Universität Bremen

Fachbereich 3 - Mathematik und Informatik

Bachelorarbeit

im Studiengang Informatik

zur Erlangung des akademischen Grades
Bachelor of Science

Thema:	IXL Modelchecking mit nuXmv
Autor:	Matthias Lange <langemat@uni-bremen.de> MatNr. 4332419
Version vom:	29. Dezember 2019
1. Prüfer:	Prof. Dr. Jan Peleska
2. Prüfer:	Prof. Dr. Rolf Drechsler

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listingverzeichnis

1. Einführung

1.1. Motivation

Im Bachelorprojekt „TEAMOD“ haben wir eine Zuganwendung programmiert, die Züge automatisiert über ein vordefiniertes Gleisnetz fahren lässt, ohne dass diese einen Unfall verursachen. Die Anwendung wurde mehrfach manuell getestet und auch in einigen Vorführungen bereits vorgestellt. Auch Unit-Tests mit einer Branch-Coverage von 100 % wurden für die Anwendung geschrieben und bestehen alle. Allerdings fangen diese Tests nur grobes Fehlverhalten ab und geben keine endgültige Sicherheit, dass das System keine Fehler enthält, die unter Umständen doch zu einem Unfall führen können. Deswegen soll nun im Rahmen dieser Bachelorarbeit mithilfe des Model-Checkers „nuXmv“ ein Modell dieses Zugsystems entworfen werden und mithilfe von Verifikationsformeln die Unfallfreiheit der Anwendung nachgewiesen werden.

1.2. Zielsetzung

Ziel dieser Bachelorarbeit ist es, das im Bachelorprojekt „TEAMOD“ erstellte Stellwerk zu verifizieren und mögliche Fehlerquellen auszuschließen. Wenn sich dabei herausstellt, dass das Stellwerk keine Fehler enthält und in keinem Fall zwei Züge miteinander kollidieren können, beweist dies die fehlerfreie Funktionalität des Stellwerkes. Sollte sich jedoch herausstellen, dass das Stellwerk Fehler enthält, kann diese Erkenntnis für das folgende Masterprojekt genutzt werden, um Fehlerquellen zu analysieren und das Stellwerk zu optimieren.

1.3. Gliederung der Arbeit

Zu Beginn der Arbeit werde ich in Kapitel 2 das zugrundeliegende System erläutern, aus welchen Bestandteilen es besteht, welche Funktionalitäten es bietet und wie es aufgebaut ist.
Anschließend werden in Kapitel 3 die Grundlagen des Model-Checkers nuXmv erläutert und die wichtigsten Funktionen, welche auch in dieser Bachelorarbeit verwendet wurden, aufgezeigt.
Nachdem die Grundlagen erläutert wurden, wird in Kapitel 4 die Modellierung des Gleisnetzes und des Stellwerkes erklärt. Dabei werde ich den Aufbau meines Modells erklären und wie ich bestimmte Funktionalitäten des Systems im Modell umgesetzt habe. Zusätzlich werde ich auf ein Verfahren eingehen, welches zur Dekomposition des modellierten Gleisnetzes eingesetzt wird und damit zu einer Verringerung des Zustandsraumes für nuXmv führt.
Kapitel 5 beschreibt Model-in-the-Loop Tests, welche ich vorgenommen habe, um das korrekte Verhalten meiner Modellierung nachzuweisen.

In Kapitel 6 werde ich dann meine erzielten Ergebnisse evaluieren und aufzeigen, welche sicherheitskritischen Eigenschaften vom Stellwerk erfüllt werden und welche nicht. Weiterhin werde ich in Kapitel 7 einen Vergleich zu dem in der parallel geschriebenen Arbeit von Felix Brüning verwendeten Model-Checker FDR4 ziehen.
Zum Schluss werde ich dann in Kapitel 8 einen Ausblick geben, wie die Ergebnisse dieser Arbeit weiter genutzt werden können.

2. Das zugrundeliegende System

Diese Bachelorarbeit basiert auf einer Anwendung für Züge, die ohne Unfall gleichzeitig auf einem Gleisnetz fahren sollen. Die Anwendung wurde im Bachelorprojekt „TEAMOD“ entwickelt. [BLH+19]

2.1. Das Gleisnetz

Das Gleisnetz besteht aus mehreren Komponenten eines Märklin Gleisnetzes, welches aus linearen Abschnitten, einfachen Weichen und Kreuzweichen besteht. Auf dem Gleisnetz können mehrere Züge fahren. In den Vorführungen, die bereits mit der Anwendung durchgeführt wurden, sind jeweils 3 Züge gefahren. Mithilfe einer Märklin Station lässt sich die Geschwindigkeit der Züge anpassen, die Weichenpositionen ändern und der Strom abschalten(Notstopp).

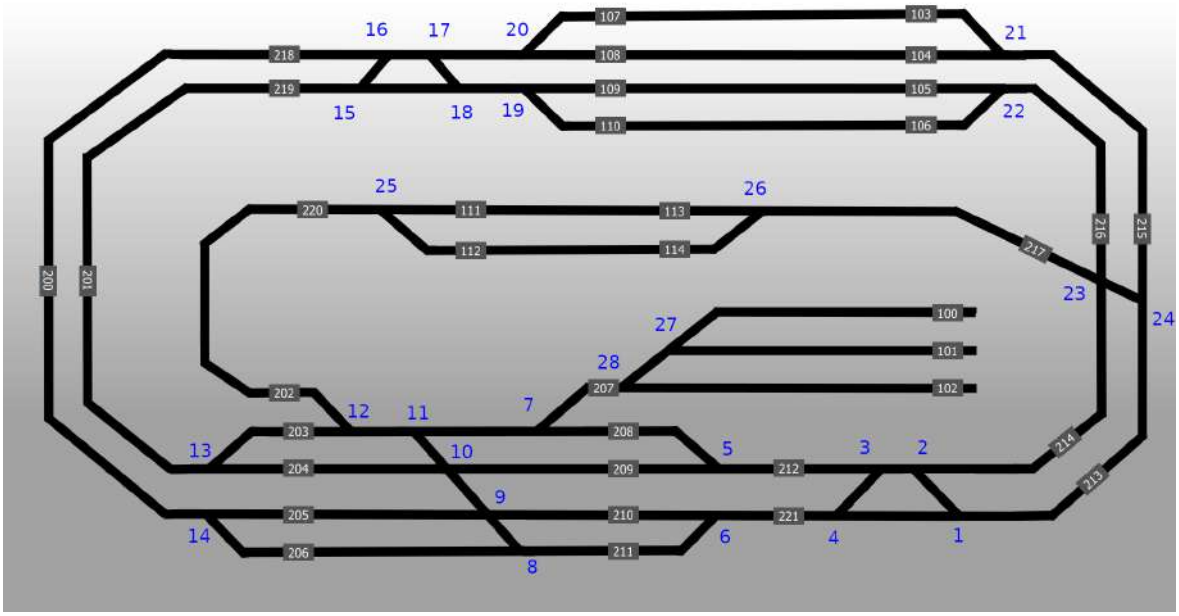

Abbildung 1: Gleisnetz aus dem Bachelorprojekt TEAMOD

2.2. Die 4 Hauptkomponenten

Im Bachelorprojekt haben wir uns in 4 Gruppen aufgeteilt, in denen jede Gruppe einen Teil der Implementierung bearbeitet hat. Die Aufgabenbereiche haben wir aufgeteilt in Positioning, Router/Communication, TCC und IXL.

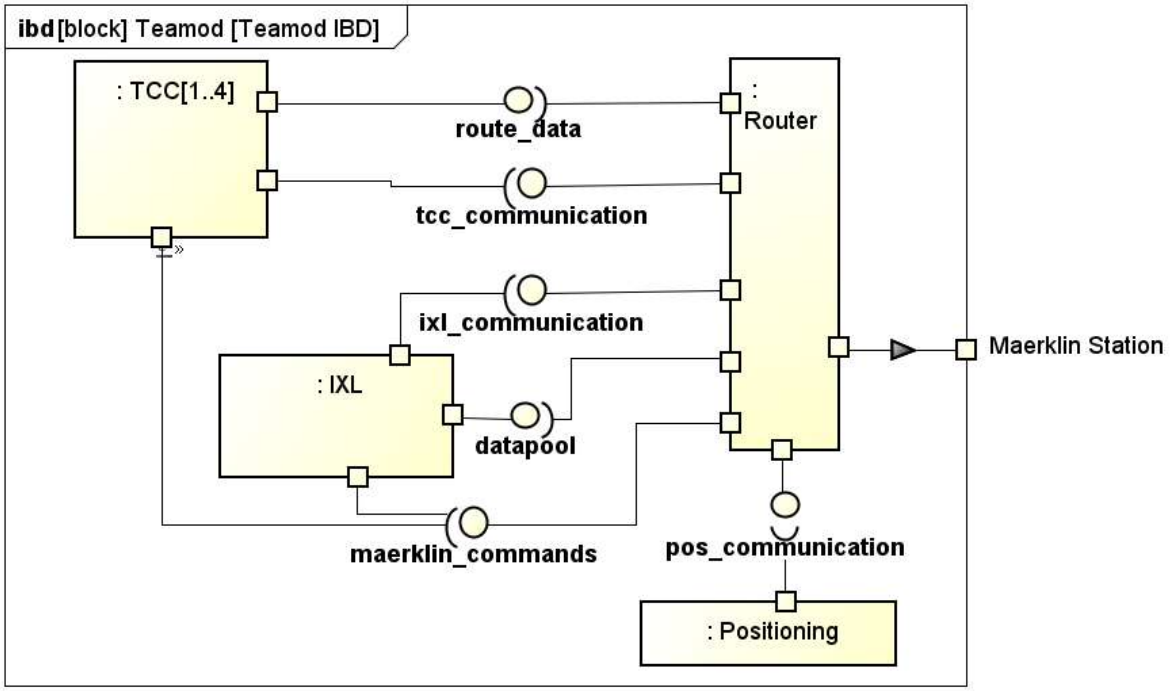

Abbildung 2: Teamod Module

2.2.1. Positioning

Das Positioning war zuständig für die Ortung der Züge. Sie mussten sich ein System überlegen, mit dem wir erkennen konnten, an welcher Position sich ein Zug zurzeit auf dem Gleisnetz befindet. Dazu wurden einzelne Gleiselemente abisoliert und die Verbindung zwischen den beiden Kontakten unter dem Gleis getrennt. Dadurch fließt auf diesen Gleiselementen nur auf einer Seite Strom. Sobald jedoch ein Zug über dieses Gleiselement fährt, wird die Verbindung über die Räder des Zuges, die über die Achse miteinander verbunden sind, wieder überbrückt und auf dem zweiten Kontakt fließt ebenfalls Strom. Um nun zu überprüfen, ob sich ein Zug auf diesem Gleiselement befindet, wurde ein Raspberry Pi an das abisolierte Gleiselement angeschlossen. Dieser misst, ob Strom fließt oder nicht. Wenn ein Zug über das Gleiselement fährt, ist der Stromkreis geschlossen und der Raspberry Pi sendet eine 1, ansonsten sendet er eine 0. Um Schwankungen abzufangen, die bei der Messung auftreten können und somit falsche Daten liefern, wurden Filteralgorithmen im Raspberry Pi eingebaut, die diese Schwankungen erkennen und falsche Daten verwerfen. Insgesamt wurden dadurch 37 Gleisnetze präpariert, die von 4 Raspberry Pi's abgehört werden. Die Raspberry Pi's senden alle 100ms per Broadcast ihre Daten. Die präparierten Gleise werden im weiteren Verlauf als Track Elemente (TE) bezeichnet. Diese sind außerdem in 2 Gruppen eingeteilt. Diejenigen mit der ID 1xx bezeichnen Streckenabschnitte, in denen ein kompletter Zug hineinfahren kann, ohne auf einer Weiche zu stehen. Für Track Elemente der ID 2xx muss diese Eigenschaft nicht gelten.

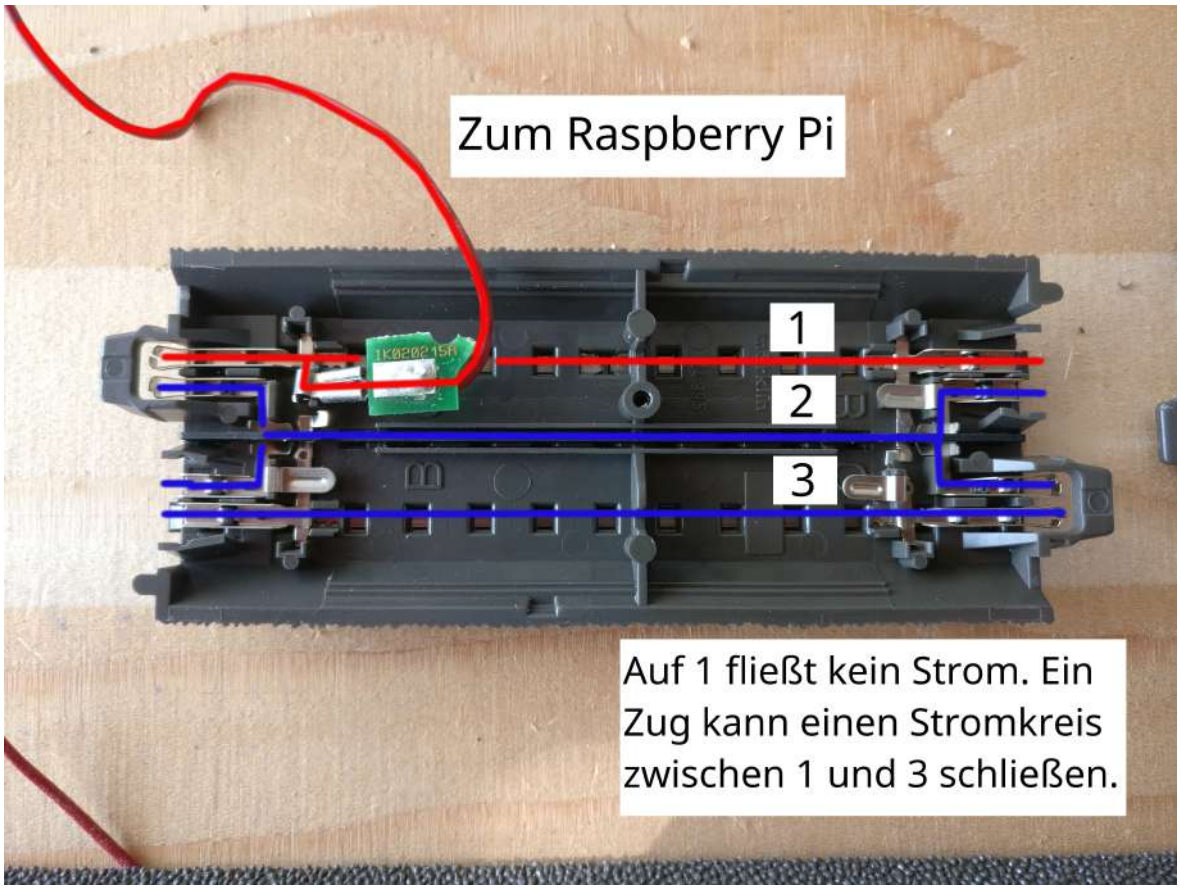

Abbildung 3: Präpariertes Gleis

2.2.2. Router

Die Gruppe Router hat sich mit der Kommunikation der Märklin Station und der einzelnen Module untereinander auseinandergesetzt. Hierfür wurde eine Bibliothek bereitgestellt, bei denen mithilfe von UDP-Paketen Kommandos oder Statusupdates verschickt wurden. Ausnahme bildet hierbei die Märklin Station, die ausschließlich über ein CAN-Protokoll kommunizieren kann. So werden Routeninformationen zwischen TCC und IXL, Zuggeschwindigkeiten, Weichenpositionen und Stoppsignale vom IXL und TCC an Märklin und die belegten Gleisabschnitte vom Positioning an IXL und TCC gesendet. Die Daten werden dabei in hoher Taktung versendet, sodass bei einem eventuellen Datenverlust sofort aktuelle Daten nachgeschickt werden. Außerdem sendet jedes Modul per Broadcast in regelmäßigen Abständen Lebenssignale. Wenn ein Modul von einem anderen Modul keine Lebenssignale bekommt, wird ein Stoppsignal an die Märklin Station vom entsprechenden Modul gesendet.

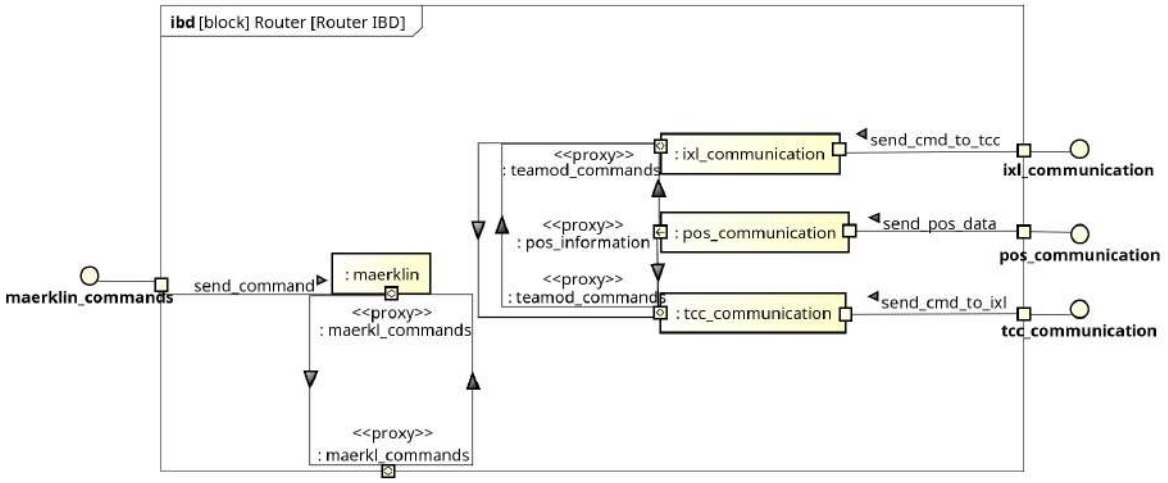

Abbildung 4: Router Kommunikation

2.2.3. TCC

Der TCC (Train Control Computer) ist die Anwendung, mit der die Züge gestartet werden und ihre Geschwindigkeit bestimmt wird. Dafür registrieren sich die Züge zunächst beim IXL. Wenn dies erfolgreich war, suchen sich die Züge ein beliebiges Ziel aus, zu dem sie fahren möchten und schicken das Ziel an das IXL. Anschließend wartet der TCC auf ein Signal vom IXL zum Starten. Wenn der TCC das Signal vom IXL bekommt, wird die Geschwindigkeit seines Zuges erhöht und er kann auf die nächste Route fahren. Gibt das IXL im Verlauf der Fahrt die Freigabe für die nächste Route, kann der Zug durchfahren, ansonsten bremst der Zug langsam ab und hält auf dem Ende der derzeitigen Route an. Dort wartet er wieder auf das Signal vom IXL. Kommt der Zug an sein Endziel an, hält er dort ebenfalls an und schickt ein neues Ziel an das IXL.

