine Route und begleitet einen Zug sicher über das Gleisnetz zum gewünschten Ziel. Dabei stellt ein Sub-Controller die Weichen, der korrespondierende Route, in die richtige Position und sperrt diese. Wurden die Weichen erfolgreich gestellt und gesperrt, werden anschließend ebenso alle Konflikrouten gesperrt. Eine Konfliktroute ist dabei jene Route, deren Pfad sich mit dieser Route überschneidet und somit ebenfalls dieselben Weichen stellen muss. Diese sind nun nicht mehr in der Lage die eigenen Weichen zu stellen (siehe Kapitel 2.2.2). Diese Tatsache verhindert das erneute Stellen einer Weiche, was eine Entgleisung des Zuges zur Folge hätte. Im Folgenden soll bewiesen werden, dass diese Spezifikation gilt.

Dazu werden zwei im Konflikt stehende Sub-Controller zur selben Zeit angefragt. Es soll untersucht werden, ob ein derailing-Event erzeugt wird. Dies zeigt, dass eine Weiche ein zweites Mal gestellt wurde, ohne dass diese vorher von einem Zug befahren wurde. Gleichzeitig soll geprüft werden, ob ein crash-Event erzeugt wird. Das kann auftreten, falls Weichen ein zweites Mal gestellt werden und ein Zug dann wiederum falsch abbiegt und dann folglich eine Kollision verursacht. Zum Testen dieser Eigenschaft wird das Modell des Sub-Controllers (Abbildung 7) als Implementierung in CSP_M verwendet. Der Test ist dabei wie in Abbildung 18 aufgebaut. Listing 18 beschreibt dabei die Implementierung des Testfalls. Der Quellcode befindet sich dabei auf dem beigelegten Speichermedium.

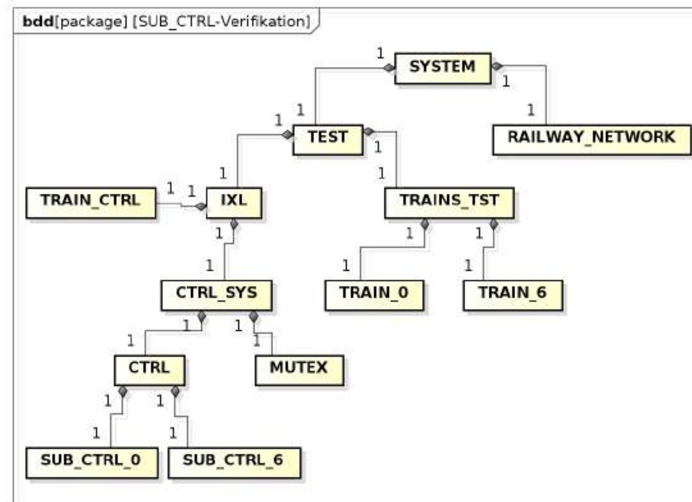


Abbildung 18: Testaufbau bei der Verifikation der Sub-Controller

TRAINS_TST ist dabei Train 0 und Train 6 in Parallelität ohne jegliche Synchronisation. Jeder Zug interagiert dabei völlig willkürlich. CTRL ist dabei die Synchronisation vom Sub-Controller 0 mit dem Sub-Controller 6 auf die Events conflict, release_active_wait, release_conflict, damit sich die Sub-Controller gegenseitig in Konflikt setzen können und auch wieder herauslösen können. IXL ist dabei der Train-Controller, der mit den jeweiligen Sub-Controllern synchronisiert ist. Da sich zunächst jeder Zug beim Train-Controller meldet, der dann wiederum die Route bei dem jeweiligen Sub-Controller anfragt und dann Routen-relevante Daten übermittelt, ist dieser über request_route und param mit den Sub-Controllern synchronisiert (siehe Kapitel 2.2.2).

```
TRAINS_TST = (TRAIN_0 ||| TRAIN_6)

CTRL = SUB_CTRL_0 [| S(0, 6) |] SUB_CTRL_6

CTRL_SYS = MUTEX [| {| mutex_lock, mutex_unlock |} |] CTRL
IXL = TRAIN_CTRL [| {| request_route, param |} |] CTRL_SYS
TEST = IXL [| {| movement_auth, request |} |] TRAINS_TST

SYSTEM = RAILWAY_NETWORK [| {| rail, set, reqsec, release, occupied |} |] TEST

assert STOP [T= SYSTEM \ diff(Events, {| crash, derailing |} )
```

Listing 18: Sub-Controller Test

Der Testfall ist dann die Synchronisation der Züge mit dem Stellwerk, die dieses wie oben beschrieben stimuliert. Ein Zug requested (fordert an) eine Route und erhält nach erfolgreichen Sperren der Route eine movement_authority, die dem Zug die Fahrterlaubnis zuweist. Zu guter Letzt bildet das Gleisnetz zusammen mit dem Testfall das Gesamtsystem ab. "rails" sind die Gleisabschnitte, die vom Zug befahren werden. set, reqsec und release beschreiben das Setzen, das Anfragen und das Lösen einer Weiche, wobei occupied den Zustand eines Track-Elements darstellt.
Die abschließend zu testende Bedingung ist, dass das System stoppen und dabei nicht die Events crash und derailing erzeugen soll. Wörtlich ausgedrückt heißt es, dass Stopp Trace-verfeinert wird vom System, wobei alle Events außer crash und derailing versteckt werden. Somit wird nur auf die beiden Events geachtet. Wenn eins der beiden Events auftritt, ist gezeigt, dass das System in der aktuellen Modellierung nicht sicher ist.

Das Ergebnis ist jedoch:

```
STOP [T= SYSTEM \ diff(Events, {|crash, derailing|}):
    Log:
        Result: Failed
        Visited States: 370
        Visited Transitions: 14.307
        Visited Plys: 32
        Estimated Total Storage: 285MB
        Counterexample (Trace Counterexample)
        Specification Debug:
        Trace: <>
        Available Events: {}
        Implementation Debug:
        (Unnamed) (Trace Behaviour):
        Trace: <...>
        Error Event: derailing.2
```

Listing 19: Ergebnis Subcontroller-Test

Das zeigt, dass es so in diesem Zustand zu Entgleisungen kommen kann.
Nach Debuggen des Fehlers und Anzeigen des Gegenbeispiels zur Assertion von FDR4 wird klar, dass es im System einen Fehler im Sub-Controller gibt. Der Fehler entsteht, wenn der Mutex bei Sperren von Weichen und Konfliktrouten zwischenzeitig gelöst wird.

```
LOCKED_X(train) = free.212 -> ... ->
        mutex_lock -> reqsec.2 -> set.2.str -> ... -> mutex_unlock ->
          LOCKING_CONFLICTS_0(train)

LOCKING_CONFLICTS_X(train) = mutex_lock -> conflict.6 -> mutex_unlock
-> movement_auth.train -> occupied.212 -> OCCUPIED_0_0(train)
```

Listing 20: Fehler im LOCKING Zustand

Im Zustand LOCKED werden zunächst alle Weichen gestellt und gesperrt. Dieses wird mit einem Mutex geschützt, der jedoch nach Sperren aller Weichen wieder gelöst wird. Direkt danach im Folgezustand werden alle Konfliktrouten gesperrt und der Mutex wird wieder gesperrt. Durch das zwischenzeitliche entsperren des Mutex ist es möglich, dass andere Sub-Controller den Mutex locken können und dann Weiche erneut stellen, bevor sie in Konflikt gesetzt werden. Dies ist eine Sicherheitsverletzung und muss geändert werden. Die Lösung des Problems ist, dass der Mutex zwischenzeitlich nicht mehr entsperrt wird, sondern erst am Ende, wenn alle Konfliktrouten gesperrt sind wieder entsperrt wird. Dadurch kann keine Konfliktroute Weichen doppelt stellen. Das verbesserte Modell ist in Listing 21 zu sehen.

```
LOCKED_0(train) = free.212 -> ... -> mutex_lock -> reqsec.2 -> set.2.str
-> ... -> conflict.6 ->
mutex_unlock -> movement_auth.train -> occupied.212 -> OCCUPIED_0_0(train
  )
```

Listing 21: Lösung des Fehlers

Das dazu korrespondieren Testergebnis ist schließlich folgendes:

```
STOP [T= SYSTEM \ diff(Events, {|crash, derailing|}): Passed
```

Listing 22: Ergebnis mit neuem Modell

4.3.2 System-Tests

Das Stellwerk soll nun als ganzes verifiziert werden. Dabei soll zentral die Sicherheit der Züge in Betracht gezogen werden. Wichtig dabei ist, dass das System im aktuellen Zustand keine Kollision und kein Entgleisen von Zügen verursacht. Als fundamentales Dokument ist dabei die Interlocking-Table, die nicht nur jede Route und die jeweils vordefinierte Strecke speichert, sondern auch Konfliktrouten und Weichenstellungen beinhaltet. Somit spielt die Verifikation der Interlocking-Table eine klare Bedeutung für die Sicherheit des Gesamtsystems.
Im Kapitel 4.3.1 wurde nun schon gezeigt, dass in Konflikt stehende Sub-Controller generell in paralleler Ausführung Kollisionen oder Entgleisungen erzeugt hätten. Durch Korrigieren der Modellierung ist der Fehler nun behoben. Doch in der Regel werden nicht nur die beiden oben genannten Routen befahren, sondern zufällig, bis zu vier

parallel. Im Bezug zur Interlocking-Table ergibt sich die Fragestellung, ob die oben genannte Behauptung auch für alle Sub-Controller gilt. Kann also bewiesen werden, dass das System mit der generierten Interlocking-Table sicher ist und, dass keine Kollisionen oder Entgleisungen auftreten, wenn bis zu vier Züge parallel zufällig Routen befahren wollen?
Die Interlocking-Table ist wie in Abbildung 4 definiert. Jede Route besitzt neben einer eindeutigen ID, einen Pfad, den der Zug befährt, benötigte Weichen-Positionen, Länge und Maximalgeschwindigkeit und eine Liste von Routen, mit der diese in Konflikt steht.

Getestet werden soll nun, ob,

1. jede Route die Weichen korrekt im Bezug zu dem zu befahrenen Pfad stellt und

2. alle Konfliktrouten korrekt aufgezählt wurden, oder, ob es Routen gibt, die nicht in den Konfliktrouten genannt wurden, jedoch dazu zählen.

Der Testaufbau ist wie in Abbildung 19 gezeigt.

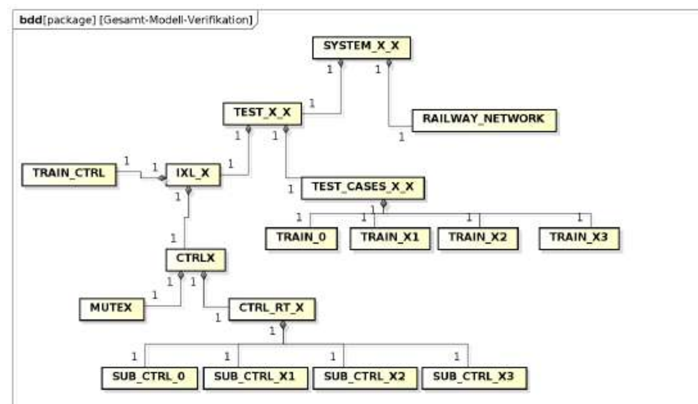


Abbildung 19: Testaufbau für Tests des gesamten Systems

TRAIN_0 steht dabei immer für den Zug, der die zu testende Route befahren soll, und SUB_CTRL_0 für den korrespondierenden Sub-Controller. TRAIN_X1, TRAIN_X2, TRAIN_X3 stehen dabei für die jeweiligen Züge, die drei Konfliktrouten (beim Testen von Konfliktrouten) oder drei Nicht-Konfliktrouten (beim Test von Nicht-Konfliktrouten) befahren. Gleiches gilt für SUB_CTRL_X1, SUB_CTRL_X2, SUB_CTRL_X3. Diese sind jeweils Sub-Controller für Konfliktrouten (beim Test von Konfliktrouten), oder Sub-Controller von Nicht-Konfliktrouten (beim Test von Nicht-Konfliktrouten). Dieser allgemeine Testaufbau veranschaulicht, wie nachfolgende Tests aufgebaut sind. Dabei gibt es jedoch kleine Veränderungen, je nachdem, welcher Test gerade durchgeführt wird.

Test der Weichenstellung Um nun zu prüfen, ob für jede Route die entsprechenden Weichen korrekt gestellt werden, genügt ein trivialer Ansatz. Jede Route wird von einem Zug befahren. Dabei fordert jeder Zug eine Route an und befährt diese. Dabei gibt es für jede Route genau einen Zug, der diese befährt. Nacheinander wird jede Route so einmal befahren. Dieser Testaufbau ähnelt dem aus Abbildung 19, jedoch gibt es nur den SUB_CTRL_0 und den TRAIN_0, SUB_CTRL_X1, SUB_CTRL_X2, SUB_CTRL_X3, sowie TRAIN_X1, TRAIN_X2, TRAIN_X3 existieren dabei nicht.

```
include "mutex.csp"
include "controller.csp"

TRAIN_CTRL = ([] x : ROUTES, y : TRAINS @ request.x.y ->
        request_route.x.y -> param.x-> TRAIN_CTRL)

TRAIN_0 = request.0.0 -> movement_auth.0 -> rail.28
        -> rail.31 -> rail.34 -> rail.36 -> rail.38
        -> rail.48 -> rail.52 -> rail.56 -> rail.70
        -> rail.69 -> STOP

CTRL0 = MUTEX [| {| mutex_lock, mutex_unlock |} |] SUB_CTRL_0
IXL_0 = TRAIN_CTRL [| {| request_route, param |} |] CTRL0

TEST_0 = IXL_0[| {| movement_auth, request |} |] TRAIN_0
SYSTEM_0 = RAILWAY_NETWORK
            [| {| rail, set, reqsec, release, occupied |} |]
            TEST_0

assert STOP [T= SYSTEM_0 \ diff(Events, {|crash, derailing |})
```