2.2.4. IXL

Das IXL (Interlocking System) ist das Kernelement der Anwendung und sorgt dafür, dass die Züge unfallfrei über das Gleisnetz fahren. Dafür hat das IXL intern in einem Datenpool eine Interlockingtable gespeichert, in der alle möglichen Routen, die ein Zug befahren kann, enthalten sind.

ID	SRC	DST	PATH	POINTS	LENGTH	MAX SPEED	CONFLICTS	DIRECTION
0	214	202	212,208,202	2:p,3:p,5:m,7:p,11:p,12:m	100	50	1,2,3,4,5,6,7,8,9,37,38,39,40,41,42,43,44,45,46,52,53,54,55,56,57,61,62,66,67,70,71,72,73,74,75,76,77,78,79,80,81,82,83	0
1	214	203	212,208,203	2:p,3:p,5:m,7:p,11:p,12:p	100	50	0,2,3,4,5,6,7,8,9,37,38,39,40,41,42,43,44,45,46,52,53,54,55,56,57,61,62,66,67,70,71,72,73,74,75,76,77,78,79,80,81,82,83	0

Abbildung 5: IXL Interlocking Tabelle

Züge auf dem Gleisnetz müssen sich dann beim IXL registrieren und anschließend ein Ziel versenden, zu dem sie fahren möchten. Der Train-Controller des IXL berechnet dann aus der Startposition des Zuges, welche sie vom Positioning bekommt, und dem

Ziel, welches sich der Zug ausgesucht hat, die kürzeste Strecke. Die Strecke besteht aus zum Teil mehreren Routen. Sollte sich der Zug dabei ein Ziel ausgesucht haben, dass von seiner derzeitigen Position aus nicht erreichbar ist, teilt das IXL dem Zug mit, dass er sich ein neues Ziel aussuchen muss. Erhält das IXL ein valides Ziel, überprüft es die erste Route, die der Zug befahren möchte und guckt, ob diese oder eine andere Route, die mit dieser Route in Konflikt steht, belegt ist. Ist das nicht der Fall, sperrt das IXL die Route und ihre Konflikte für weitere Züge und gibt dem anfragenden Zug das Signal, dass er die Route befahren kann. Andernfalls wartet das IXL darauf, dass die Route wieder frei wird. Sobald ein Zug die Route befährt, überwacht das IXL den Zug und gibt die Route wieder frei, sobald der Zug die Route wieder verlässt. Dafür wird für jede Route, die beantragt wird, ein eigener Sub-Controller gestartet, der den Status der Route überprüft und ggf. ändert.
Weiterhin überwacht das IXL im Safety-Monitor das gesamte Gleisnetz auf unvorhergesehene Ereignisse. Dafür verarbeitet es die Informationen vom Positioning und vergleicht diese mit den internen Informationen über gesperrte Routen. Sollte dabei ein Gleisabschnitt belegt sein, der außerhalb einer gesperrten Route liegt, wird sofort ein Notstopp ausgelöst. So wird verhindert, dass durch Geisterzüge oder falsch abbiegende Züge, z.B. durch defekte Weichen, ein Unfall verursacht wird. Außerdem schicken TCC und Positioning in regelmäßigen Abständen über Broadcast sogenannte Lebenssignale. Diese signalisieren, dass die Verbindung zwischen diesen beiden Komponenten noch besteht, und werden vom Health-Controller entgegengenommen. Bricht die Verbindung zwischen einem der beiden ab, z.B. weil die Verbindung zum Netzwerk nicht mehr gegeben ist oder ein Programm abstürzt, wird ebenfalls ein Notstopp ausgelöst.

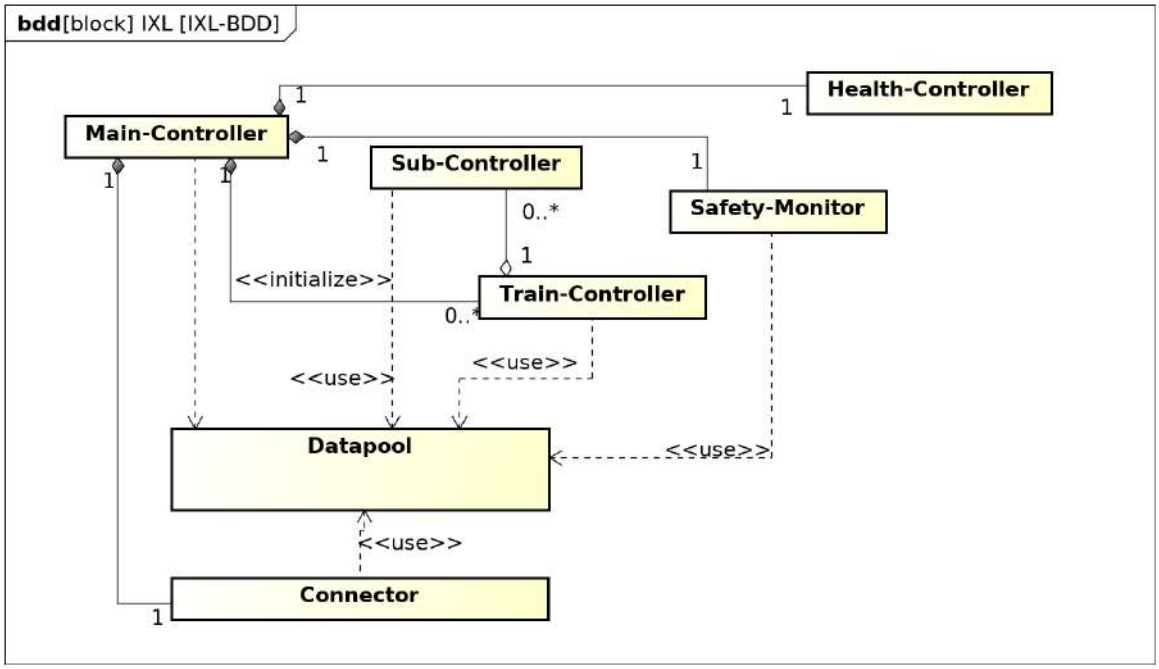

Abbildung 6: IXL Aufbau

3. nuXmv

nuXmv ist ein Symbolic Model Checker. Das bedeutet, dass er States aus dem Modell mithilfe von booleschen Funktionen optimiert und somit nicht jeden State explizit darstellen muss, wodurch größere Modelle verifiziert werden können. Des Weiteren bietet nuXmv viele Algorithmen für die Verifikation von sowohl endlichen als auch unendlichen Zustandsautomaten an. Im Folgenden werde ich die Syntax und Semantik von nuXmv näher erläutern, um anschließend die Funktionen und Verifikationsmethoden erklären zu können. Dabei gehe ich vornehmlich auf solche Elemente ein, die ich auch in meiner Bachelorarbeit verwendet habe. Für ausführliche Informationen sei hier auf die Dokumentation von nuXmv verwiesen. [BCC+16] [CCD+14]

3.1. Datentypen

nuXmv besitzt ein reiches Sortiment an verschiedenen Datentypen. Im Folgenden werde ich einen Überblick darüber geben, welche Datentypen von nuXmv unterstützt werden:

- Boolean: Boolean's sind Wahrheitswerte, die nur zwei Werte annehmen können, TRUE und FALSE.
- Enumeration: Enumerations sind Aufzählungen von Zuständen, die eine Variable annehmen kann. Es wird dabei zwischen Symbolic- und Integer Enumerations unterschieden. Symbolic Enumerations sind symbolische Werte, wie z.B. {stopped, running, waiting, finished}, wohingegen Integer Enumerations ausschließlich ganzzahlige Zahlenwerte enthalten: {2, 4, -2, 0}. Eine Enumeration kann entweder vom Typ Symbolic, Integer oder Symbolic und Integer, also einer Mischung aus beiden, sein.
- Word: Words sind Vektoren von Bits, auf denen bitweise Operationen ausgeführt werden können. Words können signed oder unsigned sein, also vorzeichenbehaftet oder nicht. Die Größe eines Words bestimmt die Anzahl an Bits in einem Vektor. Für die Darstellung eines Words als Integer wird die Bitrepräsentation verwendet.
- Integer: Ein Integer repräsentiert ganzzahlige Zahlenwerte im positiven oder negativen Bereich.
- Real: Reals sind Werte im Zahlenbereich der rationalen Zahlen.
- Array: Ein Array ist eine Liste von Elementen, die alle demselben Typen entsprechen, also z.B. Integer, Real, Boolean etc. Der kleinste und größte Index wird dabei selbst bestimmt und fängt nicht immer wie in Programmiersprachen bei 0 an. Auch ein Array von Arrays kann definiert werden.

Beispiele:

array 0..3	of	boolean		
array 10..20	of	OK, y, z		
array 1..8	of	array -1..2	of	unsigned word[5]

3.2. Ausdrücke

Im Folgenden werden die verschiedenen Ausdrücke, die in nuXmv implementiert sind, näher erläutert.

3.2.1. Konstanten

Es gibt insgesamt 6 Konstanten in nuXmv:

1. Boolean Konstante: Einer der symbolischen Werte TRUE oder FALSE.
2. Symbolische Konstante: Einzigartige Werte, die vom Nutzer benannt werden.
3. Integer Konstante: Eine ganzzahlige Zahl, positiv oder negativ
4. Real Konstante: Eine rationale Zahl. Diese kann unterschiedlich repräsentiert werden, z.B. als Fließkommazahl, Bruch oder Exponentiell.

   Beispiele:

Fließkommazahl:	123.456
Bruch:	F'123/456
	f'123/456
Exponentiell:	123e4
	123.456e7
	123.456E7
	123.456E-7

5. Word Konstante: Word Konstanten beginnen immer mit einer 0, gefolgt von einem optionalen Buchstaben u(unsigned) oder s(signed) für die Vorzeichenbehaftung und anschließend einem der Buchstaben b/B(Binär), d/D(Dezimal), o/O(Octal) oder h/H(Hexadezimal), der die Basis des Words angibt. Daraufhin folgt eine optionale Zahl, die die Anzahl der Bits angibt, gefolgt von einem _ und dann dem Wert in der vorher angegebenen Basis. Wird der Indikator für die Vorzeichenbehaftung weggelassen, wird implizit angenommen, dass es sich um ein unsigned Word handelt. Wenn die Anzahl der Bits weggelassen wird, wird die Anzahl automatisch bestimmt.

Beispiele:

$0sb5\_10111$	ist vom Typ	signed word[5]
$0uo6\_37$	ist vom Typ	unsigned word[6]
$0d11\_9$	ist vom Typ	unsigned word[11]
$0sh12\_a9$	ist vom Typ	signed word[12]

6. Range Konstanten: Range Konstanten sind eine Menge von Integers, die aufeinander folgen. Syntaktisch werden diese wie folgt benannt: integer number .. integer number.

   Beispiele:

-1..5	entspricht der Menge	{-1,0,1,2,3,4,5}
1..10	entspricht der Menge	{1,2,3,4,5,6,7,8,9,10}

3.2.2. Basis Ausdrücke

Basis Ausdrücke treten in nuXmv am häufigsten auf und sind der Grundstein aller nuXmv Programme:

- Konstanten: Die unter Abschnitt 3.2.1 aufgezählten konstanten.
- Variablen
- Definitionen: Definitionen sind eine Art Makro. Jedes Mal, wenn eine Definition in einem Ausdruck enthalten ist, wird diese durch den Inhalt der Definition ersetzt.
- Klammern: Klammern werden benutzt, um Ausdrücke zu gruppieren.
- Logische und bitweise Operatoren: Können nur auf boolean, word- und unsigned words angewandt werden. Die Operatoren sind !(NICHT), &(UND), |(ODER), xor(exklusives ODER), $\rightarrow$(IMPLIKATION) und $\leftrightarrow$(ÄQUIVALENZ).
- Gleichheit und Ungleichheit (=, !=)
- Relationale Operatoren($<, <=, >, >=$)
- Arithmetische Operatoren ($+, -, *, /$)
- If-Then-Else (x ? y : z): Bei If-Then-Else wird die Bedingung x geprüft. Wenn diese zu TRUE auswertet, wird Bedingung y gewählt, andernfalls Bedingung z.
- next: Der next-Operator verweist auf den Wert einer Variablen im nächsten Zustand bzw. Zeitschritt. Der next-Operator kann nicht zweimal verwendet werden, next(next(x)) ist also **nicht** erlaubt.

3.2.3. Simple und Next Ausdrücke

Simple Ausdrücke bezeichnen Ausdrücke, die ausschließlich aus Variablen aus dem aktuellen Zustand bestehen, sie dürfen also **kein** *next* enthalten. Next-Ausdrücke dürfen hingegen Ausdrücke enthalten, die aus Variablen vom aktuellem und vom nächsten State bestehen, sie dürfen den next-Operator also enthalten.

3.3. Definition einer FSM

Eine Finite State Machine(FSM) beschreibt eine Menge von Statevariablen, die unterschiedliche Werte in unterschiedlichen Zuständen annehmen. Die Übergänge von einem Zustand zu beliebig vielen anderen Zuständen werden mithilfe von Transitionsübergängen bestimmt. Im Folgenden wird beschrieben, wie man eine FSM mit nuXmv bauen kann.

3.3.1. Variablen Deklarationen

Variablen können den Wert einer jeden Konstante annehmen, also boolean, integer, real, (un)signed word, enum, Menge(range) oder array. Außerdem können sie den Typ eines Modules annehmen, die in Abschnitt 3.3.6 besprochen werden.

3.3.2. State Variablen

State Variablen sind die am häufigsten vorkommende Art von Variablen. Sie definieren die Zustände in einer FSM. Deklariert werden sie wie folgt:

var_declaration :: **VAR** var_list
var_list :: identifier : type_specifier ;
| var_list identifier : type_specifier ;

3.3.3. DEFINE Deklarationen

Eine DEFINE Deklaration ist ein Ausdruck, der durch ein einzelnes Symbol ausgedrückt wird. Es kann somit als Makro interpretiert werden: Jedes Mal wenn das DEFINE in einem Ausdruck erscheint, wird dieses syntaktisch durch den damit in Verbindung gebrachten Ausdruck ersetzt. DEFINES dürfen dabei auch benutzt werden, wenn sie erst später im Code definiert wurden(Forward reference), allerdings sind Schleifen in den DEFINE's nicht erlaubt.

define_declaration :: **DEFINE** define_body
define_body :: identifier := next_expr ;
| define_body identifier := next_expr ;

3.3.4. INIT Bedingungen

In der INIT Bedingung wird durch einen booleschen Ausdruck die Menge der Startzustände ermittelt. Da sich in dieser nur simple Ausdrücke befinden dürfen, ist der next-Operator dort nicht erlaubt und es dürfen nur boolesche Ausdrücke verwendet werden. Gibt es mehr als eine INIT Bedingung, besteht die Menge der Startzustände aus der Vereinigung aller INIT Bedingungen.

init_constraint :: **INIT** simple_expr [;]

3.3.5. TRANS Bedingungen

Die TRANS Bedingung gibt die Transitionsübergänge zu dem Modell an. Sie besteht aus einer Menge von current/next Zustandspaaren. Ob sich ein Paar in der Menge befindet, wird durch einen booleschen Ausdruck bestimmt. Ist ein Ausdruck nicht vom Typ boolean wird ein Fehler geworfen. Gibt es mehrere TRANS Bedingungen, bestehen die Transitionsübergänge aus der Vereinigung aller TRANS Bedingungen.

trans_constraint :: **TRANS** next_expr [;]

3.3.6. MODULE Deklarationen

Eine MODULE Deklaration ist eine Sammlung von Deklarationen, Bedingungen und Spezifikationen. Sie haben einen eigenen Sichtbarkeitsbereich, wodurch identifier, die in anderen MODULE's verwendet wurden, hier erneut verwendet werden dürfen. Einmal deklariert, dürfen MODULE's so oft wie nötig verwendet werden. Dabei verweist jede Instanz eines MODULE's auf eine andere Datenstruktur. Wenn also ein Wert in einer MODULE Instanz geändert wird, dann wird er nur in dieser Instanz geändert und nicht in allen anderen, die möglicherweise nebenbei existieren. Außerdem können MODULE's Instanzen anderer MODULE's beinhalten, was den Bau einer strukturellen Hierarchie ermöglicht. Der Name, der direkt hinter dem MODULE Schlüsselwort steht, ist der Name des Modules. Diese haben einen separaten Namensraum und müssen sich dementsprechend von den Namen von Variablen und Deklarationen unterscheiden. Optional können Module auch formale Parameter entgegennehmen.

module :: **MODULE** identifier [(module_parameters)] [module_body]
module_parameters :: identifier
| module_parameters, identifier
module_body :: module_element
| module_body module_element
module_element :: var_declaration
| ivar_declaration
| frozenvar_declaration
| define_declaration
| constants_declaration
| assign_declaration
| trans_constraint
| init_constraint
| invar_constraint
| fairness_constraint
| ctl_specification
| invar_specification
| ltl_specification
| pslspec_specification
| compute_specification
| isa_declaration
| pred_declaration
| mirror_declaration

3.3.7. Ein Programm und das *main-Modul*

Ein nuXmv Programm ist eine Liste von Modulen. Eines dieser Module muss immer den Namen *main* ohne formale Parameter besitzen. Dieses Modul wird dann vom Interpreter von nuXmv ausgeführt.

program :: module_list
module_list :: module
| module_list module

3.4. Spezifikationen

Spezifikationen sind Eigenschaften, die von der erstellten FSM erfüllt werden sollen. Sie werden in Temporal Logiken wie der Computation Tree Logic(CTL) oder Linear Temporal Logic(LTL), welche um Vergangenheitsoperatoren erweitert wird, ausgedrückt. CTL und LTL Spezifikationen werden von nuXmv ausgewertet und anschließend die Wahr- oder Falschheit in der FSM bestimmt. Wenn eine Spezifikation sich als falsch herausstellt, konstruiert nuXmv ein Gegenbeispiel, welches den Verlauf, der zum Fehler

führt, darstellt und gibt dieses auf dem Ausgabekanal aus.