Listing 23: CSP-Codeausschnitt aus den Tests

Dabei werden zum Test alle Events, außer crash und derailing, verborgen. Diese beiden Events deuten darauf hin, dass der Zug entweder falsch abgebogen ist, oder entgleist ist. Hierbei steht crash immer dafür, dass entweder zwei Züge kollidierten oder ein Zug an einer Weiche falsch abgebogen ist, weil diese falsch vom Stellwerk gestellt wurde. Das derailing-Event steht dabei dafür, dass die Weiche für den ankommenden Zug falsch gestellt ist.
Es soll nun getestet werden, ob beim Ausführen dieses Testes das Event crash und/oder derailing nicht auftreten und das System somit in den STOP-Zustand gelangt. Das Verbergen der Events außer crash und derailing bedeutet, dass diese Events intern generiert werden (Umwandlung zu τ) und die Zustandsmaschinen somit korrekt stimuliert werden. Für den Test hingegen sind diese Events (wie zum Beispiel rail oder movement_auth) bedeutungslos. Falls jedoch entweder crash oder derailing erzeugt werden, liegt ein Fehler vor. STOP sagt dabei aus, dass das System terminiert ist. Folglich, wenn kein crash- und kein derailing-Events erzeugt wurde und das System stoppt, gilt die Bedingung.

Ergebnis Nach Ausführen aller Tests wurden jedoch Fehler in vier Routen erkannt, die ein Entgleisen von Zügen verursacht hätten. Die Routen **38**, **39**, **40**, **78** waren dabei falsch spezifiziert. Der Sub-Controller der jeweiligen Routen stellte immer die Weiche 27 für jede Route in die falsche Position, wie anhand des Listings 4.3.2 beim Ergebnis der Route 38 zu sehen.

```
STOP [T= SYSTEM_38 \ diff(Events, {|crash, derailing|}):
    Log:
        Result: Failed
        Visited States: 30.441
        Visited Transitions: 1.227.237
        Visited Plys: 49
        Estimated Total Storage: 570MB
        Counterexample (Trace Counterexample)
        Specification Debug:
        Trace: <>
        Available Events: {}
        Implementation Debug:
        (Unnamed) (Trace Behaviour):
        Trace: <...>
        Error Event: derailing.27
```

Listing 24: Fehlgeschlagender Test

Somit bog entweder ein Zug falsch ab, was zu einer Kollision führen könnte, oder befuhr die Weiche in die falsche Richtung und entgleiste.

Test der Konfliktrouten Konfliktrouten haben im Stellwerk eine wichtige Bedeutung. Diese sorgen dafür, dass Weichen nicht doppelt gestellt werden, oder diese nicht doppelt im gleichen Zeitintervall befahren werden (Kollision von Zügen). Diese sorgen dafür, dass Züge eine Route sicher ohne Kollision mit anderen Zügen oder Entgleisungen passieren können. Genauer gesagt ist eine Route R_0 im Konflikt zu einer Route R_1, wenn diese dieselben Weichen befahren und/oder dieselben Track-Elemente überqueren. Darum ist es umso wichtiger zu verifizieren, dass alle Konfliktrouten für eine Route aufgezählt wurden.
Zunächst wird geprüft, ob es sicher ist, wenn vier Züge das Gleisnetz gleichzeitig befahren, wobei drei der Züge auf Konfliktrouten vom vierten Zug fahren. Für die Tests wird immer die zu testende Route befahren, sowie immer drei Konfliktrouten parallel. Dabei werden doppelte missachtet, um ein korrektes Testergebnis schnell zu erzeugen. Listing 25 zeigt ein Aufbau eines solchen Tests:

```
CTRL0 = MUTEX [| {| mutex_lock, mutex_unlock |} |] CTRL_RT_0
IXL_0 = TRAIN_CTRL [| {| request_route, param |} |] CTRL0

TEST_0_0 = IXL_0[| {| movement_auth, request |} |] TEST_CASE_0_0
SYSTEM_0_0 = RAILWAY_NETWORK [| {| rail, set, reqsec, release, occupied
  |} |] TEST_0_0
assert STOP [T= SYSTEM_0_0\ diff(Events, {|crash, derailing |})
```

Listing 25: Test Konfliktrouten

CTRL_RT_0 beschreibt hierbei die für den Test benötigten Sub-Controller. In diesem Beispiel ist es der Sub-Controller 0 für die Route 0, sowie die Sub-Controller für die ersten drei in der Interlocking-Table vorkommenden Konfliktrouten. Diese synchronisieren sich mit mutex_lock und mutex_unlock auf den einen Mutex, der das Sperren und Stellen der Weiche schützt. Das Stellwerk ist folglich der Train-Controller in Synchronisation zu den für den Test benötigten Controllern (CTRL0).
Der Testfall (Test_0_0) bildet das Stellwerk in Synchronisation zu den benötigten Zügen (TEST_CASE_0_0) für den Test ab. Dabei sind es genau die Züge, die die jeweiligen Sub-Controller benutzen und die jeweiligen Konfliktrouten befahren. Das Testsystem ist schließlich der Test (Stellwerk mit Zügen) in Synchronisation zum Gleisnetz.
Abschließend soll wieder getestet werden, ob das Testsystem stoppt, oder ob es die Events crash oder derailing erzeugt.

Ergebnis Das Ergebnis aller Test war erfolgreich, alle Tests endeten mit **PASS** und bestätigten die Erwartungen. Somit ist das Konzept der Konfliktrouten hinreichend als sicher einzustufen. Der vollständige Testbericht befindet sich auf das beigelegte Speichermedium.

Test der nicht in Konflikt stehenden Routen Um jedoch zu verifizieren, dass alle Konfliktrouten genannt wurden, muss getestet werden, ob alle Nicht-Konfliktrouten sicher befahren werden können. Der Testablauf ist äquivalent wie oben, nur, dass anstatt Konfliktrouten, Nicht-Konfliktrouten benutzt werden. Listing 26 zeigt einen solchen Test:

```
MTX_CTRL_NEG_0_0 = MUTEX [| {| mutex_lock, mutex_unlock |} |] CTRL_NEG_0_0
IXL_NEG_0_0 = TRAIN_CTRL [| {| request_route, param |} |] MTX_CTRL_NEG_0_0

TEST_NEG_0_0 = IXL_NEG_0_0
                [| {| movement_auth, request |} |]
               TEST_CASE_NEG_0_0
SYSTEM_NEG_0_0 = RAILWAY_NETWORK
                    [| {| rail, set, reqsec, release, occupied |} |]
                 TEST_NEG_0_0
assert STOP [T= SYSTEM_NEG_0_0 \ diff(Events, {|crash, derailing |})
```

Listing 26: Test mit Nicht-Konfliktrouten

Wieder werden alle Routen in Verbindung mit immer drei nicht im Konflikt stehenden Routen getestet. Falls wieder kein crash oder derailing Events erzeugt wird, so können diese sicher befahren werden.

Ergebnis Alle Tests endeten mit PASS. Wie erwartet, wurden alle Konfliktrouten für jede Route korrekt aufgezählt. Die Interlocking-Table ist dadurch auch in Bezug zu nicht in Konflikt stehenden Routen sicher.

4.3.3 Ergebnis der Tests

Die Tests sollten zeigen, dass das Modell des Sub-Controllers, sowie das Modell des Gesamtsystems sicher ist. Jedoch wurde im Sub-Controller-Modell, sowie in der Interlocking-Table noch gravierende Sicherheitsverletzungen entdeckt.
Im Sub-Controller gab es den Fehler, dass es möglich war eine Weiche zweimal von Sub-Controllern zu stellen, deren Routen im Konflikt waren. Das führe dazu, dass Züge im laufenden System wohl möglich entgleisen oder gar kollidieren. Gerade diese Situation soll vom Stellwerk verhindert werden. Aufgrund des Testergebnisses und des Debuggers in FDR4 konnte die Fehlerquelle schnell ausfindig gemacht werden und konnte im Nachhinein im Modell, sowie im implementierten System verbessert werden.
Weiter gab es in der Interlocking-Table einige Fehler, die das falsche Stellen der Weichen verursachte. Die Folge dieser falschen Weichenstellung wäre, entweder eine Entgleisung des darauf fahrenden Zuges, oder ein falsches Abbiegen, was zu einer Kollision mit einem auf dem anderen Gleis befindlichen Zug führt.
Die erkannten Fehler wurden verbessert und das Modell aktualisiert. Somit ist nun das sichere Befahren des Gleisnetzes garantiert. Ebenso gelten, die von mir getesteten Komponenten als sicher.

Kapitel 5

Evaluation

5.1 Testkonfiguration und Ergebnisse

Durch das Testen des Gleisnetzes, der Sub-Controller und des Gesamtsystems konnten einige Fehler im System ausfindig gemacht werden. Gerade schwerwiegende Sicherheitslücken wurden dabei im Sub-Controller, sowie in der Interlocking-Table gefunden, die eine Gefahr für Mensch und Maschine darstellen können.
Zunächst wurde das Gleisnetz auf Äquivalenz mit dem realen Gleisnetz getestet. Für diesen Test reicht es aus, wenn ein Zug jede Weiche und jedes Track-Element einmal befährt. Damit wird schließlich gezeigt, dass die Weichen und Track-Elemente korrekte synchronisiert werden, sodass dieses gleich mit dem realen Gleisnetz ist. Das kann wird damit begründet, dass ein Zug zunächst die erste Weiche befährt und dann durch Stellen der Folgeweiche auf diese gelangt. Dabei wird die Folgeweiche so gestellt, wie sie in dem realen Gleisnetz auch gestellt werden würde. Wird kein crash- oder derailing-Event erzeugt, so entspricht es dem realen. Um zu testen, dass ein Zug eine Weiche tatsächlich befährt, kann ein Fehler eingebaut werden (Weiche falsch herum), wodurch dann ein Fehler ausgelöst wird (Kapitel 4.2).
Dadurch wurde ein Fehler bei der Implementierung des Gleisnetzes gefunden und behoben.
Um die Sub-Controller zu testen, genügt es, zwei Sub-Controller parallel zu testen. Jeder Sub-Controller implementiert denselben Zustandsautomaten, arbeitet jedoch mit unterschiedlichen Routendaten. Für den Test wurden zwei Sub-Controller mit unterschiedlichen Routen benötigt, die in Konflikt stehen. Der Grund dafür ist, dass diese Sub-Controller zum Teil gleiche Weichen stellen. Somit sollte allgemein verifiziert werden, dass die jeweiligen Weichen zur selben Zeit nur einmal von einem Sub-Controller gestellt werden. Anschließend soll der andere Sub-Controller in den Konfliktzustand versetzt werden. Aus dem Grund, dass alle Sub-Controller denselben Zustandsautomaten implementieren, ist dieser Test ausreichend für alle weiteren Konfigurationen von Sub-Controller Verbindungen (siehe Kapitel 4.3.1).

Die Tests für das Gesamtsystem decken die in dem System vorkommenden Fälle ab. Das wird durch die Konfliktrouten realisiert. Sobald zwei oder mehrere Routen in Konflikt stehen, darf nur eine Route, aufgrund des wechselseitigen Ausschlusses, diese sperren. Ist es der Fall, dass vier Züge das Gleisnetz befahren und zwei Züge, aufgrund ihrer angeforderten Route, in Konflikt stehen, befahren die nicht in Konflikt stehenden Routen das Gleisnetz ohne Einschränkungen. Von den in Konflikt stehenden Routen darf dabei nur einer Fahren. Dadurch ist es egal, welche Nicht-Konfliktrouten dabei zusätzlich zu Konfliktrouten befahren werden. Gerade diese stellen dadurch, dass diese nicht zu den anderen Routen in Konflikt stehen, keine unmittelbare Gefahr dar.
Somit ist es lediglich wichtig, wie sich Konfliktrouten untereinander verhalten.
Es ist notwendig, dass jede Route gleichzeitig mit jeder anderen befahren wird. Beim Testen der Konfliktrouten, sollte gezeigt werden, dass in Konflikt stehende Routen, in paralleler Ausführung, keine Sicherheitsrisiken darstellen. Es darf nur eine Route zur selben Zeit gesperrt und zum Befahren freigegeben werden.
Ebenfalls ist es notwendig zu prüfen, ob Routen, die nicht in Konflikt zu einer anderen Route stehen, ohne Sicherheitsrisiken parallel befahren werden können. Falls es dann zu einem Sicherheitsrisiko kommt, gibt es eine zusätzliche Konfliktroute, die nicht aufgezählt wurde.
Aus dem Grund, dass Nicht-Konfliktrouten, die zur Zeit befahren werden, keinen Einfluss auf zwei in Konflikt stehende Routen, die ebenfalls zur Zeit befahren werden, haben, ist die Testkonfiguration vollständig und ausreichend. Es konnten so alle Fehler erkannt werden und die Sicherheit nachgewiesen werden.