3.4.1. CTL Spezifikationen

CTL Spezifikationen werden durch das Schlüsselwort **CTLSPEC** oder **SPEC** eingeführt und bestehen aus CTL Formeln. Eine CTL Formel wertet zu *true* aus, wenn sie in **allen** Startzuständen *true* ergibt. Sie haben folgende Syntax:

ctl_specifikation ::	**CTLSPEC** ctl_expr [;]
	| **SPEC** ctl_expr [;]
	| **CTLSPEC NAME** name := ctl_expr [;]
	| **SPEC NAME** name := ctl_expr [;]

ctl_expr ::	simple_expr	a simple boolean expression
	| (ctl_expr)	
	| **!** ctl_expr	logical not
	| ctl_expr **&** ctl_expr	logical and
	| ctl_expr | ctl_expr	logical or
	| ctl_expr **xor** ctl_expr	logical exclusive or
	| ctl_expr **xnor** ctl_expr	logical NOT exclusive or
	| ctl_expr → ctl_expr	logical implies
	| ctl_expr ↔ ctl_expr	logical equivalence
	| **EG** ctl_expr	exists globally
	| **EX** ctl_expr	exists next state
	| **EF** ctl_expr	exists finally
	| **AG** ctl_expr	forall globally
	| **AX** ctl_expr	forall next state
	| **AF** ctl_expr	forall finally
	| **E** [ctl_expr U ctl_expr]	exists until
	| **A** [ctl_expr U ctl_expr]	forall until

Da eine *simple_expr* keinen next-Operator enthalten kann, kann eine *ctl_expr* ebenfalls keinen next-Operator enthalten. Die CTL Operatoren haben dabei folgende Bedeutung:

- **EX** p ist wahr in einem Zustand s, wenn **ein** Zustand s' existiert, sodass eine

Transition von s zu s' führt und p in s' wahr ist

- **AX** p ist wahr in einem Zustand s, wenn **für alle** Zustände s', bei denen eine Transition von s zu s' führt, p in s' wahr ist
- **EF** p ist wahr in einem Zustand s_0, wenn **eine** Folge von Transitionen $s_0 \rightarrow s_1, s_1 \rightarrow s_2, \cdots, s_{n-1} \rightarrow s_n$ existiert, sodass p in s_n wahr ist
- **AF** p ist wahr in einem Zustand s_0, wenn **für alle** Folgen von Transitionen $s_0 \rightarrow s_1, s_1 \rightarrow s_2, \cdots, s_{n-1} \rightarrow s_n$ p in s_n wahr ist.
- **EG** p ist wahr in einem Zustand s_0, wenn **eine** unendliche Folge von Transitionen $s_0 \rightarrow s_1, s_1 \rightarrow s_2, \cdots$ existiert, sodass p in jedem s_i wahr ist
- **AG** p ist wahr in einem Zustand s_0, wenn **für alle** unendlichen Folgen von Transitionen $s_0 \rightarrow s_1, s_1 \rightarrow s_2, \cdots$ p in jedem s_i wahr ist
- **E** $[p$ **U** $q]$ ist wahr in einem Zustand s_0, wenn **eine** Folge von Transitionen $s_0 \rightarrow s_1, s_1 \rightarrow s_2, \cdots, s_{n-1} \rightarrow s_n$ existiert, sodass p in jedem Zustand von s_0 bis s_{n-1} wahr ist und q in s_n wahr ist
- **A** $[p$ **U** $q]$ ist wahr in einem Zustand s_0, wenn **für alle** Folgen von Transitionen $s_0 \rightarrow s_1, s_1 \rightarrow s_2, \cdots, s_{n-1} \rightarrow s_n$ p in jedem Zustand von s_0 bis s_{n-1} wahr ist und q in s_n wahr ist.

3.4.2. LTL Spezifikation

LTL Spezifikationen werden durch das Schlüsselwort **LTLSPEC** eingeführt und bestehen aus LTL Formeln. Ihre Syntax ist wie folgt:

ltl_specification ::	**LTLSPEC** ltl_expr [;]	
	LTLSPEC NAME name := ltl_expr [;]	
ltl_expr ::	next_expr	a next boolean expression
	| (ltl_expr)	
	| **!** ltl_expr	logical not
	| ltl_expr **&** ltl_expr	logical and
	| ltl_expr | ltl_expr	logical or
	| ltl_expr **xor** ltl_expr	logical exclusive or
	| ltl_expr **xnor** ltl_expr	logical NOT exclusive or
	| ltl_expr → ltl_expr	logical implies
	| ltl_expr ↔ ltl_expr	logical equivalence
	FUTURE	
	X ltl_expr	next state
	G ltl_expr	globally
	G bound ltl_expr	bounded globally
	F ltl_expr	finally
	F bound ltl_expr	bounded finally
	ltl_expr **U** ltl_expr	until
	ltl_expr **V** ltl_expr	releases
	PAST	
	Y ltl_expr	previous state
	Z ltl_expr	not previous state
	H ltl_expr	historically
	H bound ltl_expr	bounded historically
	O ltl_expr	once
	O bound ltl_expr	bounded once
	ltl_expr S ltl_expr	since
	ltl_expr T ltl_expr	triggered

Die LTL Operatoren haben dabei folgende Bedeutung:

- **X** p ist wahr zum Zeitpunkt t, wenn p zum Zeitpunkt $t+1$ wahr ist
- **F** p ist wahr zum Zeitpunkt t, wenn p zu irgendeinem Zeitpunkt $t' \geq t$ wahr ist

- **F** *[l,u]* p ist wahr zum Zeitpunkt t, wenn p zu irgendeinem Zeitpunkt $t+l \leq t' \leq t+u$ wahr ist
- **G** p ist wahr zum Zeitpunkt t, wenn p zu jedem Zeitpunkt $t' \geq t$ wahr ist
- **G** *[l,u]* p ist wahr zum Zeitpunkt t, wenn p zu jedem Zeitpunkt $t+l \leq t' \leq t+u$ wahr ist
- p **U** q ist wahr zum Zeitpunkt t, wenn q zu irgendeinem Zeitpunkt $t' \geq t$ und p zu jedem Zeitpunkt $t \leq t'' < t'$ wahr ist
- p **V** q ist wahr zum Zeitpunkt t, wenn q zu jedem Zeitpunkt $t' \leq t$ wahr ist, bis einschließlich dem Zeitpunkt *t"*, bei dem p ebenfalls wahr ist. Es kann sein, dass p niemals wahr wird. In diesem Fall muss q zu jedem Zeitpunkt $t' \geq t$ wahr sein
- **Y** p ist wahr zum Zeitpunkt $t > t_0$, wenn p zum Zeitpunkt $t-1$ wahr ist. **Y** p ist falsch zum Zeitpunkt t_0
- **Z** p ist äquivalent zu **Y** p mit der Ausnahme, dass die Aussage zum Zeitpunkt t_0 wahr ist
- **H** p ist wahr zum Zeitpunkt t, wenn p zu jedem vorherigen Zeitpunkt $t' \leq t$ wahr ist
- **H** *[l,u]* p ist wahr zum Zeitpunkt t, wenn p zu jedem Zeitpunkt $t-u \leq t' \leq t-l$ wahr ist
- **O** p ist wahr zum Zeitpunkt t, wenn p zu mindestens einem vorherigen Zeitpunkt $t' \leq t$ wahr war
- p **S** q ist wahr zum Zeitpunkt t, wenn q zum Zeitpunkt $t' \leq t$ wahr war und p zu jedem Zeitpunkt $t' < t'' \leq t$ wahr ist
- p **T** q ist wahr zum Zeitpunkt t, wenn p zum Zeitpunkt $t' \leq t$ wahr war und q zu jedem Zeitpunkt $t' \leq t'' < t$ wahr ist. Wenn p nie wahr war, muss q zu jedem Zeitpunkt $t_0 \leq t'' \leq t$ wahr sein.

Eine LTL Formel ist wahr, wenn sie zum Ausgangszeitpunkt t_0 wahr ist.

4. Modellierung

In diesem Kapitel geht es nun um die Modellierung des Gleisnetzes und des Stellwerkes, sowie die Verifikation, dass diese zusammen keine Gefahr eines Unfalls beherbergen. Weiterhin wird am Ende noch auf die Dekomposition des modellierten Gleisnetzes eingegangen, um den Zustandsraum für nuXmv zu verkleinern.

4.1. Das Modell

Für die Modellierung des Gleisnetzes und des Stellwerkes müssen verschiedene Aspekte mit eingebracht werden. Zum einen natürlich die Weichen und geraden Streckenelemente, aus denen das Gleisnetz besteht. Zum anderen die Routen, mit denen das Stellwerk arbeitet, um die Züge unfallfrei über das Gleisnetz zu führen. Außerdem müssen die Allokierung der Routen, das Stellen der Weichen, die Bewegung der Züge und das Auslösen eines Notstopps bei einem Safety Error durch Transitionen korrekt wiedergegeben werden. Als Ansatz dafür habe ich mich an der Modellierung des neuen dänischen Stellwerkes orientiert. [HHP17] Das dänische Stellwerk benutzt ähnlich wie unser Stellwerk Weichen, gerade Streckenelemente und Routen, hat allerdings noch einige Features wie das Sequential Release mehr zur Verfügung, die wir nicht besitzen. Für die Modellierung des dänischen Stellwerkes wurde die zeitliche Komponente der Abläufe mit berücksichtigt. Das bedeutet, dass die Allokierung der Routen schneller stattfindet als das Stellen der Weichen und dass das Stellen der Weichen schneller vonstattengeht als das Fahren der Züge auf dem Gleis. Die Anfrage von Zügen, eine Route befahren zu dürfen, kann jedoch jederzeit stattfinden. Diesen Ansatz habe ich in den Grundzügen übernommen, jedoch an einigen Stellen an die Funktionsweise des Stellwerkes aus dem Bachelorprojekt angepasst.
In einem ersten Anlauf habe ich versucht das gesamte Gleisnetz mitsamt aller Routen aus dem Stellwerk zu modellieren. Hierbei hat sich jedoch herausgestellt, dass eine State Explosion auftritt, d.h. dass der Zustandsraum so groß wurde, dass nuXmv nicht mehr terminiert ist. Daher habe ich das Gleisnetz in einem zweiten Versuch durch eine Dekomposition in mehrere Teilnetzwerke aufgeteilt. Durch die Verifikation, dass alle diese Teilnetzwerke unfallfrei sind, lässt sich folgern, dass auch das Gesamtsystem unfallfrei ist. Die Idee hierfür stammt von Anne E. Haxthausen, Alessandro Fantechi und Hugo Macedo, die sich mit der kompositionellen Verifikation von Bahnsystemen beschäftigen. [MFH16] [MFH17] [FHM17]
Im Folgenden werde ich zunächst den Aufbau meines Modells beschreiben und anschließend auf die Dekomposition des Gleisnetzes eingehen.

4.2. Die Variablen

Zunächst habe ich Variablen für das Modell erstellt, die zum einen das Gleisnetz(Track Elemente, Weichen) und zum anderen das Stellwerk(Routen) widerspiegeln sollen. Bei den Track Elementen habe ich für die Erkennung, ob sich ein Zug auf diesen befindet oder nicht, zwei separate Variablen eingeführt: U2D(Up to Down, Zug fährt im Uhrzeigersinn) und D2U(Down to Up, Zug fährt gegen den Uhrzeigersinn). Beide Variablen geben an, ob sich ein Zug auf dem Track Element befindet oder nicht und zusätzlich speichern diese, in welche Richtung der Zug momentan fährt. Dies resultiert daraus, dass ich in meiner Modellierung Züge nicht explizit darstelle, sondern nur über den Belegtheitsstatus der Track Elemente arbeite. Da ich jedoch allein durch den Belegtheitsstatus nicht weiß, in welche Richtung der Zug als nächstes fährt, speichere ich mir diese Information zusätzlich. Dabei ist zu beachten, dass wenn sich ein „Zug“ auf einem Track Element befindet, nur U2D oder D2U auf belegt, also 1, gesetzt wird, je nachdem in welche Richtung der Zug fährt, aber nicht beide auf einmal.

Weiterhin haben Track Elemente eine MB(Markerboard) Variable. Diese gibt an, ob ein Zug, der auf diesem Track Element steht, weiterfahren darf oder nicht. In der eigentlichen Implementierung des Stellwerkes wird dieses Signal direkt an den Zug geschickt. Da ich in meiner Modellierung jedoch auf explizite Züge verzichtet habe, gebe ich dieses Signal nun an die Track Elemente, die den Start der freigegebenen Route markieren, weiter. In Listing 1 wird die Variablendeklaration eines Track Elementes gezeigt. Die ersten beiden Variablen können Werte zwischen 0 und 2 annehmen und die dritte Variable Werte zwischen 0 und 1.

```
VAR
        te_107e_U2D  :  0..2;
        te_107e_D2U  :  0..2;
        te_107e_MB   :  0..1;
```

Listing 1: Track Element Variablen

Weiterhin habe ich die Weichen des Stellwerkes mit jeweils 3 Werten modelliert: U2D, D2U(wie bei den Track Elementen) und POS(Position der Weiche). Der Belegtheitsstatus der Weiche wird genau wie bei den Track Elementen behandelt. Anzumerken ist hier jedoch, dass wir beim eigentlichen Stellwerk nicht die Möglichkeiten haben, einen Zug an einer Weiche zu tracken. Ich habe diese Option trotzdem hinzugefügt, da ich ansonsten nicht nachweisen könnte, dass Züge auch an Weichen nicht verunfallen können.

POS gibt die aktuelle Position der Weiche an, also PLUS oder MINUS. PLUS bedeutet dabei immer, der Zug fährt geradeaus, MINUS der Zug biegt ab. Die Positionen

von Weichen ändern sich immer dann, wenn eine Route für einen Zug blockiert wird. Listing 2 zeigt die Variablendeklaration für Weichen. Die ersten zwei Variablen nehmen Werte zwischen 0 und 3 an und die POS Variable Werte zwischen 0 und 1.

```
p_1_U2D : 0..3;
p_1_D2U : 0..3;
p_1_POS : 0..1;
```

Listing 2: Weichen Variablen

Eine Besonderheit bilden sogenannte Kreuzweichen, bei denen Züge in jeder Richtung von zwei unterschiedlichen Gleisabschnitten auf die Weiche fahren können. Je nach Weichenstellung und auch Herkunftsort des Zuges kann dieser auch wieder auf zwei verschiedene Gleisabschnitte herunterfahren. Da der Herkunftsort des Zuges mitbestimmend dafür ist, auf welchen Gleisabschnitt der Zug als Nächstes fährt, habe ich für Kreuzweichen eine zusätzliche Variable STRAIGHT eingeführt. Diese gibt an, ob ein Zug sich „gerade“ auf eine Weiche bewegt oder ob er von einem oberen/unteren Gleisabschnitt kommt. Die Variablendeklaration dieser wird in Listing 3 gezeigt. Die Variablen U2D und D2U nehmen Werte zwischen 0 und 3 an, die anderen Variablen Werte zwischen 0 und 1.

```
p_9_U2D : 0..3;
p_9_D2U : 0..3;
p_9_POS : 0..1;
p_9_STRAIGHT : 0..1;
```

Listing 3: Kreuzweichen Variablen

Und zuletzt habe ich für jede Route, die in der Interlockingtable steht, eine Variable für den Zustand, in der sich die Route zurzeit befindet. Diese können insgesamt 7 verschiedene Zustände annehmen(FREE, MARKED, CONFLICT, ACTIVE WAIT, LOCKED, OCCUPIED, DONE). Die Variablendeklaration für Routen zeigt Listing 4. Diese nehmen Werte zwischen 0 und 6 an.

```
r_3_MODE : 0..6;
```

Listing 4: Routen Variablen

U2D und D2U für Track Elemente und Weichen haben einen Wertebereich von 0..2 bei Track Elementen und 0..3 bei Weichen, Routen haben einen Wertebereich zwischen 0 und 6 und alle anderen Werte einen Bereich von 0..1. Die unterschiedlichen Wertebereiche für U2D und D2U bei Track Elementen und Weichen rühren daher, dass bei einem Track Element, das bereits belegt ist, maximal ein Zug aus derselben Richtung kommen kann. Bei Weichen kann es jedoch sein, dass bei einem belegten Gleis ein Zug von der PLUS und einer von der MINUS Position kommt. Sobald dabei mehr als ein

Zug auf einem Track Element oder einer Weiche steht heißt das, es ist zu einem Unfall gekommen. Die Werte und ihre Bedeutung werden in den Tabellen 1 bis 4 erläutert.