5.2 Genutzte Rechnerressourcen

Insgesamt entstanden ca. 3230 Testfälle, um das Gleisnetz (Testfälle: 6), sowie die Sub-Controller (Testfälle: 1), das Gesamtsystem in Bezug zu Konfliktrouten
(Testfälle: 3118) und das Gesamtsystem in Bezug zu korrekte Weichenstellung
(Testfälle: 96) mit einer annehmbaren Teststärke zu testen. Dabei verursachten das vollständig in CSP_M implementierte Gleisnetz, sowie das Stellwerk mit dem Traincontroller, den Subcontrollern und Zügen eine hohe Ressourcenauslastung. Kleine Tests, wie der Subcontroller Test und die Model-in-the-Loop Tests waren problemlos auf einem Notebook mit einem Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz mit 4 Kernen und 8 HT Kernen und 16GB DDR4 Arbeitsspeicher möglich.
Für die Tests des Gesamtsystems reichte dieser nicht mehr aus, wodurch ein Server von Google über die Google Cloud Platform (GCP) [Goo19] für eine kurze Zeit gemietet werden musste. Darüber war es möglich, Server stündlich zu mieten, ohne Mindestvertragslaufzeit. Der gemietete Server besaß einen Intel(R) Xeon(R) Gold 6242 CPU @ 2.8 GHz mit 16 Kernen und 128GB DDR4 Arbeitsspeicher. Die implementierten Testfälle wur-

de von FDR4 immer sequentiell ausgeführt. FDR4 verteilte dabei die CSP_M-Prozesse eigenständig auf die jeweiligen CPU-Kerne. Wie die Prozesse auf die jeweiligen Kerne aufgeteilt werden, ist in Kapitel 3.2.2 beschrieben.
Dabei war auf dem Notebook das Betriebssystem Linux xUbuntu 18.04.2 LTS mit dem Linux Kernel 5.0.0 und auf dem Server Linux CentOS 7 mit dem Linux Kernel 3.10.0. Von FDR4 wurde immer die aktuell neuste Version FDR 4.2.4 genutzt.
Die Konfliktrouten-Tests wurden dabei unterteilt, wobei Tests für immer jeweils 25 Routen zusammen in eine CSP-Datei gefasst wurden. Daher gibt es acht Testdateien (vier für Konfliktrouten-Tests und vier für Nicht-Konfliktrouten-Tests).

Dauer und Ressourcen der Test:

Test	Dauer(gesamt)	RAM-Verbrauch	Rechner
Model-in-the-Loop	$<$ 1 Sekunden	512MB	Notebook
Subcontroller	$<$ 10 Sekunden	285MB	Notebook
Gesamtsystem: Routen	26 Sekunden	∅ 285MB	Notebook
Gesamtsystem: Konflikte 1	ca. 17 Minuten	86,8 GB	Server
Gesamtsystem: Konflikte 2	ca. 10 Minuten	62,3 GB	Server
Gesamtsystem: Konflikte 3	ca. 16 Minuten	102,7 GB	Server
Gesamtsystem: Konflikte 4	ca. 9 Minuten	53,8 GB	Server
Gesamtsystem: Nicht-Konflikte 1	ca. 4 Minuten	12,7 GB	Server
Gesamtsystem: Nicht-Konflikte 2	ca. 5 Minuten	16 GB	Server
Gesamtsystem: Nicht-Konflikte 3	ca. 3 Minuten	9,8 GB	Server
Gesamtsystem: Nicht-Konflikte 4	ca. 3,5 Minuten	11,1 GB	Server

Tabelle 1: Benchmarkergebnisse

Dabei konnten alle Tests ausgeführt werden und erzielten ein Ergebnis. Alle Tests waren am Ende erfolgreich.

Kapitel 6

Fazit und Ausblick

6.1 Vergleich FDR4 und nuXmv

Parallel zu dieser Bachelor-Thesis wurde von Matthias Lange eine Arbeit mit ähnlichem Thema geschrieben. Beide Arbeiten thematisieren das Model Checking eines autonomen Bahnsystems. Dabei wurde in dieser Arbeit das Tool FDR4 der Oxford University genutzt. [TGR13] Matthias Lange hingegen nutzte das aus nuSMV hervorgegangene Tool nuXmv von Fondazione Bruno Kessler. [Kes14] Beide Tools unterscheiden sich fundamental in ihre Art und Weise, wie es das Modell gegen die einzuhaltenden Spezifikationen testet, mit dem Ziel möglichst vollständig zu testen. Jedoch gab es dabei unterschiedliche Herangehensweisen und fast identische Ergebnisse. Im Folgenden werden beide Model-Checker kurz erläutert.

FDR4 FDR4 ist kein klassischer Model-Checker, sondern ein sogenannter Refinement-Checker. Es wird, im Vergleich zum konventionellen Model Checking, nicht global geprüft, ob eine LTL oder CTL-Formel eine bestimmte Spezifikation erfüllt, sondern, ob eine bestimmte Folge von Signalen (Trace) in einem Prozess vorhanden sind. FDR4 nutzt dafür die Prozessalgebra Communicating Sequential Processes (kurz: CSP/CSP_M), die erstmals 1985 von C.A.R. Hoare vorgestellt wurde. [Hoa85] Diese legt den Schwerpunkt auf die Kommunikation zwischen Prozessen, mit der, über sogenannte Channels (Kanäle) Daten und Signale versendet und empfangen werden können. CSP_M verwendet im Vergleich zum klassischen CSP Elemente und Sprachstile einer funktionalen Programmiersprache, wie Pattern-Matching.
CSP-Prozesse können sich dabei auch auf bestimmte Signale von Kanälen synchronisieren. CSP_M beschreibt dabei das Verhalten des Systems, die durch Kanäle kommunizieren. Ein einfaches Beispiel ist der Prozess $P = c \rightarrow STOP$, der zunächst das Event c erzeugt und sich dann beendet.
FDR4 prüft, ob ein Prozess einen anderen verfeinert in Bezug zu Trace-, Fehler- oder

Fehlerabweichungs-Verfeinerung. Zudem ist FDR4 in der Lage auf Deadlocks zu prüfen. Ein Trace ist dabei eine Folge von Signalen, die erzeugt werden. Nach einem Trace kann auch direkt ein Zustandswechsel erfolgen, indem ein neuer Prozess am Ende des Trace aufgerufen wird. Mit dem Muster ist es möglich so Zustandsautomaten zu modellieren:

```
channel a,b,c

Q = b -> c -> STOP

P = a -> b -> P'
P' = c -> STOP

assert Q [T= P \{|a|}
```