Wert	Status	Bedeutung
0	nicht belegt	Es befindet sich zurzeit kein Zug auf diesem Streckenelement
1	belegt	Es befindet sich zurzeit ein Zug auf dem Streckenelement
2 & 3	mehrfach belegt	Es befinden sich mehrere Züge auf einem Streckenelement, mindestens 2 Züge sind verunfallt

Tabelle 1: Belegtheitsstatus von Track Elementen und Weichen

Wert	Status	Bedeutung
0	Stop	Ein Zug, der auf diesem Track Element steht, darf nicht weiterfahren
1	Freie fahrt	Ein Zug, der auf diesem Track Element steht, darf weiterfahren

Tabelle 2: Status von Marker Board Signalen

Wert	Status	Bedeutung
0	PLUS	Die Weiche ist in PLUS Position und Züge fahren geradeaus weiter
1	MINUS	Die Weiche ist in MINUS Position und Züge biegen ab

Tabelle 3: Status einer Weiche

Wert	Status	Bedeutung
0	FREE	Die Route ist frei. Sie wird von keinem Zug belegt und kein Zug möchte diese befahren
1	MARKED	Ein Zug möchte diese Route befahren
2	CONFLICTED	Eine Route, die mit dieser Route in Konflikt steht, wird gerade befahren
3	ACTIVE WAIT	Ein Zug möchte diese Route befahren, sie ist aber noch durch einen Konflikt gesperrt
4	LOCKED	Die Route wurde für einen Zug gesperrt/freigegeben und wird bald befahren
5	OCCUPIED	Ein Zug befährt gerade diese Route
6	DONE	Ein Zug hat die Route wieder verlassen und sie wird in kurzer Zeit wieder freigegeben

Tabelle 4: Status einer Route

4.3. Initialisierung

Initial nehmen alle Variablen den Wert 0 an. Das bedeutet, dass alle Routen frei sind, alle Weichen auf Position PLUS stehen und sich kein Zug auf dem Gleisnetz befindet.

Züge befahren später das Gleisnetz, indem eine Route, die von außen in das Gleisnetz hineinführt, angefragt wird. In so einem Fall „materialisiert“ sich dann ein Zug und befährt das Gleisnetz. Die Initialisierung wird in Listing 5 anhand des Track Elementes 107e gezeigt. Dasselbe wird dann für alle anderen Variablen auch gemacht.

```
INIT
        (te_107e_U2D = 0) &
        (te_107e_D2U = 0) &
        (te_107e_MB = 0) &
```

Listing 5: Initialisierung

4.4. Transitionen

Im Folgenden werde ich die Transitionsübergänge meiner Modellierung anhand der Funktionsweise des Stellwerkes erläutern.

4.4.1. Allgemeiner Ablauf

Im Hintergrund läuft ständig der Safety-Monitor des Stellwerkes. Dieser überprüft, ob Track Elemente des Gleisnetzes belegt sind, obwohl sie nicht belegt sein sollten. Ist dies der Fall, wird sofort ein Notstopp ausgelöst.
Routen können jederzeit von einem Zug beantragt werden. Dies setzt den Status einer Route, wenn diese nicht bereits durch einen anderen Zug oder eine andere Route gesperrt wird, auf MARKED. Gleichzeitig kann sich aber auch der Status einer Route anderweitig ändern, z.B. wenn sie von MARKED auf LOCKED oder von OCCUPIED auf DONE gesetzt wird. Diese Routenänderungen passieren in Bruchteilen einer Sekunde und werden daher, neben dem Notstopp, immer als erstes ausgeführt. Dabei wird immer entweder eine Route beantragt oder ein Routenstatus geändert. Liegen keine Bedingungen vor, um den Routenstatus zu ändern, werden im Anschluss die Züge in Bewegung gesetzt, soweit diese denn fahren dürfen. Ist eine Route gesperrt, die von außerhalb in das Gleisnetz hineinführt, „materialisiert“ sich am Eingangspunkt der Route ein Zug. Dieser befährt dann so lange das Gleisnetz, bis er an einem Markerboard eines Track Elementes halten muss, da dieses auf 0 gesetzt ist. Im Idealfall gelangt er somit immer an das gewünschte Ziel, ohne einen Unfall zu verursachen. Sollte ein Zug auf einem Track Element stehen, das in keinem Pfad einer Route mit Status LOCKED oder OCCUPIED enthalten ist, wird ein Notstopp ausgelöst. Das bedeutet, dass keine Züge mehr fahren, keine Weichen mehr schalten und sich der Status der Routen nicht mehr ändert. Der Notstopp wird vor allen anderen Transitionen ausgeführt, wenn die oben genannte Bedingung erfüllt ist.
Listing 6 stellt dabei die Transitionsbedingungen für mein Modell dar und soll den oben beschriebenen Ablauf modellieren. Begriffe wie *safety_error*, *emergency_stop*

usw. sind dabei Makros, die ich mithilfe von DEFINE Deklarationen erstellt habe. Die TRANS Bedingung stellt den folgenden Ablauf dar: Die Bedingung safety_error prüft, ob ein Zug sich auf einem Track Element befindet, das nicht in einer Route enthalten ist, die LOCKED oder OCCUPIED ist. Ist diese Bedingung erfüllt, wird die Transition emergency_stop genommen. In dieser nehmen Variablen keine Änderungen vor, sondern behalten ihre derzeitigen Werte bei. Liegt kein safety_error vor, wird stattdessen entweder mit request_route eine Route von einem Zug beantragt oder mit condition_for_route_mode_switch_fullfilled geprüft, ob der Status einer Route geändert werden kann. Wird die Transition request_route genommen, wechselt eine Route aus dem Gleisnetz auf den Zustand MARKED und bei der nächsten Transition wird die TRANS Bedingung von vorne geprüft. Ist die Bedingung condition_for_route_mode_switch_fullfilled erfüllt, wird die Transition switch_route_mode genommen, in der sich der Status einer Route ändert und gegebenenfalls Weichen gestellt und Markerboard Signale gesendet werden. Ist die Bedingung nicht erfüllt, wird stattdessen die Transition move_train genommen, in der sich ein Zug auf dem Gleisnetz bewegt, vorausgesetzt er darf fahren. In beiden Fällen wird bei der nächsten Transition die TRANS Bedingung erneut geprüft.

```
TRANS
        (
        safety_error ? emergency_stop : (request_route | (
            condition_for_route_mode_switch_fullfilled ? switch_route_mode
             : move_train))
        )
```

Listing 6: Transitionsübergänge

4.4.2. Safety Error und Notstopp

Ein Safety Error liegt vor, sobald ein Track Element belegt ist, obwohl keine Route gesperrt oder belegt ist, die über dieses Track Element führt. In diesem Fall wird sofort ein Notstopp ausgelöst, der alle Züge anhält und keine Routenstatus-, Markerboard- und Weichenänderungen zulässt. In Listing 7 wird die Modellierung des safety_error anhand des Track Elementes 107e und des emergency_stop gezeigt. Das Track Element 107e wird nur durch die Route 17 befahren. Daher soll, wenn das Track Element belegt ist, auch der Status der Route 17 LOCKED oder OCCUPIED sein. Ansonsten liegt ein Safety Error vor. Beim emergency_stop werden DEFINE's verwendet, die den nächsten Wert aller Variablen unverändert lassen.

```
safety_error :=
        (((te_107e_U2D + te_107e_D2U != 0) &
        (r_17_MODE != 4) & (r_17_MODE != 5))
```

```
emergency_stop :=
        no_route_change &
        no_mb_change &
        no_point_change &
        no_train_movement;
```

Listing 7: safety error und Notstopp

4.4.3. Änderungen des Routenstatus

In den folgenden Abschnitten spielt die zeitliche Abfolge eine entscheidende Rolle. So werden nachfolgend die Bedingungen geprüft, ob sich der Status einer Route oder einer Zugposition ändert. Dabei finden Änderungen der Route schneller statt als Änderungen an der Zugposition. Daher werden die Bedingungen einzeln und nacheinander geprüft. Die erste Bedingung betrifft die Änderung des Routenstatus. Sollte eine Bedingung erfüllt sein, um den Status einer Route zu ändern, wird dieser geändert. Die möglichen Änderungen des Routenstatus und wann sie auftreten werden in der folgenden Tabelle erläutert:

aktueller Status	nächster Status	Bedingung
free	marked	Eine Route ist frei und ein Zug möchte diese nun befahren
marked	locked	Die Track Elemente der Route und alle Konfliktrouten sind frei
locked	occupied	Das erste Track Element der Route ist belegt
occupied	done	Alle Track Elemente der Route sind frei
done	free	Ist immer möglich. Wird in der echten Anwendung zum Aufräumen von Listen verwendet
free	conflicted	Eine Konfliktroute wird auf LOCKED gesetzt
marked	active wait	Eine Konfliktroute wird auf LOCKED gesetzt
conflicted	free	Die Konfliktroute ist wieder frei
active wait	marked	Die Konfliktroute ist wieder frei

Tabelle 5: Mögliche Statusänderungen einer Route und wann sie auftreten

4.4.4. Routen beantragen

Grundsätzlich kann ein Zug zu jeder Zeit eine Route anfragen. Wenn die Route bereits gesperrt, belegt oder durch eine andere Route gesperrt ist, muss der Zug so lange warten, bis die Route wieder freigegeben ist. Ansonsten wird der Status der Route auf MARKED gesetzt.

In meiner Modellierung erlaube ich es daher, eine Route zu jedem Zeitpunkt auf MARKED zu setzen, solange sie nicht LOCKED, OCCUPIED, DONE oder CONFLICT ist. Dabei wird lediglich der Status dieser Route auf MARKED geändert, nicht jedoch der Status anderer Routen, Markerboard Signale, Weichenpositionen oder Zugstandorte. Dies wird in Listing 8 anhand der Route 3 aufgezeigt.

```
request_route :=
        (((r_3_MODE != 2) &
        (r_3_MODE != 4) &
        (r_3_MODE != 5) &
        (r_3_MODE != 6) &
        switch_r_3_to_marked &
        no_mb_change &
        no_point_change &
        no_train_movement)
```

Listing 8: Anfragen einer Route

4.4.5. Routen blockieren/sperren

Wurde eine Route von einem Zug erfolgreich beantragt und ist nun im Status MARKED, wird als nächstes überprüft, ob die Route für den Zug gesperrt werden kann. Dafür müssen alle Track Elemente der Route frei sein und es darf keine Konfliktroute für einen anderen Zug gesperrt sein oder bereits befahren werden. Listing 9 zeigt dies am Beispiel der Route 3.

```
condition_for_route_mode_switch_fullfilled :=
        (((r_3_MODE = 1) &
        r_3_tes_free &
        r_3_conflicts_not_locked_or_occupied)
```

Listing 9: Bedingungen für Statuswechsel von MARKED zu LOCKED

Bei der Überprüfung, ob die Konfliktrouten frei sind, werden zunächst alle Routen durchlaufen und für die aktuelle Route nachgesehen, ob es sich um eine Konfliktroute handelt. Dies wird anhand einer Konfliktmatrix ermittelt, in der für jede Route steht, ob sie sich mit der zu vergleichenden Route im Konflikt befinden. Die Werte dafür wurden aus der entsprechenden Interlockingtable aus dem Bachelorprojekt entnommen. Wenn es sich bei einer Route dann um eine Konfliktroute handelt, wird überprüft, ob diese sich bereits im Status LOCKED oder OCCUPIED befindet. Wenn ja, darf die Route nicht befahren werden und die nächste Bedingung wird geprüft. Bei einer Route, die keine Konfliktroute ist, müssen keine Bedingungen geprüft werden und es wird daher lediglich TRUE eingesetzt.

In Listing 10 wird diese Überprüfung am Beispiel der Route 3 gezeigt. Das Array conflicts ist dabei die Konfliktmatrix, welche die Konfliktrouten aus der Interlockingtable mit den booleschen Werten TRUE und FALSE speichert. Am Beispiel wird dabei für die Route 3 geprüft, ob die Route 4 mit ihr im Konflikt steht. Ist das der Fall, wird für die Route 4 geprüft, ob sich diese im Status LOCKED oder OCCUPIED befindet. Andernfalls soll keine Bedingung geprüft werden, was in der Modellierung durch die Überprüfung auf TRUE realisiert wird.

```
r_3_conflicts_not_locked_or_occupied :=
        ((conflicts[3][4]) ? (r_4_MODE != 4 & r_4_MODE != 5) : TRUE) &
```

Listing 10: Prüfung, ob Konfliktrouten frei sind

Sind all diese Bedingungen erfüllt, werden die Konfliktrouten auf CONFLICT oder ACTIVE WAIT gesetzt, je nachdem, ob sie sich vorher im Status FREE oder MARKED befunden haben. Außerdem werden die Positionen der Weichen entsprechend der Informationen aus der Interlockingtable gestellt, der Status der Route auf LOCKED aktualisiert und das Markerboard des Start-Track-Elementes der Route auf 1 gesetzt. Auch hier bewegen sich die Züge noch nicht. Die Modellierung wird im Listing 11 anhand der Route 3 gezeigt.

```
switch_route_mode :=
        (((r_3_MODE = 1) &
        r_3_tes_free &
        r_3_conflicts_not_locked_or_occupied &
        switch_r_3_to_locked_and_set_conflicts &
        switch_points_for_r_3 &
        set_signal_for_r_3 &
        no_train_movement)
```

Listing 11: Statuswechsel von MARKED zu LOCKED

Beim Sperren der Konfliktrouten wird ähnlich vorgegangen wie beim Überprüfen, ob diese bereits belegt sind. Es werden wieder alle Routen mit der aktuellen Route verglichen und überprüft, ob diese im Konflikt stehen und ob diese sich im Status FREE oder MARKED befinden. Ist das der Fall, werden sie entsprechend auf den Status CONFLICT bzw. ACTIVE WAIT gesetzt. Ansonsten behalten sie den Zustand bei, in dem sie sich zurzeit befinden.
Listing 12 zeigt wieder ein Beispiel für die Route 3. Der Status der Route 3 wird dabei in der ersten Zeile auf LOCKED gesetzt. In der zweiten Zeile findet wieder die Überprüfung statt, ob die Route 3 mit Route 4 im Konflikt steht. Zusätzlich soll die Route sich im Status FREE oder MARKED befinden. Sind beide Bedingungen erfüllt, wird der Status der Route 4 um 2 erhöht, wodurch die Route sich im Status CONFLICT befindet, wenn sie vorher auf FREE war und im Status ACTIVE WAIT wenn sie vor-

her auf MARKED war. Sind die Anfangsbedingungen nicht erfüllt, soll der Status der Route 4 unverändert bleiben.

```
switch_r_3_to_locked_and_set_conflicts :=
        (next(r_3_MODE) = 4) &
        ((conflicts[3][4] & (r_4_MODE = 0 | r_4_MODE = 1)) ? (next(
            r_4_MODE) = r_4_MODE + 2) : next(r_4_MODE) = r_4_MODE) &
```

Listing 12: Routenstatus auf LOCKED setzen und Konfliktrouten sperren

Beim Stellen der Weichen wird eine Weichenmatrix durchlaufen, die Informationen über Weichenpositionen aus der Interlockingtable pro Route gespeichert hat. Weichen, die dabei nicht von einer Route beeinflusst werden, enthalten den Wert -1, Weichen die auf PLUS gesetzt werden 0 und Weichen die auf MINUS gesetzt werden 1. Dabei wird die Position der Weichen, die gestellt werden müssen, auf den Wert in der Weichenmatrix gesetzt. Alle anderen Weichen behalten ihre derzeitige Position bei.
Listing 13 zeigt das Stellen der Weiche 1 für die Route 3. route_points ist dabei die Weichenmatrix, die an erster Stelle die Nummer der Route und an zweiter Stelle die Nummer der Weiche minus 1 übergeben bekommt. Die Weichennummer muss dabei minus 1 genommen werden, da unsere Weichennummerierung bei 1 beginnt und der Array Index bei 0. Steht an dieser Position keine -1 in der Weichenmatrix, wird die Position der Weiche auf den Wert der Weichenmatrix geändert. Dieser beträgt 0, wenn die Weiche auf PLUS gestellt werden soll und 1, wenn die Weiche auf MINUS gestellt werden soll. Andernfalls soll die Position der Weiche gleich bleiben.

```
switch_points_for_r_3 :=
        ((route_points[3][0] != -1) ? (next(p_1_POS) = route_points
            [3][0]) : (next(p_1_POS) = p_1_POS)) &
```

Listing 13: Weichenposition ändern

Beim Senden des Signales an den Zug wird einfach die Variable MB für das entsprechende Start-Track-Element der Route auf 1 gesetzt, während alle anderen unverändert bleiben. Dies wird in Listing 14 anhand der Route 3 gezeigt, die auf dem Track Element 4e beginnt.

```
set_signal_for_r_3 :=
        (next(te_1te_MB) = te_1te_MB) &
        (next(te_4e_MB) = 1) &
        (next(te_4te_MB) = te_4te_MB) &
```

Listing 14: Signal an das Track Element zur Freigabe einer Route setzen

4.4.6. Route wird befahren

Der Status einer Route wechselt von LOCKED zu OCCUPIED, sobald das erste Track Element der Route belegt ist, also ein Zug die Route befährt. Listing 15 zeigt dies erneut am Beispiel der Route 3.

```
((r_3_MODE = 4) &
((te_4te_U2D + te_4te_D2U) > 0))
```

Listing 15: Bedingungen für Statuswechsel von LOCKED zu OCCUPIED

Entsprechend wird, sobald diese Bedingung erfüllt wird, auch lediglich der Status der Route von LOCKED auf OCCUPIED aktualisiert. Alle anderen Variablen behalten ihren vorherigen Wert bei.
Listing 16 zeigt den Statuswechsel der Route 3 von LOCKED zu OCCUPIED. Dabei wird der Status der Route 3 auf OCCUPIED gesetzt, während alle anderen Variablen ihren alten Wert beibehalten.

```
        switch_r_3_to_occupied &
        no_mb_change &
        no_point_change &
        no_train_movement

switch_r_3_to_occupied :=
        (next(r_3_MODE) = 5) &
        (next(r_4_MODE) = r_4_MODE) &
        (next(r_5_MODE) = r_5_MODE) &
```

Listing 16: Statuswechsel von LOCKED zu OCCUPIED

4.4.7. Zug verlässt seine Route

Sobald ein Zug seine Route verlässt und kein Gleisabschnitt der Route mehr belegt ist, wird der Status der Route von OCCUPIED auf DONE gesetzt und die Konfliktrouten wieder freigegeben. Der Statuswechsel wird in Listing 17 am Beispiel der Route 3 gezeigt.

```
((r_3_MODE = 5) &
r_3_tes_free &
r_3_points_free &
switch_r_3_to_done_and_free_conflicts &
no_mb_change &
no_point_change &
no_train_movement)
```

Listing 17: Statuswechsel von OCCUPIED zu DONE

Eine Ausnahme bilden dabei die Routen, die auf den Track Elementen 100, 101 und 102 Enden. Züge können an diesen Track Elementen nicht weiter fahren, sondern nur umdrehen. Da die Routen, die von den Track Elementen 100, 101 und 102 wegführen jedoch alle im Konflikt mit Routen stehen, die zu diesen Track Elementen hinführen, müssen diese Routen früher wieder freigegeben werden, da die Züge sonst nicht fahren können. Daher wird in diesen Fällen die Route bereits wieder freigegeben, wenn die Züge am Ende der Route angekommen sind. Dadurch kann der Zug an diesen Track Elementen umdrehen und weiterfahren. Zu Konflikten mit anderen Zügen kann es dabei nicht kommen, da die Track Elemente 100, 101 und/oder 102 jeweils durch einen Zug belegt sind und dadurch kein zweiter Zug eine Route zu diesen Track Elementen befahren darf.
Im Listing 18 wird der Richtungswechsel beim Track Element 100 gezeigt. Die Route 43 hat dabei das Track Element 100 als Endziel. Wenn der Zug also Route 43 befährt und am Track Element 100 ankommt, wird die Richtung des Zuges in swap_train_direction_100 geändert, indem die Variable te_100_D2U auf 0 und die Variable te_100_U2D auf 1 gesetzt wird. Anschließend wird der Status der Route auf DONE gesetzt und die Konfliktrouten wieder freigegeben.

```
((r_43_MODE = 5) &
(te_100_D2U = 1) &
swap_train_direction_100 &
switch_r_43_to_done_and_free_conflicts &
no_mb_change &
no_point_change)
```

Listing 18: Richtungswechsel eines Zuges

Das Freigeben der Konflikte folgt dabei wieder demselben Prinzip wie beim Sperren der Konflikte: Die Route wird mit allen Routen verglichen und es wird überprüft, ob diese im Konflikt miteinander stehen und ob der Status der Routen CONFLICT oder ACTIVE WAIT ist. Ist beides der Fall, wird der Status der Route wieder auf FREE bzw. MARKED gesetzt. Ansonsten bleibt der Status der Route unverändert.
Listing 19 zeigt das Freigeben der Konflikte für Route 3. Hier wird genauso vorgegangen wie beim Sperren der Konflikte aus Listing 12, außer das hier der Status der Route 4 auf CONFLICT oder ACTIVE WAIT sein soll statt FREE oder MARKED. Wird die Bedingung erfüllt, soll der Status um 2 verringert werden, sodass der Status der Route 4 dann FREE ist, wenn er vorher CONFLICT war und MARKED, wenn er vorher ACTIVE WAIT war. Ansonsten soll der Status der Route sich nicht ändern.

```
switch_r_3_to_done_and_free_conflicts :=
        (next(r_3_MODE) = 6) &
        ((conflicts[3][4] & (r_4_MODE = 2 | r_4_MODE = 3)) ? (next(
```

```
r_4_MODE) = r_4_MODE - 2) : (next(r_4_MODE) = r_4_MODE)) &
```

Listing 19: Routenstatus auf DONE setzen und Konflikte freigeben

4.4.8. Route wird wieder freigegeben

Der Zustand DONE wird in der eigentlichen Implementierung für Aufräumzwecke wie das Freigeben von Speicher verwendet. Sobald diese erledigt sind, wechselt der Status der Route von DONE zu FREE. Da dieser Zustand nicht im Zusammenhang mit dem Geschehen auf dem Gleisnetz steht, kann in der Modellierung die Route in diesem Zustand jederzeit auf FREE wechseln. Dies zeigt Listing 20 am Beispiel der Route 3.

```
((r_3_MODE = 6) &
switch_r_3_to_free &
no_mb_change &
no_point_change &
no_train_movement)
```

Listing 20: Statuswechsel von DONE zu FREE

4.4.9. Ein Zug bewegt sich

Im letzten Schritt wird die Bewegung der Züge simuliert. Dabei wird immer das nächste Track Element oder die nächste Weiche belegt, welche auf einen derzeitig belegten Gleisabschnitt folgt. Die Folgeposition des Zuges wird dabei aus der Richtung des Zuges (U2D oder D2U), den Weichenpositionen auf der Strecke und bei Kreuzweichen zusätzlich vom Gleisabschnitt, von dem der Zug kommt, bestimmt. Züge fahren grundsätzlich immer zum nächsten Gleisabschnitt weiter, außer das Markerboard eines Track Elementes steht auf 0. Dann bleibt der Zug so lange stehen, bis das Markerboard auf 1 gesetzt wird. In jedem Falle wird das Markerboard eines Track Elementes wieder auf 0 gesetzt, sobald ein Zug von diesem herunterfährt.
Wenn also ein Gleisabschnitt in eine Richtung belegt ist, wird zuerst geprüft, ob der Zug fahren darf(MB=1). Wenn er fahren darf, wird der benachbarte Gleisabschnitt der entsprechenden Richtung belegt und der vorher belegte Gleisabschnitt wird wieder frei. Im Folgenden zeige ich die 3 Hauptbewegungsarten von Zügen anhand eines Track Elementes, einer Weiche und einer Kreuzweiche auf. Dafür ist es hilfreich, die Karte des Gleisnetzes aus Abbildung 1 vor Augen zu haben.
Für Track Elemente gibt es immer genau zwei benachbarte Gleisabschnitte, einen in U2D Richtung und einen in D2U Richtung. Am Beispiel des Track Elementes 205 kann man sehen, dass in U2D Richtung die Weiche 14 und in D2U Richtung die Kreuzweiche 9 liegen. Da das Track Element 205 ein Startpunkt für Routen in U2D Richtung ist, muss das Marker Board dieses Track Elementes auf 1 stehen, bevor Züge auf die Weiche 14 fahren dürfen. Steht dieses auf 1, weil die Route vom Stellwerk gesperrt wurde,

kann der Zug die Weiche 14 befahren und das Marker Board wird wieder auf 0 gesetzt. In die D2U Richtung ist dieses Track Element kein Startpunkt einer Route. Also dürfen die Züge in diese Richtung zu jeder Zeit auf die Kreuzweiche 9 fahren. Beim Befahren auf die Kreuzweiche 9 wird dabei zusätzlich in der Variable p_9_STRAIGHT gespeichert, dass ein Zug nun von einem geraden Abschnitt aus die Weiche befährt und nicht von oben/unten kommt. Welchen Grund das hat, erläutere ich beim Beispiel der Kreuzweiche. Die Modellierung für das Track Element 205 wird in Listing 21 gezeigt

```
((te_205_U2D = 1) &
(te_205_MB = 1) &
set_te_205_MB_to_0 &
move_train_from_te_205_U2D &
no_route_change &
no_point_change)
|
((te_205_D2U = 1) &
no_mb_change &
move_train_from_te_205_D2U &
no_route_change &
set_p_9_straight_to_1)
```

Listing 21: Zugbewegung auf einem Track Element

Während es bei Track Elementen immer nur 2 Möglichkeiten gibt, auf welchen Abschnitt der Zug als nächstes fahren kann, gibt es bei Weichen je nach Position, in der sie gerade stehen, 3 Möglichkeiten wo der Zug lang fahren kann. Die Weiche 14 beispielsweise hat 3 benachbarte Gleisabschnitte: Track Element 200 am Stamm, Track Element 205 auf dem PLUS Ende und Track Element 206 auf dem MINUS Ende. Fährt ein Zug nun in U2D Richtung, fährt er als Nächstes immer auf das Track Element 200, egal ob er vom PLUS oder MINUS Ende kommt. In der entgegengesetzten Richtung D2U wiederum hängt es von der Position der Weiche ab, ob der Zug auf Track Element 205 oder 206 fährt. Für Weichen gibt es keine Markerboards wie bei Track Elementen, da es keine Route gibt, die auf einer Weiche startet. Außerdem gibt es, wie bereits eingangs erwähnt, in der echten Implementierung keine Möglichkeit, die Züge auf Weichen zu orten. Daher dürfen Züge, die auf einer Weiche stehen, immer fahren. Das Verhalten für Weiche 14 wird im Listing 22 aufgezeigt.

```
((p_14_U2D = 1) &
move_train_from_p_14_U2D &
no_route_change &
no_mb_change &
no_point_change)
|
((p_14_D2U = 1) &
```

```
move_train_from_p_14_D2U &
no_route_change &
no_mb_change &
no_point_change)
```

Listing 22: Zugbewegung auf einer Weiche

Weiche 9 wiederum repräsentiert eine Kreuzweiche und hat somit 4 benachbarte Gleisabschnitte. Bei einer Kreuzweiche ist dabei zu beachten, dass in PLUS Position ein Zug immer auf das gegenüberliegende Ende fährt, während in MINUS Position der Zug abbiegt. Da es in beide Richtungen aber 2 Möglichkeiten gibt, von wo der Zug kommen kann, fährt er je nachdem, von wo er kommt, auch auf einen anderen Gleisabschnitt, obwohl die Position der Weiche gleich ist. Dies kann man am Beispiel der Weiche 9 gut sehen. Ein Zug kann in U2D Richtung entweder von unten, also von Weiche 8 oder von rechts, also von Track Element 210 aus kommen. Steht die Weiche in PLUS Position und der Zug kommt von Weiche 8 aus, fährt er als Nächstes auf die Weiche 10. Kommt er jedoch von Track Element 210, fährt er als nächstes auf Track Element 205. In D2U Richtung verhält es sich genauso. Dementsprechend benötigt man für Kreuzweichen neben der Richtung des Zuges und der Position der Weiche zusätzlich noch die Information, von wo der Zug gekommen ist. Diese wird in der Variable STRAIGHT gespeichert. Im Falle von Weiche 9 wäre p_9_STRAIGHT auf 1 gesetzt, wenn ein Zug von Track Element 205 oder 210 kommt und auf 0, wenn ein Zug von Weiche 8 oder 10 aus kommt. Listing 23 zeigt die Modellierung für die Kreuzweiche 9.

```
((p_9_U2D = 1) &
move_train_from_p_9_U2D &
no_route_change &
no_mb_change &
no_point_change)
|
((p_9_D2U = 1) &
move_train_from_p_9_D2U &
no_route_change &
no_mb_change &
no_point_change)
```

Listing 23: Zugbewegung auf einer Kreuzweiche

4.4.10. Ein Zug fährt in das Gleisnetz ein

Einen Sonderfall bilden Zugbewegungen, die von außerhalb des betrachteten Gleisnetzes beginnen und in der nächsten Bewegung das Gleisnetz befahren. Dies ist bei Routen der Fall, die von einem Eintrittspunkt des Gleisnetzes aus zu einem Punkt innerhalb des Gleisnetzes führen. In diesem Fall wird, sobald die Route gesperrt wurde, ein Zug

am Eingangspunkt des Gleisnetzes, wo die entsprechende Route beginnt, „materialisiert“. Im Anschluss befährt der Zug wie oben beschrieben das Gleisnetz.
Listing 24 zeigt das Befahren eines Zuges in das Gleisnetz am Track Element 218e. Dabei wird geprüft, ob eine Route, die am Track Element 218e startet, auf LOCKED steht und der Streckenabschnitt von keinem Zug belegt ist. Im Beispiel startet nur die Route 88 am Track Element 218e. Wird die Bedingung erfüllt, wird das Track Element 218e belegt.

```
(((r_88_MODE = 4) &
r_88_tes_free &
r_88_points_free) &
enter_train_on_te_218e &
no_route_change &
no_mb_change &
no_point_change)
```

Listing 24: Einfahren eines Zuges in das Gleisnetz

4.5. Dekomposition des Gleisnetzes

Da das gesamte Gleisnetz zu groß ist, um es mit nuXmv zu modellieren, führe ich eine Dekomposition des Gleisnetzes in mehrere Teile aus. Dafür bediene ich mich der kompositionellen Verifikation durch sogenannte „Cuts“, mit denen ich das Gleisnetz an Punkten, die bestimmte Bedingungen erfüllen, zertrennen kann.
Im Folgenden werde ich die drei Hauptcuts, den Border Cut [MFH16], den Linear Cut [MFH17] und den Horizontal/General Cut [FHM17] beschreiben und erläutern, wie ich diese auf unser Gleisnetz angewandt habe. In diesem Zusammenhang ist es wichtig zu wissen, dass ein Up-Markerboard ein Markerboard ist, welches Züge in U2D-Richtung anhält und ein Down-Markerboard ein Markerboard, welches Züge in D2U-Richtung anhält. Ein Overlap bezeichnet Gleisabschnitte, die für Routen zusätzlich gesperrt werden, um das Fahren eines Zuges über ein Markerboard hinaus frühzeitig zu erkennen und einen Unfall zu vermeiden. Front Protection bezeichnet das Stellen von Weichen, um einen Frontalzusammenstoß von Zügen zu verhindern. Flank Protection bezeichnet das Stellen von Weichen, um das seitliche Einfahren eines Zuges in einen anderen zu vermeiden.

4.5.1. Border Cut

Der Border Cut ist der erste Ansatz zur Dekomposition des Gleisnetzes. Dieser versucht das Gleisnetz an Punkten zu zerteilen, an denen keine Route aus dem Stellwerk sich mit einer anderen Route überschneidet. Es gibt insgesamt 3 Bedingungen, die für

einen Border Cut erfüllt sein müssen:

1. Der Cut trennt einen linearen Abschnitt, sodass ein Up-Markerbord auf dem U2D-Ende und ein Down-Markerboard auf dem D2U-Ende liegen.
2. Es gibt keine Overlap Abschnitte aus Routen vom U2D-Ende für das D2U-Ende und andersherum.
3. Es gibt keine flank/front-Protection Requirements aus Routen vom U2D-Ende für das D2U-Ende und andersherum.

Die Abbildungen 7 und 8 zeigen dies in einem praktischen Beispiel.

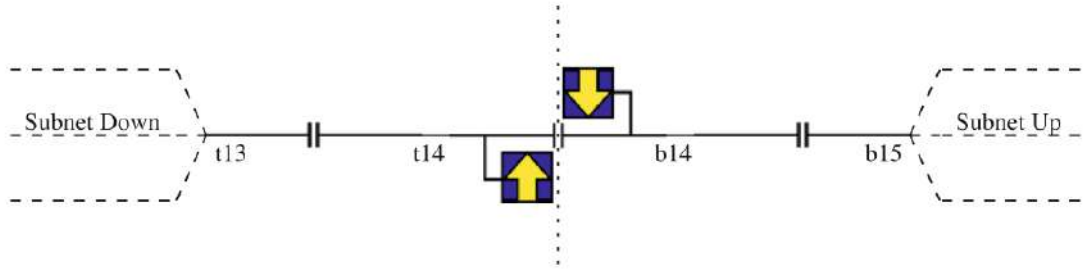

Abbildung 7: Border Cut Vorbedingung

Abbildung 8: Border Cut nachher

Wie man sehen kann, wird das Gleisnetz zwischen den Markerboards getrennt, sodass zwei Teilnetzwerke entstehen.

Um nun zu überprüfen, ob alle Bedingungen für das vorliegende Gleisnetz zutreffen, muss zunächst erst einmal aufgezeigt werden, welche Markerboards im Gleisnetz Up- und welche Down-Markerboards sind. Dies ist in Abbildung 9 zu sehen. Up-Markerboards werden darauf blau markiert und Down-Markerboards rot.