Listing 27: CSP_M Beispiel

Mit Listing 6.1 soll nun getestet werden, ob der Prozess P den Trace $b \rightarrow c$ implementiert (ausgedrückt durch Q). So beschreibt Zeile 8 im Beispiel: Behaupte, dass der Prozess Q vom Prozess P ohne das Erzeugen aller Events vom Kanal a Trace-verfeinert wird. Ist diese Behauptung richtig, so schließt der Test mit PASSED ab, findet FDR4 ein Gegenbeispiel, so nennt FDR4 das fehlerhafte Event und schließt mit FAILED ab. [TGR13]

nuXmv Bei nuXmv handelt es sich um einen Symbolic Model-Checker, der aus dem Model-Checker nuSMV hervorgegagen ist. Bei Symbolic Model-Checkern wird der Zustandsraum durch boolsche Funktionen, im Falle von nuXmv mithilfe von Binary Decision Diagrams(BDD's) vereinfacht, wodurch größere Modelle Verifiziert werden können. Dabei werden Modelle durch Zustände und Zustandsübergänge dargestellt. Der Zustandsraum wird mithilfe von Variablen und Modulen erstellt und anschließend die Zustandsübergänge durch Transitionsbedingungen definiert. Die next-Anweisung weist dabei einer Variablen einen Wert zu, den sie im nächsten Zustand haben soll, während die init-Anweisung den Initialen Wert einer Variablen bestimmt. Anschließend können mithilfe von Spezifikationen durch Temporale Logiken wie der Linear Temporal Logic(LTL) oder der Computation Tree Logic(CTL) bestimmte Eigenschaften für ein Modell nachgewiesen werden. [Kes14]
Listing 6.1 zeigt dabei ein Beispiel Programm, bei dem eine Variable existiert, die Werte zwischen 0 und 15 annehmen kann und Initial auf 0 steht. Durch Transitionsbedingungen wird dieser Variable im nächsten Zustand der Wert 0 zugewiesen, wenn sie vorher den Wert 7 hatte und in allen anderen Fällen wird sie im nächsten Zustand um 1 erhöht und Modulo 16 gerechnet.

```
MODULE main

VAR
y : 0..15;

ASSIGN
init(y) := 0;

TRANS
case
y = 7   : next(y) = 0;
TRUE    : next(y) = ((y+1) mod 16);
esac

LTLSPEC G ( y=4 -> X y=6 )
```

Listing 28: nuXmv Beispiel

Die LTL Spezifikation überprüft dann, ob im nächsten Zustand immer y = 6 ist, wenn vorher y = 4 war. Das ist in diesem Modell offensichtlich nicht der Fall und nuXmv würde ein Gegenbeispiel dafür generieren.

Ergebnisse Durch beide Arbeiten wurde nachgewiesen, dass in der Interlocking-Table noch Fehler existierten. So stellten die Sub-Controller der Routen 38, 39,40 und 78 einige Weichen für den darauf fahrenden Zug falsch. Diese Situation hätte entweder in einer Entgleisungssituation enden können, oder, wenn auf der anderen Seite der Weiche sich ein Zug befindet, zu einem Zusammenstoß der Züge kommen können. Diese Fehler wurden verbessert, sodass diese Sicherheitsverletzungen nicht mehr auftreten können.
Einen Fehler, der nur mit FDR4 gefunden wurde, ist, dass zwei im Konflikt stehende Routen Signale zum Setzen der jeweils anderen Route im Konflikt verpassen können. Das kann wieder eine Entgleisung der Züge oder sogar einen Zusammenstoß dieser auslösen. (siehe Kapitel 4.3.1)
Im Bezug zur Laufzeit kann abschließend erwähnt werden, dass FDR4 beim Model Checking deutlich performanter war, als nuXmv. Das liegt an der Tatsache, dass FDR4 einerseits auf Multicore Systemen ausgelegt ist, andererseits auch nur auf Verfeinerung testet, sodass es nicht so schnell zu einer Zustandsexplosion kommen kann. NuXmv hat jedoch den Nachteil, dass dieser symbolische Model-Checker nur einen CPU-Kern nutzen kann und zudem, durch die Durchführung der Modellprüfung, schnell in eine Zustandsexplosion geraten kann.
Jedoch ist abschließend zu sagen, dass, sowohl FDR4, als auch nuXmv Fehler in der Stellwerkslogik erkannt haben. Dadurch konnte die Sicherheit des Systems gesteigert, beziehungsweise dargelegt werden.

6.2 Ausblick

Durch das Model-Checken des Stellwerkes konnten noch einige bestehende Fehler im System ausfindig gemacht werden. Gerade tiefer liegende Fehler im System konnte da-

durch entdeckt werden. Jedoch konnte auch gezeigt werden, dass das System in Bezug zu den Spezifikationen sicher ist.
Die Verifikation mit FDR4 und CSP_M forderte zunächst eine lange Einarbeitungszeit, stellte sich bei der Modellierung als sehr simpel heraus. Jedoch ist das Modellieren des Gleisnetzes sehr mühsam und fordert viel Handarbeit. Doch die Einfachheit und "lazyness" der Sprache macht es einfach, Zustandsautomaten schnell zu modellieren. Der Umfang des Codes vom implementierten Modell ist dabei überschaubar und umfasst weniger als 10.000 Zeilen CSP-Code.
In Sachen Ausführungszeit liegt FDR4 klar vor nuXmv. FDR4 nutzt zum Suchen von Gegenbeispielen von Verifikations-Behauptungen die Breitensuche im linearen Transitionsgpraphen (siehe Kapitel 3.2). Dadurch, dass FDR4 zudem skalierbar ist, kann jeder CPU-Kern zum Refinement Check genutzt werden. Dies kann man weiter ausreizen, indem man ganze Server zu Cluster mit MPI erstellt und dadurch die Performanz weiter steigern kann. Das Tool MPI (Message Passing Interface) wird dabei offiziell von FDR4 unterstützt. Dadurch sind in Sachen Ressourcen keine Grenzen gesetzt.
Um den Geschwindigkeitsvorteil von FDR4 weiter auszureizen, wäre es sinnvoll Modelle automatisch zu generieren. Dabei könnte man beispielsweise die Interlocking-Table direkt aus einer CSV-Datei zu entsprechenden CSP-Code generieren. Ebenfalls könnte man das Gleisnetz automatisch generieren lassen.

Literaturverzeichnis

[Dij20] Wikipedia/Edsger W. Dijkstra. Dijkstra-algorithmus. Wikipedia, January 2020.

[FB19] Matthias Lange Thomas Lipps Tom Niewöhner Jan R. Kropp Lars Forquignon Jan Leuschner Felix Kohlhase Nikas Kandsorra Felix Brüning, Raven Hölting. Projektbericht bachelorprojekt teamod. techreport, Universität Bremen, AG Betriebssysteme-Verteilte Systeme, 2019.

[Goo19] Google. Google cloud platform. website, December 2019. `https://cloud.google.com/`, November 2019.

[Gre19] Dr. Gebhard Greiter. Zur total cost of ownership von software, August 2019. `http://greiterweb.de/spw/SoftwareTCO.htm`.