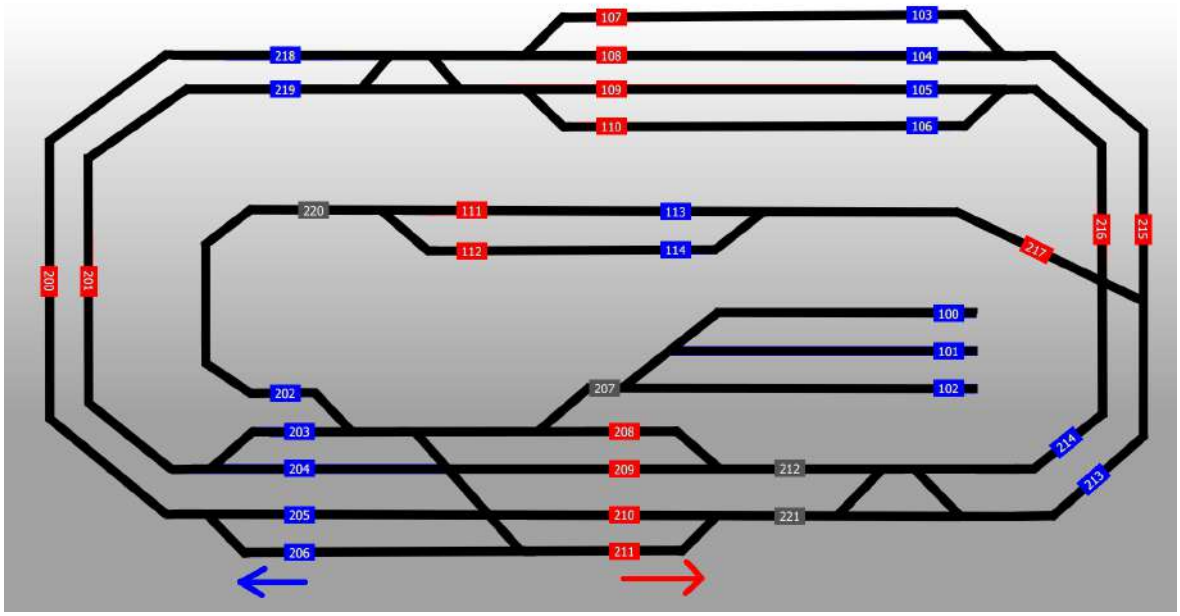

Abbildung 9: Markerboards im Gleisnetz

Die erste Bedingung sagt nun, dass man das Gleisnetz an Punkten trennen kann, bei denen Routen an einem linearen Abschnitt gegenüberliegend enden. In Abbildung 10 sind alle Punkte, an denen die Markerboards gegenüberliegen, markiert.

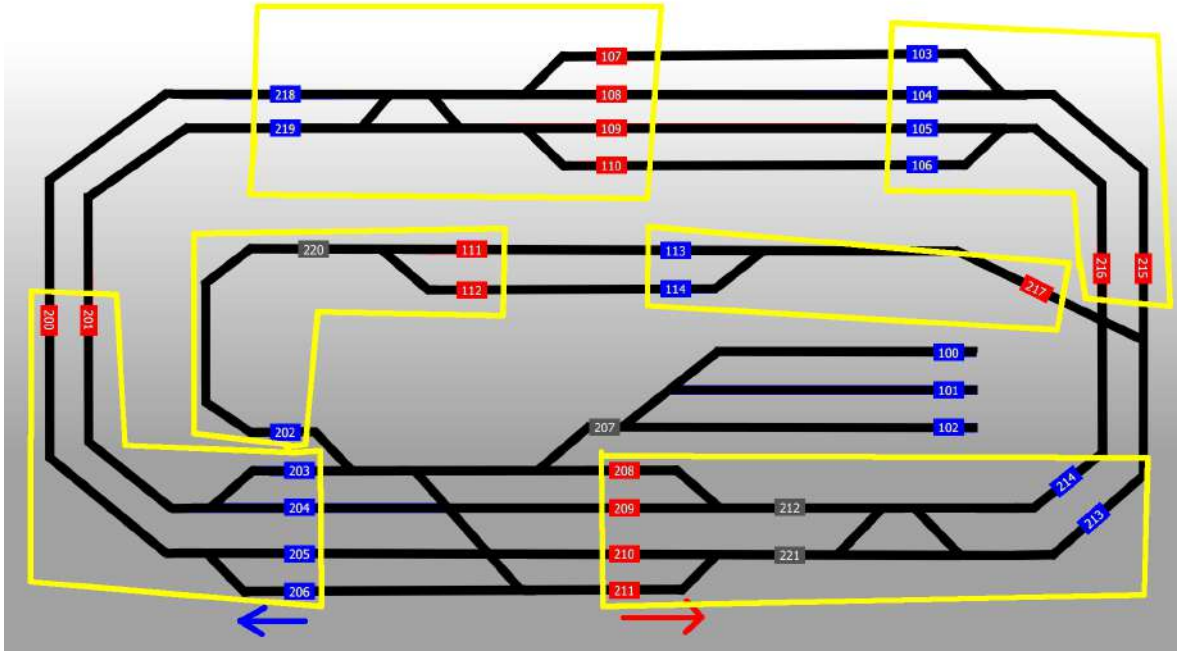

Abbildung 10: Markerboards im Gleisnetz die sich gegenüberstehen

Wie man in dieser Abbildung sehen kann, werden alle Markerboards die sich gegenüberliegen, durch mindestens eine Weiche getrennt. Dementsprechend handelt es sich in keinem Fall um einen linearen Abschnitt und bereits die erste Bedingung trifft auf

keinen Punkt im Gleisnetz zu.

4.5.2. Linear Cut

Der Linear Cut ist eine weitere Methode zur Dekomposition des Gleisnetzes und in seiner Art sehr ähnlich zum Border Cut. Allerdings wird beim Linear Cut das Gleisnetz an einem sogenannten „Interface“ getrennt, das am Ende in beiden Gleisnetzen vorkommt. Das Interface muss wie schon beim Border Cut ein linearer Abschnitt sein. In diesem Interface können sich Routen überschneiden, was der große Unterschied zum Border Cut ist. Auch beim Linear Cut gibt es 3 Bedingungen, die erfüllt sein müssen, um ihn anzuwenden:

1. Das Interface enthält ein Up-Markerboard in D2U-Richtung und ein Down-Markerboard in U2D-Richtung.
2. Die beiden neuen Teilnetzwerke haben nur das Interface gemeinsam.
3. Keine flank/front Protection Requirements für Routen aus dem Up-Netzwerk sind abhängig von Elementen aus dem Down-Netzwerk und andersherum, außer die Routen enden auf dem Interface.

Wie so etwas aussieht, zeigt das Beispiel in Abbildung 11 und 12.

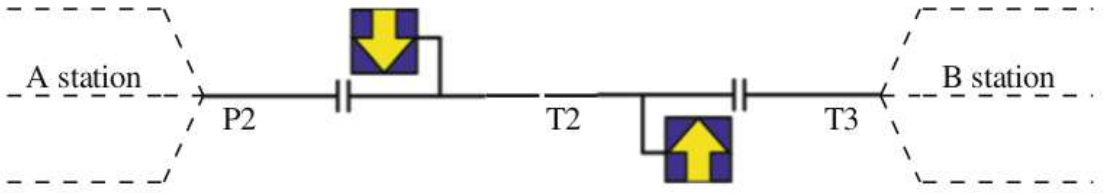

Abbildung 11: Linear Cut Vorbedingung

Abbildung 12: Linear Cut nachher

Wie man in den Abbildungen sieht, wird das Gleisnetz aufgeteilt in zwei Teilnetzwerke, die jeweils das Track Element T2 als Interface gemeinsam haben. Außerdem wird in jedem Teilnetzwerk ein Entry-Markerboard hinzugefügt, an dem Züge in das

Teilnetzwerk einfahren können.

Nun kann man sich das Gleisnetz aus Abbildung 9 erneut ansehen und die Bedingungen vom linear Cut prüfen. Anders als beim Border Cut müssen die Markerboards sich nun nicht gegenüberstehen, sondern voneinander wegführen, sodass zwischen diesen das benötigte Interface entsteht. Auch hier habe ich dafür alle Möglichkeiten in Abbildung 13 aufgezeigt.

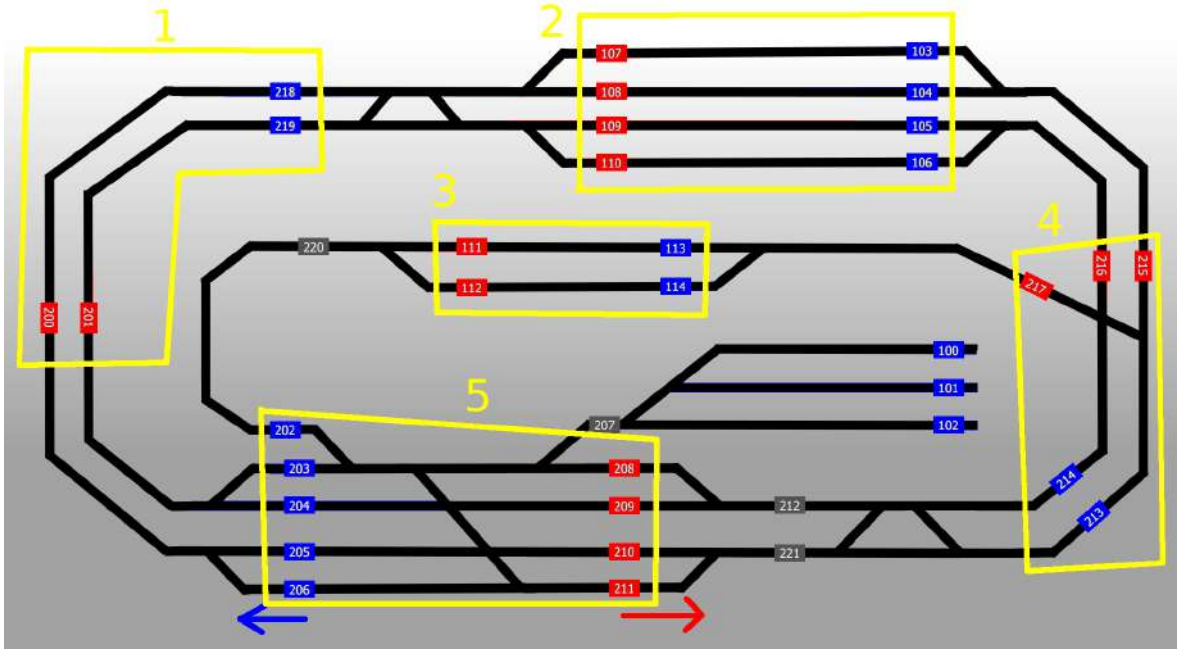

Abbildung 13: Markerboards im Gleisnetz die voneinander wegführen

Es muss jetzt geprüft werden, ob es sich bei den möglichen Interfaces um lineare Abschnitte handelt. Wie man in Abbildung 13 gut sehen kann, trifft dies auf die oberen Abschnitte 1-3 zu, wohingegen bei den anderen beiden Abschnitten 4 und 5 jeweils wieder mindestens eine Weiche zwischen den Markerboards liegt. Dementsprechend fallen diese beiden Abschnitte als mögliche Cut punkte raus. Für die oberen 3 Abschnitte müssen jetzt noch die Bedingungen für den linear Cut geprüft werden. Die erste Bedingung ist für alle Abschnitte erfüllt, da wir danach ja bereits unsere Abschnitte ausgewählt haben. Die zweite Bedingung ist ebenfalls erfüllt, da wir an die Teilnetzwerke jeweils das entsprechende Interface dranhängen können. Und die dritte Bedingung ist auch erfüllt, da es keine Routen gibt, die eine flank/front Protection im anderen Gleisnetz vornehmen und nicht auf dem Interface halten. Somit können wir insgesamt 3 Linear Cuts in unserem Gleisnetz setzen.

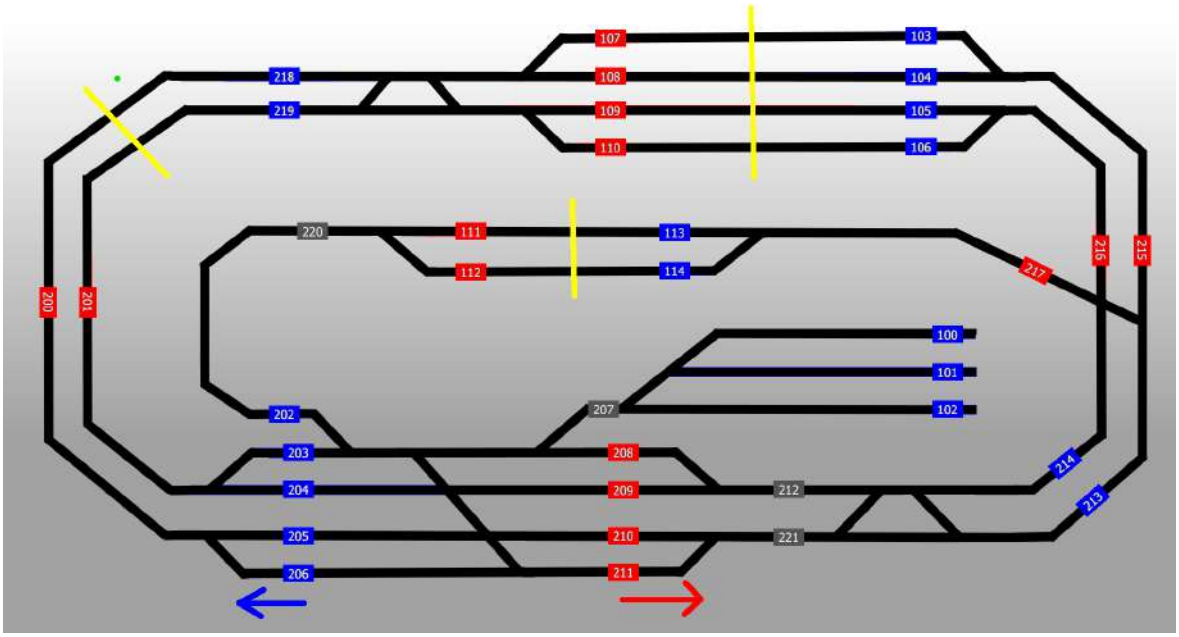

Abbildung 14: Linear Cuts im Gleisnetz

Wie man in Abbildung 14 sehen kann, haben wir nun einen relativ kleinen Abschnitt im oberen Teil des Gleisnetzes, allerdings noch einen sehr großen restlichen Abschnitt. Somit können wir nun einen Abschnitt mit nuXmv verifizieren, haben jedoch noch einen zweiten Abschnitt, der immer noch zu groß ist und zu einer State Explosion führt. Dementsprechend bedarf es weiterer Optionen für eine Dekomposition des Gleisnetzes.

4.5.3. Horizontal Cut

Eine letzte Variante zur Dekomposition des Gleisnetzes bietet der Horizontal Cut. Im Gegensatz zu den ersten beiden Varianten teilt dieser das Gleisnetz nicht vertikal, sondern, wie der Name schon sagt, horizontal. Somit findet eine Dekomposition bei diesem Cut immer an Weichenübergängen statt. Ein großer Vorteil dieser Variante gegenüber den anderen beiden Varianten ist, dass es keine Bedingung für diesen Cut gibt. Dazu sei jedoch gesagt, dass zu dieser Dekompositionsmethode immer noch geforscht wird. Daher muss noch bewiesen werden, ob aus der Verifikation der Teilnetzwerke auch die Verifikation des Gesamtnetzes folgt und ob es wirklich keine Bedingungen für diesen Cut gibt. In den Abbildungen 15 und 16 wird der Cut an einem Beispiel gezeigt.

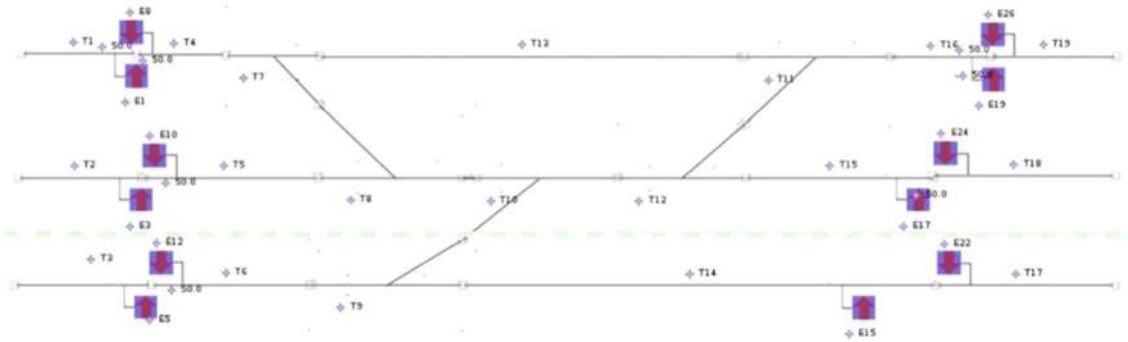

Abbildung 15: Horizontal Cut vorher

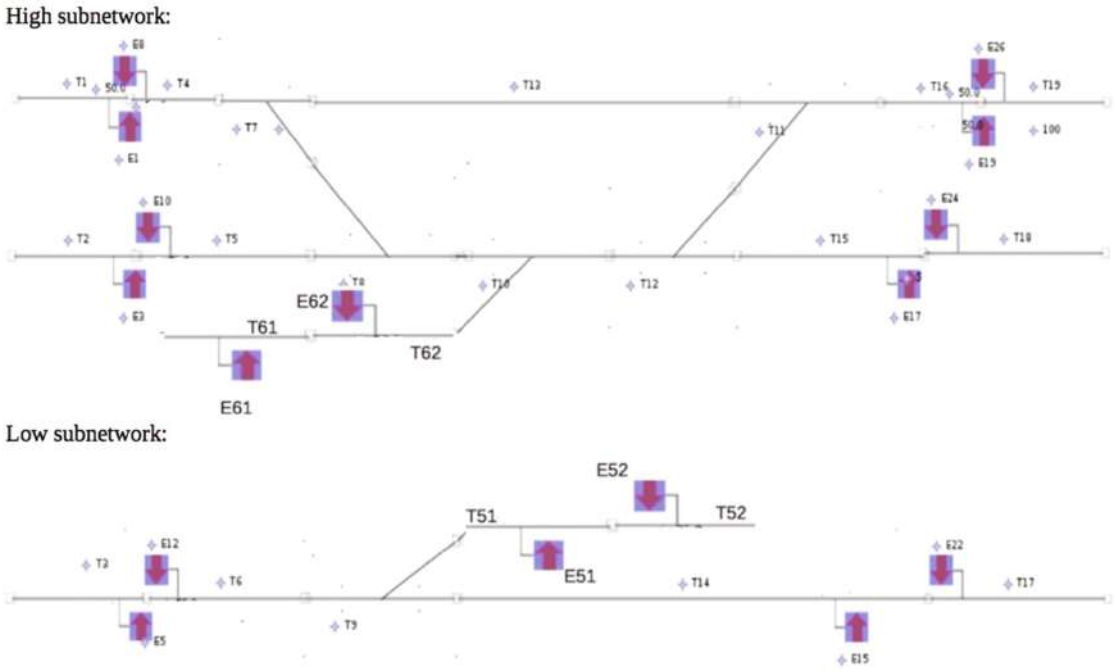

Abbildung 16: Horizontal Cut nachher

An diesem Beispiel kann man sehr schön sehen, dass der Cut an einem Weichenübergang erfolgt. An diesem Übergang wird dann in den Teilnetzwerken ein neuer linearer Abschnitt mit jeweils zwei Markerboards eingebaut, eines in up und eines in down Richtung. Diese bilden dann jeweils den Anfang bzw. das Ende der Routen, die in dieses Teilnetzwerk ein bzw. ausfahren. Bei einem Horizontal Cut können außerdem die Routen für die Teilnetzwerke in Äquivalenzklassen eingeteilt werden. Denn Routen, die in dem gesamten Gleisnetz vorher unterschiedlich waren, können in dem neuen Gleisnetz nun äquivalent sein, da wir im Gegensatz zu den vorherigen Cuts auch durch Routen hindurch einen Cut setzen können.

In unserem Gleisnetz können wir nun erneut gucken, wo wir Horizontal Cuts setzen können. Da es keine Bedingungen gibt, die dafür erfüllt sein müssen, kann dieser an

jedem Weichenübergang gesetzt werden. In der Abbildung 17 zeige ich auf, wo ich die Horizontal Cuts gesetzt habe. Die vorherigen Linear Cuts sind dabei als gelbe Trennlinien und die Horizontal Cuts als orange Trennlinien gekennzeichnet.

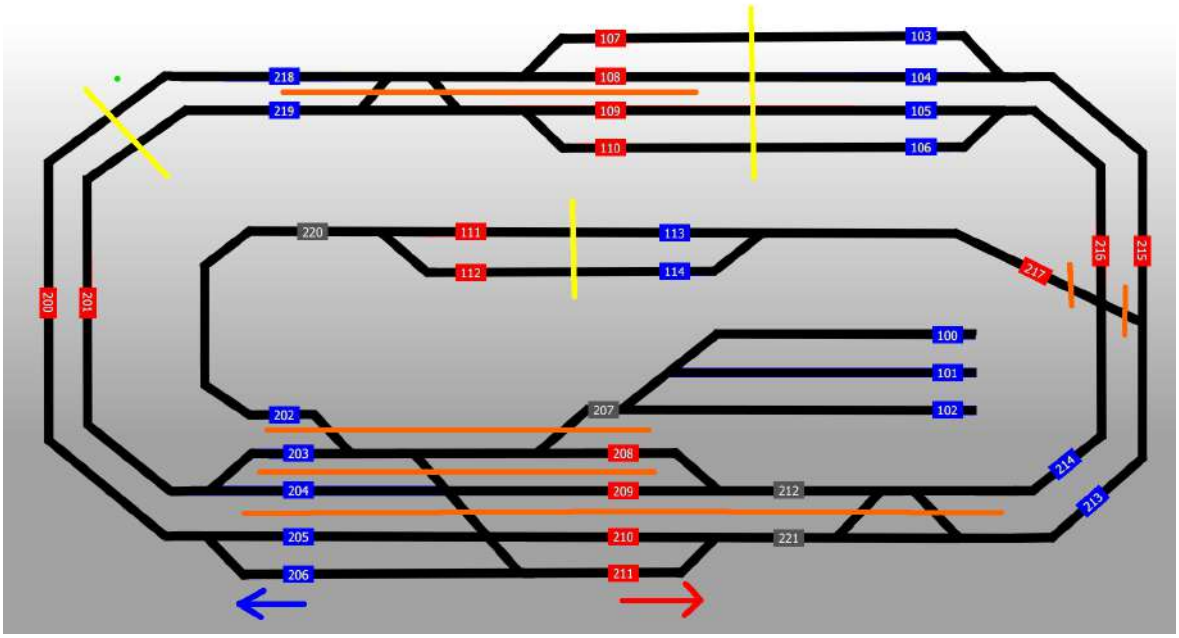

Abbildung 17: Horizontal Cuts im Gleisnetz

In Appendix A werden die insgesamt 8 daraus resultierenden Teilnetzwerke aufgelistet.

5. Verifikation der Modellierung

Um sicherzustellen, dass das Verhalten meines Modells der echten Implementierung des Stellwerkes entspricht, führe ich sogenannte Model-in-the-Loop Tests durch. Dabei überprüfe ich, ob folgende Eigenschaften auf mein Modell zutreffen:

1. Jede Route wird irgendwann einmal von einem Zug befahren.
2. Jedes Track Element und jede Weiche wird irgendwann einmal belegt.
3. Jede Weiche wird irgendwann einmal geschaltet.
4. Wenn eine Weiche nicht gestellt wird, wird ein Notstopp ausgelöst oder es kommt zu einem Crash.
5. Wenn eine Konfliktroute befahren wird, kommt es zu einem Crash.

Die ersten drei Bedingungen überprüfen, ob jeder Zustand einer Variablen mindestens einmal erreicht wird. Wäre das nicht der Fall, würden einige meiner definierten Transitionen nie genommen werden und das Modell wäre folglich fehlerhaft. Diese Eigenschaften überprüfe ich mithilfe von CTL Spezifikationen, die in Listing 25 ausschnittsweise aufgezeigt werden.

```
CTLSPEC
        EF (r_3_MODE = 5) &
        EF (r_4_MODE = 5) &

CTLSPEC
        EF ((te_107e_U2D + te_107e_D2U) = 1) &
        EF ((te_107_U2D + te_107_D2U) = 1) &

CTLSPEC
        EF ((p_1_U2D + p_1_D2U) = 1) &
        EF (p_1_POS = 1) &
```

Listing 25: MiL Tests

Die letzten beiden Bedingungen prüfen, ob mein Modell bei einem Fehler z.B. in der Interlockingtable auch wirklich einen Fehler erkennt und nicht ein fehlerhaftes System als sicher bezeichnet. Dafür werden separate Tests ausgeführt, in denen ich Einträge in meiner Weichen/Konfliktmatrix, welche die Informationen aus der Interlockingtable beinhalten, manipuliere.

Die Model-in-the-Loop Tests wurden für 6 der 8 Teilnetzwerke ausgeführt und weisen alle das erwartete Verhalten auf. Die Tests für die Teilnetzwerke 7 und 8 haben leider trotz der Dekomposition einen extremen Arbeitsspeicherbedarf, sodass ich die Tests für diese Netzwerke nicht verifizieren kann.

6. Evaluation

Die Verifikation des Stellwerkes erfolgt über die Verifikation der 8 Teilnetzwerke, die aus der Dekomposition des Gleisnetzes entstanden sind. Wenn gezeigt werden kann, dass alle 8 Teilnetzwerke mit den vorhandenen Routen und dem Stellwerk unfallfrei befahren werden, folgt dies auch für das gesamte Gleisnetz. Es wurden dabei folgende Eigenschaften verifiziert:

1. Es befinden sich zu keiner Zeit 2 Züge auf einem Track Element oder einer Weiche.
2. Es wird zu keiner Zeit eine Weiche gestellt, wenn sich ein Zug darauf befindet.
3. Es fährt zu keiner Zeit ein Zug von einem PLUS/MINUS Ende einer Weiche auf den Stamm, wenn die Weiche auf MINUS/PLUS gestellt ist.
4. Es liegt zu keiner Zeit ein Safety Error vor.

Eigenschaft 1 soll dabei zeigen, dass Züge zu keinem Zeitpunkt kollidieren können. Die Eigenschaften 2 und 3 zeigen auf, dass es zu keinem Entgleisen eines Zuges kommt und Eigenschaft 4 besagt, dass zu keiner Zeit ein Safety Error vorliegt, da dies nur der Fall sein sollte, wenn Weichen defekt sind oder sich Geisterzüge auf dem Gleisnetz befinden, was ich in meiner Modellierung jedoch nicht berücksichtige. Alle Eigenschaften werden mithilfe von LTL Formeln verifiziert, die ausschnittsweise in den Listings 26 bis 29 aufgezeigt werden.

```
LTLSPEC
        G !(te_107e_U2D + te_107e_D2U >= 2) &
        G !(te_107_U2D + te_107_D2U >= 2) &
```

Listing 26: Verifikation: Es befinden sich zu keiner Zeit 2 Züge auf einem Track Element oder einer Weiche

```
LTLSPEC
        G !((p_1_POS != next(p_1_POS)) & (p_1_U2D + p_1_D2U > 0)) &
        G !((p_4_POS != next(p_4_POS)) & (p_4_U2D + p_4_D2U > 0)) &
```

Listing 27: Verifikation: Es wird zu keiner Zeit eine Weiche gestellt, wenn sich ein Zug darauf befindet

```
LTLSPEC
        G !((p_4_D2U = 1) & (next(p_4_D2U) = 0) & (next(p_1_D2U) = 1) & (
          p_1_POS = 1)) &
        G !((te_1te_D2U = 1) & (next(te_1te_D2U) = 0) & (next(p_1_D2U) =
          1) & (p_1_POS = 0)) &
```

Listing 28: Verifikation: Es fährt zu keiner Zeit ein Zug von einem PLUS/MINUS Ende einer Weiche auf den Stamm, wenn die Weiche auf MINUS/PLUS gestellt ist

```
LTLSPEC
        G !safety_error;
```

Listing 29: Verifikation: Es liegt zu keiner Zeit ein Safety Error vor

Diese Eigenschaften wurden für 6 der 8 Teilnetzwerke geprüft. Tabelle 6 zeigt dabei den zeitlichen Aufwand und den benötigten Arbeitsspeicher für alle Gleisnetze. Die Daten wurden mithilfe des Programms „runlim“ ausgewertet. Für die Teilnetzwerke 7 und 8 habe ich leider keine Daten, da diese für die Auswertung einen enormen Arbeitsspeicherbedarf besitzen und daher nicht terminieren. Alle Tests liefen mit nuXmv-Version 1.1.1 auf einem Computer mit Intel(R) Core(TM) i5-7200 CPU @ 2.50GHz, 8 GB Arbeitsspeicher und Ubuntu 16.04 LTS. Die Teilnetzwerke 7 und 8 liefen zusätzlich auf einem Cloud Server von Google Cloud Platform(GCP) mit Intel(R) Xeon(R) CPU @ 2.20GHz, 102 GB Arbeitsspeicher und Debian 9. Doch auch diese Menge an Arbeitsspeicher war nicht genug, um ein Ergebnis zu erzielen.

Gleisnetz	Dauer(in Sekunden)	Arbeitsspeicher(in MB)
1	774.99	1775.2
2	705.60	1992.2
3	6.12	131.1
4	7.26	135.9
5	2.20	76.9
6	717.71	1357.9
7	-	-
8	-	-

Tabelle 6: Ausführungszeiten und Arbeitsspeicherverbrauch von nuXmv

Dabei kann man sehen, dass die Teilnetzwerke 1 bis 6, die in ihrer Größe sehr begrenzt sind, sich in relativ kurzer Zeit berechnen lassen und vergleichsweise wenig Arbeitsspeicher benötigen. Man kann jedoch auch sehen, dass schon kleine Änderungen in der Größe der Gleisnetze (Vergleich Gleisnetz 2 und 3) große Auswirkungen auf die Ausführungszeit und den Arbeitsspeicherbedarf haben.

Trotz dessen, dass die Gleisnetze 7 und 8 nicht ausgewertet werden konnten, konnte bereits ein kleiner Fehler in der Interlockingtable festgestellt werden. Beim Sperren der Routen 38 und 40 von Track Element 100 bzw. 101 nach 203 (bzw. 7te im Gleisnetz 5) wird die Weiche 27 falsch gestellt, sodass diese auf PLUS steht, obwohl der Zug vom MINUS Ende der Weiche kommt und andersherum bei Route 40. Dies ist in der Praxis bisher nicht aufgefallen, weil dort die Züge die Weichen einfach zur Seite drücken können. Trotzdem kann dies ein Sicherheitsrisiko darstellen und sollte behoben werden.

Bei der Auswertung des Modells musste festgestellt werden, dass die Modellierung eines Stellwerkes nuXmv vor große Herausforderungen stellt. Der exponentielle An-

stieg von Auswertungsdauer und Speicherbedarf bei schon geringer Vergrößerung des Gleisnetzes setzt eine starke Zerteilung der Modellierung voraus. Beim Versuch, das Gleisnetz als Ganzes zu verifizieren, lief nuXmv bereits mehrere Wochen auf einem Server, ohne jedoch zu terminieren. Dies deckt sich mit den Ergebnissen aus dem neuen dänischen Stellwerk, in dem die Performance von verschiedene Verfahren zum Testen von Invarianten in nuXmv einer eigens entwickelten Toolchain gegenübergestellt wurden. [HHP17] Dort sind Verfahren, die kein Bounded-Model-Checking(BMC) oder IC3 verwendet haben, schon bei kleinen Gleisnetzen nicht mehr terminiert und auch die Verfahren mit BMC und IC3 hatten einen deutlich höheren Arbeitsspeicherverbrauch als die eigens entwickelte Toolchain.

7. Related Work

Parallel zu meiner Bachelorarbeit wurde von Felix Brüning eine Arbeit mit ähnlichem Thema geschrieben. Wir beide beschäftigten uns mit dem Model-Checking des von uns vom Wintersemester 2018/19 bis Sommersemester 2019 entwickelten autonomen Bahnsystems. Dabei benutzte ich das Tool nuXmv von der Fondazione Bruno Kessler. Felix hingegen nutzte das Tool FDR4 von der Oxford University [TGR14] [TGR13]. 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.

FDR4 FDR4 ist kein gewöhnlicher Model-Checker, sondern ein sogenannter Refinement-Checker. Es wird, im Vergleich zum konventionellen Model-Checking, nicht global geprüft, ob eine LTL-Formel eine Bestimmte Spezifikation erfüllt, sondern es wird geprüft, ob eine bestimmte Folge von Signalen (Trace) in einem Prozess vorhanden sind. FDR4 nutzt dafür die Prozessalgebra *Communicating Sequential Processes* (kurz: CSP), die erstmals 1985 von C.A.R. Hoare vorgestellt wurde. Dabei nutzt FDR4 jedoch die maschinenlesbare Version CSP_M, die eine Erweiterung von CSP ist. Diese legt den Schwerpunkt auf die Kommunikation zwischen Prozessen, mit der über sogenannte Channels (Kanäle) Daten und Signale zwischen einzelnen Prozessen versendet und empfangen werden können. Prozesse können sich dabei auch auf bestimmte Signale von Kanälen synchronisieren. Die Syntax von CSP ähnelt dabei stark einer funktionalen Programmiersprache. CSP sowie CSP_M beschreibt das Verhalten des Systems und reagiert auf Signale von „außen“, die durch Kanäle empfangen bzw. verwendet werden. Ein einfaches Beispiel ist der Prozess $P = c \rightarrow STOP$, der zunächst das Event c erzeugt und dann beendet.
FDR4 prüft, ob ein Prozess einen anderen verfeinert im Bezug zu Trace-, Fehler- oder Fehlerabweichungsverfeinerung. 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 30: CSPm Beispiel

Mit Listing 30 soll nun getestet werden, ob der Prozess P den Trace $b \rightarrow c$ imple-

mentiert (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.

nuXmv Bei nuXmv handelt es sich um einen Symbolic Model-Checker, der aus dem Model-Checker nuSMV hervorgegangen ist. Bei Symbolic Model-Checkern wird der Zustandsraum durch boolesche 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.
Listing 31 zeigt dabei ein Beispielprogramm, 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 31: 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 Felix und ich konnten beide nachweisen, dass es in der Interlockingtable noch einige Fehler gab. 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 oder, wenn sich auf der anderen Seite der Weiche ein Zug befindet, zu einem Zusammenstoß der Züge führen können. Diese Fehler wurden von uns im Nachhinein verbessert, sodass diese Sicherheitsverletzungen nicht mehr auftreten können. In meiner Arbeit wurden dabei nur die Routen 38 und 40 als falsch erkannt. Dies hat den Grund, dass ich Routen durch die Dekomposition des Horizontal Cuts in Äquivalenzklassen zusammenfassen kann. Route 39 war damit äquivalent zu Route 40 und Route 78 äquivalent zu Route 44 im Gleisnetz 5.
Ein Fehler, den Felix nur mit FDR4 gefunden hat, 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.