[Gro17] Object Management Group. Omg system modeling language 1.5, 05 2017. `https://www.omg.org/spec/SysML/1.5/PDF`.

[HHP17] Linh Vu Hong, Anne Elisabeth Haxthausen, and Jan Peleska. Formal modelling and verification of interlocking systems featuring sequential release. *Sci. Comput. Program.*, 133:91–115, 2017.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985. `https://dblp.org/rec/bib/books/ph/Hoare85`, Juli 2019.

[HP15] Anne Elisabeth Haxthausen and Jan Peleska. Model checking and model-based testing in the railway domain. In *Formal Modeling and Verification of Cyber-Physical Systems, 1st International Summer School on Methods and Tools for the Design of Digital Systems, Bremen, Germany, September 2015*, pages 82–121, 2015.

[HWK+17] Norman Hansen, Norbert Wiechowski, Alexander Kugler, Stefan Kowalewski, Thomas Rambow, and Rainer Busch. Model-in-the-loop and software-in-the-loop testing of closed-loop automotive software with arttest. In Maximilian Eibl and Martin Gaedke, editors, *INFORMATIK 2017*, pages 1537–1549. Gesellschaft für Informatik, Bonn, 2017.

[Kes14] Fondazione Bruno Kessler. The nuxmv model-checker, January 2014. https://nuxmv.fbk.eu/.

[Kne18] Ralf Kneuper. Die geschichtliche entwicklung des v-modells, 11 2018.

[OU19] UK Oxford University. Overview over fdr4, 07 2019. https://cocotec.io/fdr/manual/index.html, Juli 2019.

[PHH16] Jan Peleska, Wen-ling Huang, and Felix Hübner. A novel approach to HW/SW integration testing of route-based interlocking system controllers. In *Reliability, Safety, and Security of Railway Systems. Modelling, Analysis, Verification, and Certification - First International Conference, RSSRail 2016, Paris, France, June 28-30, 2016, Proceedings*, pages 32–49, 2016.

[TGR13] Alexandre Boulgakov A.W. Roscoe Thomas Gibson-Robinson, Philip Armstrong. *Failures Divergences Refinement (FDR) Version 4*, 2013. https://cocotec.io/fdr/manual/implementation/refinement_checking.html, Dezember 2019.

[TGR19a] Alexandre Boulgakov A.W. Roscoe Thomas Gibson-Robinson, Philip Armstrong. *CSPM Manual.* University of Oxford, September 2019. https://www.cs.ox.ac.uk/projects/fdr/manual/cspm.html, September 2019.

[TGR19b] Alexandre Boulgakov A.W. Roscoe Thomas Gibson-Robinson, Philip Armstrong. *CSPM Refinement-Checking.* University of Oxford, September 2019. https://cocotec.io/fdr/manual/implementation/refinement_checking.html#refinement-checking, Dezember 2019.

Anhang

Implementierung einer Weiche

```
1      ----------------------------------------
2      -- CSP-Implementation of a Point-Section
3      ----------------------------------------
4
5  POINT(id, s0, s1, s2) =
6      ([] x : {s0, s1, s2} @ rail.x -> crash.id -> STOP)
7      []
8      reqsec.id -> MARKED_P(id, s0, s1, s2)
9
10 MARKED_P(id, s0, s1, s2) =
11     set.id.str -> P_LOCKED_STR(id, s0, s1, s2)
12     []
13     set.id.crss -> P_LOCKED_CRSS(id, s0, s1, s2)
14     []
15     ([] x : {s0, s1, s2} @ rail.x -> crash.id -> STOP)
16     []
17     reqsec.id -> derailing.id -> STOP
18
19 P_LOCKED_STR(id, s0, s1, s2) =
20     rail.s2 -> P_LOCKED_STR_OCC_FROM_RIGHT(id, s0, s1, s2)
21     []
22     rail.s0 -> P_LOCKED_STR_OCC_FROM_LEFT(id, s0, s1, s2)
23     []
24     rail.s1 -> derailing.id -> STOP
25     []
26     release.id -> POINT(id, s0, s1, s2)
27     []
28     reqsec.id -> derailing.id -> STOP
29     []
30     set.id.str -> derailing.id -> STOP
31     []
32     set.id.crss -> derailing.id -> STOP
33
34 P_LOCKED_STR_OCC_FROM_RIGHT(id, s0, s1, s2) =
35     rail.s0 -> P_LOCKED_STR(id, s0, s1, s2)
36     []
37     ([] x : {s1, s2} @ rail.x -> crash.id -> STOP)
38     []
39     reqsec.id -> derailing.id -> STOP
40     []
41     set.id.str -> derailing.id -> STOP
42     []
43     set.id.crss -> derailing.id -> STOP
44
45 P_LOCKED_STR_OCC_FROM_LEFT(id, s0, s1, s2) =
46     rail.s2 -> P_LOCKED_STR(id, s0, s1, s2)
47     []
48     ([] x : {s0, s1} @ rail.x -> crash.id -> STOP)
49     []
50     reqsec.id -> derailing.id -> STOP
51     []
52     set.id.str -> derailing.id -> STOP
53     []
54     set.id.crss -> derailing.id -> STOP
55
56 P_LOCKED_CRSS(id, s0, s1, s2) =
57     rail.s0 -> P_LOCKED_CRSS_OCC_FROM_LEFT(id, s0, s1, s2)
58     []
59     rail.s1 -> P_LOCKED_CRSS_OCC_FROM_LEFT2(id, s0, s1, s2)
60     []
```

```
61      rail.s2 -> derailing.id -> STOP
62      []
63      release.id -> POINT(id, s0, s1, s2)
64      []
65      reqsec.id -> derailing.id -> STOP
66      []
67      set.id.str -> derailing.id -> STOP
68      []
69      set.id.crss -> derailing.id -> STOP
70
71
72  P_LOCKED_CRSS_OCC_FROM_LEFT(id, s0, s1, s2) =
73      rail.s1 -> P_LOCKED_CRSS(id, s0, s1, s2)
74      []
75      ([] x : {s0, s2} @ rail.x -> crash.id -> STOP)
76      []
77      reqsec.id -> derailing.id -> STOP
78      []
79      set.id.str -> derailing.id -> STOP
80      []
81      set.id.crss -> derailing.id -> STOP
82
83  P_LOCKED_CRSS_OCC_FROM_LEFT2(id, s0, s1, s2) =
84      rail.s0 -> P_LOCKED_CRSS(id, s0, s1, s2)
85      []
86      ([] x :{s1, s2} @ rail.x -> crash.id -> STOP)
87      []
88      reqsec.id -> derailing.id -> STOP
89      []
90      set.id.str -> derailing.id -> STOP
91      []
92      set.id.crss -> derailing.id -> STOP
```

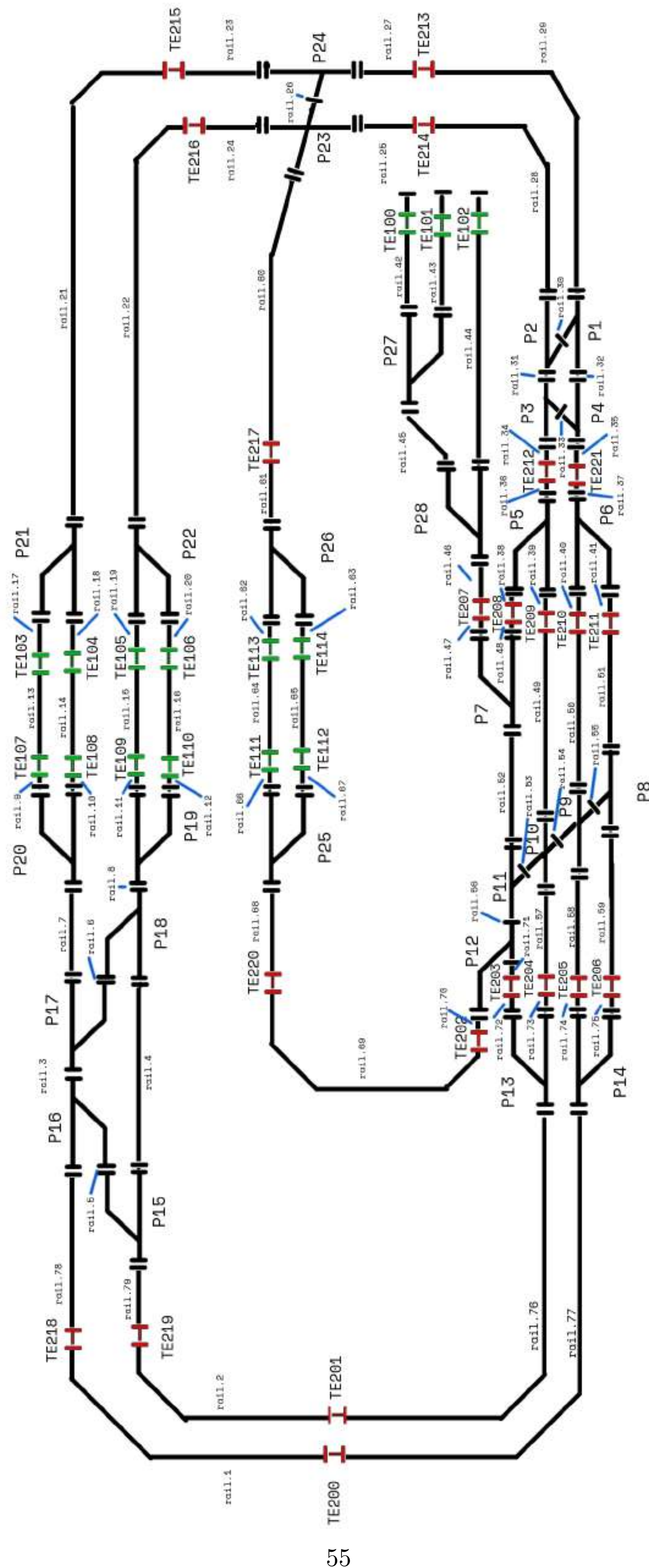
Listing 29: Weichen Implementierung

Implementierung eines Track-Elements

```
1       -----------------------------------------
2       -- CSP-Implementation of a Track-Element-Section
3       -----------------------------------------
4   TRACK_ELEMENT(id, s0, s1) =
5       rail.s0 -> occupied.id -> TRAIN_FROM_LEFT(id, s0, s1)
6       []
7       rail.s1 -> occupied.id -> TRAIN_FROM_RIGHT(id, s0, s1)
8       []
9       free.id -> TRACK_ELEMENT(id, s0, s1)
10
11  TRAIN_FROM_LEFT(id, s0, s1) =
12      rail.s1 -> TRACK_ELEMENT(id, s0, s1)   -- drive through
13      []
14      rail.s0 -> crash.id -> STOP
15      []
16      release.id -> free.id -> TRACK_ELEMENT(id, s0, s1)
17
18  TRAIN_FROM_RIGHT(id, s0, s1) =
19      rail.s0 -> TRACK_ELEMENT(id, s0, s1)   -- drive through
20      []
21      rail.s1 -> crash.id -> STOP
22      []
23      release.id -> free.id -> TRACK_ELEMENT(id, s0, s1)
```

Listing 30: Track-Element Implementierung

Gleisnetz mit Track-Elementen und Rails



Inhalt des Speichermediums

```
.
├── BeispielCode
│   ├── example.csp
│   ├── graph.dot
│   ├── graph.png
│   ├── proc_ex.csp
│   └── refinement.csp
├── BenchmarkTests
│   └── ergebnisse_benchmark.txt
├── MiL-Tests
│   ├── MiL-Tests.csp
│   ├── small_network.csp
│   ├── Testergebnis_gesamtes_gleisnetz.txt
│   └── Testergebnis_kleines_gleisnetz.txt
├── Models
│   ├── channel.csp
│   ├── controller.csp
│   ├── crossing.csp
│   ├── mutex.csp
│   ├── point.csp
│   ├── railway_network.csp
│   ├── track_elem.csp
│   └── trains.csp
├── ReadMe.md
├── RouteTests
│   ├── route_tests.csp
│   └── Testergebnis.txt
├── TestKonfliktRouten
│   ├── KonfliktTest
│   │   ├── ctrl_sync.csp
│   │   ├── positiveSystemTests_0_0.csp
│   │   ├── positiveSystemTests_0_1.csp
│   │   ├── positiveSystemTests_1_0.csp
│   │   ├── positiveSystemTests_1_1.csp
│   │   └── pos_test_cases.csp
│   ├── NichtKonfliktTest
│   │   ├── negativeSystemTests_0_0.csp
│   │   ├── negativeSystemTests_0_1.csp
│   │   ├── negativeSystemTests_1_0.csp
│   │   ├── negativeSystemTests_1_1.csp
│   │   └── path_ctrl.csp
│   └── Testergebnisse
│       ├── negative00.txt
│       ├── negative01.txt
│       ├── negative10.txt
│       ├── negative11.txt
│       ├── positive00.txt
│       ├── positive01.txt
│       ├── positive10.txt
│       └── positive11.txt
├── TestSubController
│   ├── ctrl_new.csp
│   ├── ctrl_old.csp
│   ├── sub_ctrl_test.csp
│   ├── Testergebnis_new.txt
│   └── Testergebnis_old.txt
└── Thesis
    └── thesis.pdf

11 directories, 46 files
```

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit ohne Hilfe Dritter und ohne Benutzung Anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form in keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Ort, Datum *Unterschrift*