8. Fazit und Ausblick

Es wurde festgestellt, dass bei einigen Routen eine Weiche falsch gestellt wird, was unter Umständen zum Entgleisen eines Zuges oder sogar einer Kollision zwischen zwei Zügen führen kann. Dieser Fehler wurde jedoch inzwischen behoben und stellt kein Sicherheitsrisiko mehr dar. Ansonsten konnten keine weiteren Mängel beim Stellwerk festgestellt werden.
Die Verifikation hat sich dabei als schwierig in dem Sinne herausgestellt, dass nuXmv mit der Größe des Modells bzw. des Stellwerkes und des Gleisnetzes große Probleme hatte und in eine State Explosion lief. Daher musste die Arbeit um eine Dekomposition des Gleisnetzes mithilfe von Cuts erweitert werden, um den Zustandsraum genügend zu verringern und damit handhabbar für nuXmv zu sein. Dabei musste jedoch festgestellt werden, dass schon kleine Unterschiede in der Größe des Gleisnetzes große Auswirkungen auf die Ausführungszeit und den Arbeitsspeicherbedarf haben. Dadurch hat bei 2 Gleisnetzen die Dekomposition nicht ausgereicht, wodurch diese trotzdem nicht verifiziert werden konnten.
Aus den Ergebnissen dieser Arbeit lässt sich folgern, dass für das Model-Checking von Bahnanwendungen noch viel Spielraum nach oben vorhanden ist. Die Mittel zur Dekomposition des Gleisnetzes haben sehr geholfen, um zumindest einen Teil des Streckennetzes zu verifizieren. Allerdings gibt es in diesem Beispiel nur sehr wenig bis gar keine Möglichkeiten, den Linear- bzw. Border Cut anzuwenden. Der Horizontal Cut bietet zwar viele Möglichkeiten das Gleisnetz in der Höhe zu zerteilen, allerdings bleibt bei zwei Gleisabschnitten die Länge des Gleisnetzes ein Problem, was sich nicht mit dem Horizontal Cut lösen lässt. Daher wäre es durchaus interessant, nach weiteren Möglichkeiten wie dem Border- bzw. Linear Cut zu suchen, die das Gleisnetz auch vertikal teilen, jedoch mit weniger Bedingungen einhergehen.
Außerdem wurde festgestellt, dass der Horizontal Cut bei den Routen eine zu starke Abstrahierung vornimmt. Durch die Bildung von Äquivalenzklassen in den Routen werden nicht mehr alle Routen betrachtet, wodurch es vorkommen kann, dass Fehler in der Interlockingtable nicht gefunden werden. Dies zeigt, dass die Verifikation bei der Dekomposition eines Gleisnetzes durch den Horizontal Cut nicht zwangsläufig zum selben Ergebnis führt wie die Verifikation des gesamten Gleisnetzes.
Weiterhin wäre es für Folgearbeiten sinnvoll, einen Modell-Generator für nuXmv zu entwickeln, mit dem sich beliebige Gleisnetze anhand der Interlockingtable generieren lassen. Dies würde die Erstellung von Modellen deutlich vereinfachen. Die Verifizierung des Modells würde dann durch die Verifizierung des Generators stattfinden und man bräuchte keine eigenen MiL-Tests mehr für die einzelnen Modelle durchführen.

A. Gleisnetze aus der Dekomposition

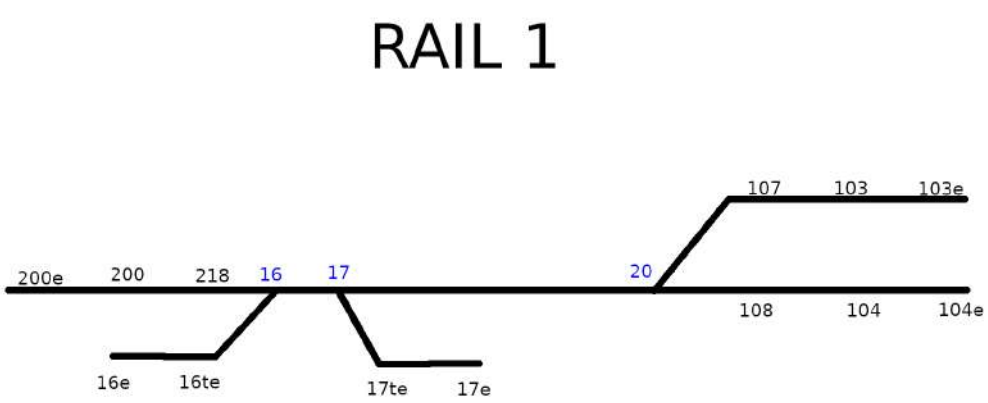

Abbildung 18: Rail 1

ID	SRC	DST	PATH	CONFLICTS
16	200e	218	200e,200,218	88,90,92
17	218	103	218,107,103	18,19,21,22,84,88,89,90,91,92
18	218	104	218,108,104	17,19,21,22,85,88,89,90,91,92
19	218	17te	218,17te	17,18,21,22,88,89,90,91,92
21	16e	103	16e,16te,107,103	17,18,19,22,84,88,89,90,91,92
22	16e	104	16e,16te,108,104	17,18,19,21,85,88,89,90,91,92
84	103e	107	103e,103,107	17,21,85
85	104e	108	104e,104,108	18,22,84
88	107	200	107,218,200	16,17,18,19,21,22,89,90,91,92
89	107	16te	107,16te	17,18,19,21,22,88,90,91,92
90	108	200	108,218,200	16,17,18,19,21,22,88,89,91,92
91	108	16te	108,16te	17,18,19,21,22,88,89,90,92
92	17e	200	17e,17te,218,200	16,17,18,19,21,22,88,89,90,91

Tabelle 7: Rail 1 Routen

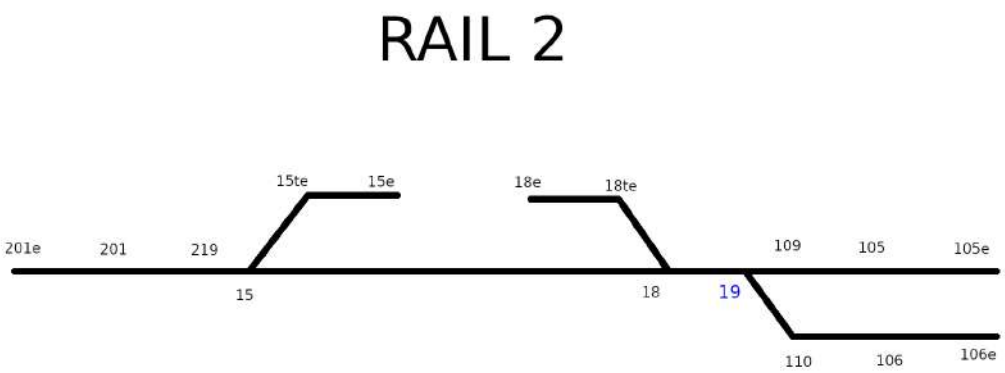

Abbildung 19: Rail 2

ID	SRC	DST	PATH	CONFLICTS
14	201e	219	201e,201,219	91,93,95
19	18e	105	18e,18te,109,105	20,22,23,24,86,91,92,93,94,95
20	18e	106	18e,18te,110,106	19,22,23,24,87,91,92,93,94,95
22	219	15te	219,15te	19,20,23,24,91,92,93,94,95
23	219	105	219,109,105	19,20,22,24,86,91,92,93,94,95
24	219	106	219,110,106	19,20,22,23,87,91,92,93,94,95
86	105e	109	105e,105,109	19,23,87
87	106e	110	106e,106,110	20,24,86
91	15e	201	15e,15te,219,201	14,19,20,22,23,24,92,93,94,95
92	109	18te	109,18te	19,20,22,23,24,91,93,94,95
93	109	201	109,219,201	14,19,20,22,23,24,91,92,94,95
94	110	18te	110,18te	19,20,22,23,24,91,92,93,95
95	110	201	110,219,201	14,19,20,22,23,24,91,92,93,94

Tabelle 8: Rail 2 Routen

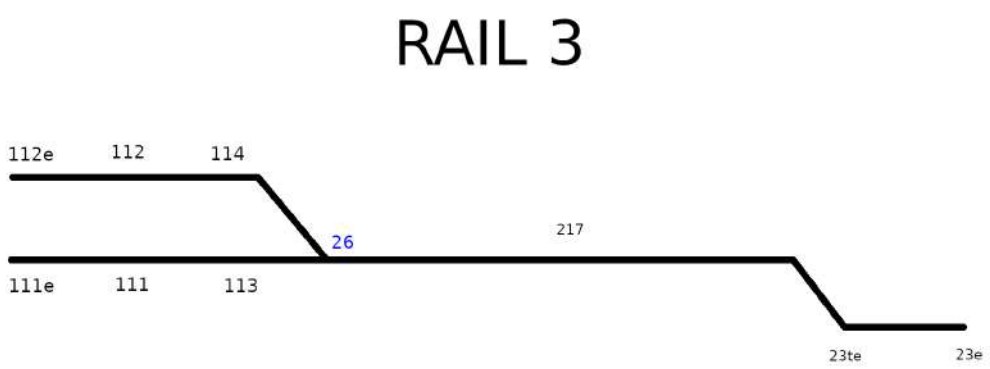

Abbildung 20: Rail 3

ID	SRC	DST	PATH	CONFLICTS
31	111e	113	111e,111,113	32,68
32	112e	114	112e,112,114	31,69
33	113	23te	113,217,23te	35,67,68,69
35	114	23te	114,217,23te	33,67,68,69
67	23e	217	23e,23te,217	33,35
68	217	111	217,113,111	31,33,35,69
69	217	112	217,114,112	32,33,35,68

Tabelle 9: Rail 3 Routen

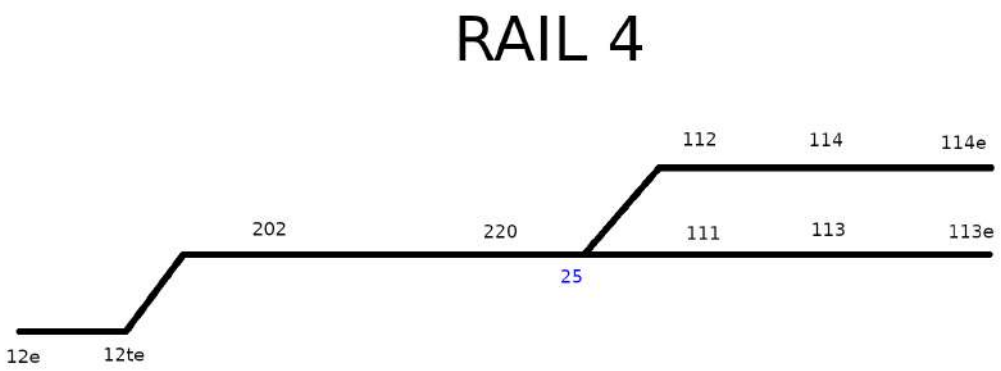

Abbildung 21: Rail 4

ID	SRC	DST	PATH	CONFLICTS
31	202	113	202,220,111,113	32,68,70,77
32	202	114	202,220,112,114	31,69,70,77
39	12e	202	12e,12te,202	70,77
68	113e	111	113e,113,111	31,69
69	114e	112	114e,114,112	32,68
70	111	12te	111,220,202,12te	31,32,39,77
77	112	12te	112,220,202,12te	31,32,39,70

Tabelle 10: Rail 4 Routen

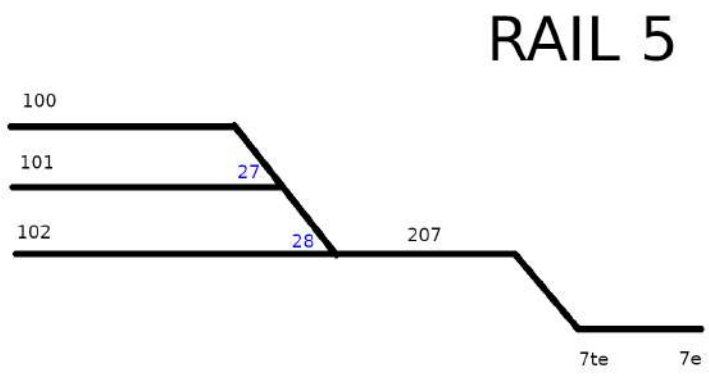

Abbildung 22: Rail 5

ID	SRC	DST	PATH	CONFLICTS
38	100	7te	100,207,7te	40,42,43,44,45
40	101	7te	101,207,7te	38,42,43,44,45
42	102	7te	102,207,7te	38,40,43,44,45
43	7e	100	7e,7te,207,100	38,40,42,44,45
44	7e	101	7e,7te,207,101	38,40,42,43,45
45	7e	102	7e,7te,207,102	38,40,42,43,45

Tabelle 11: Rail 5 Routen

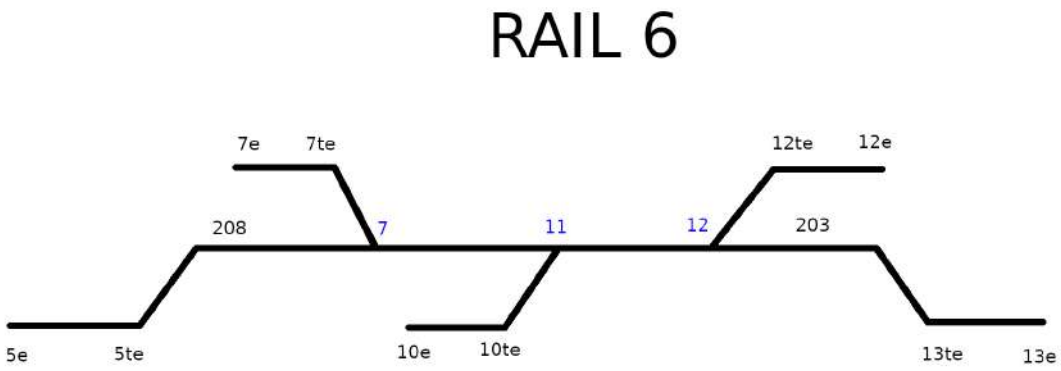

Abbildung 23: Rail 6

ID	SRC	DST	PATH	CONFLICTS
5	5e	12te	5e,5te,208,12te	6,8,9,37,38,45,46,52,72,73,74
6	5e	203	5e,5te,208,203	5,8,9,37,38,45,46,52,72,73,74
8	10e	12te	10e,10te,12te	5,6,9,37,38,45,46,72,73,74
9	10e	203	10e,10te,203	5,6,8,37,38,45,46,72,73,74
13	203	13te	203,13te	45,46
37	7e	12te	7e,7te,12te	5,6,8,9,38,45,46,72,73,74
38	7e	203	7e,7te,203	5,6,8,9,37,45,46,72,73,74
45	13e	7te	13e,13te,203,7te	5,6,8,9,13,37,38,46,72,73,74
46	13e	208	13e,13te,203,208	5,6,8,9,13,37,38,45,72,73,74
52	208	5te	208,5te	5,6
72	12e	7te	12e,12te,7te	5,6,8,9,37,38,45,46,73,74
73	12e	208	12e,12te,208	5,6,8,9,37,38,45,46,72,74
74	12e	10te	12e,12te,10te	5,6,8,9,37,38,45,46,72,73

Tabelle 12: Rail 6 Routen

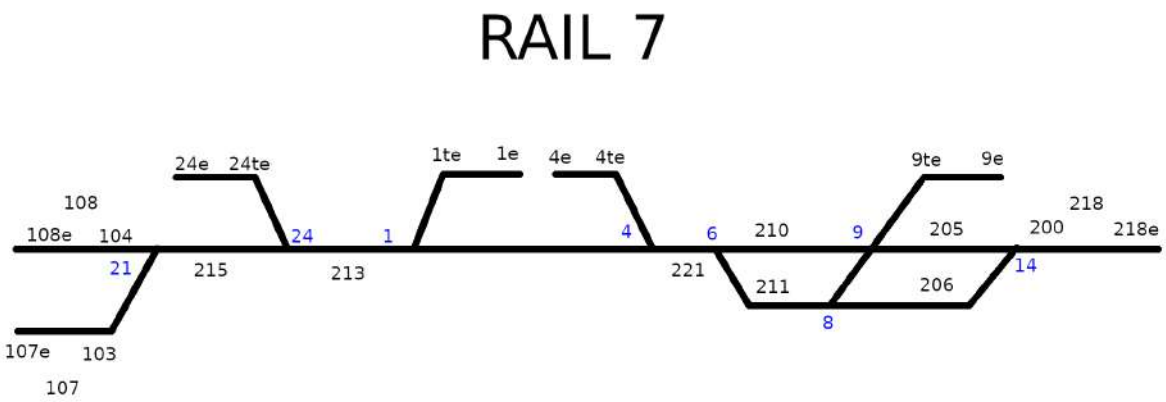

Abbildung 24: Rail 7

ID	SRC	DST	PATH	CONFLICTS
3	4e	205	4e,4te,221,210,205	4,5,10,11,12,48,49,50,57,58,59,62,63,64,66
4	4e	206	4e,4te,221,211,206	3,5,10,11,12,49,51,57,58,59,62,63,64,66
5	213	1te	213,1te	3,4,10,11,12,57,58,59,62,63,64,66
10	213	9te	213,221,210,9te	3,4,5,11,12,48,49,50,57,58,59,62,63,64,66
11	213	205	213,221,210,205	3,4,5,10,12,48,49,50,57,58,59,62,63,64,66
12	213	206	213,221,211,206	3,4,5,10,11,49,51,57,58,59,62,63,64,66
15	205	218	205,200,218	16,50,51,88
16	206	218	206,200,218	15,50,51,88
17	107e	103	107e,107,103	18,84,88
18	108e	104	108e,108,104	17,88
25	103	213	103,215,213	26,27,57,58,59,63,64,84
26	104	213	104,215,213	25,27,57,58,59,63,64,84
27	24e	213	24e,24te,213	25,26,57,58,59,62,63,64,66
48	9e	210	9e,9te,210	3,10,11,49,50
49	9e	211	9e,9te,211	3,4,10,11,12,48,50,51
50	200	210	200,205,210	3,10,11,15,16,48,49,51
51	200	211	200,206,211	4,12,15,16,49,50
57	1e	215	1e,1te,213,215	3,4,5,10,11,12,25,26,27,58,59,62,63,64,66
58	210	215	210,221,213,215	3,4,5,10,11,12,25,26,27,57,59,62,63,64,66
59	210	24te	210,221,213,24te	3,4,5,10,11,12,25,26,27,57,58,62,63,64,66
62	210	4te	210,221,4te	3,4,5,10,11,12,27,57,58,59,63,64,66
63	211	215	211,221,213,215	3,4,5,10,11,12,25,26,27,57,58,59,62,64,66
64	211	24te	211,221,213,24te	3,4,5,10,11,12,25,26,27,57,58,59,62,63,66
66	211	4te	211,221,4te	3,4,5,10,11,12,27,57,58,59,62,63,64
84	215	107	215,103,107	17,25,26,85
85	215	108	215,104,108	18,25,26,84
88	218e	200	218e,218,200	15,16,17,18

Tabelle 13: Rail 7 Routen

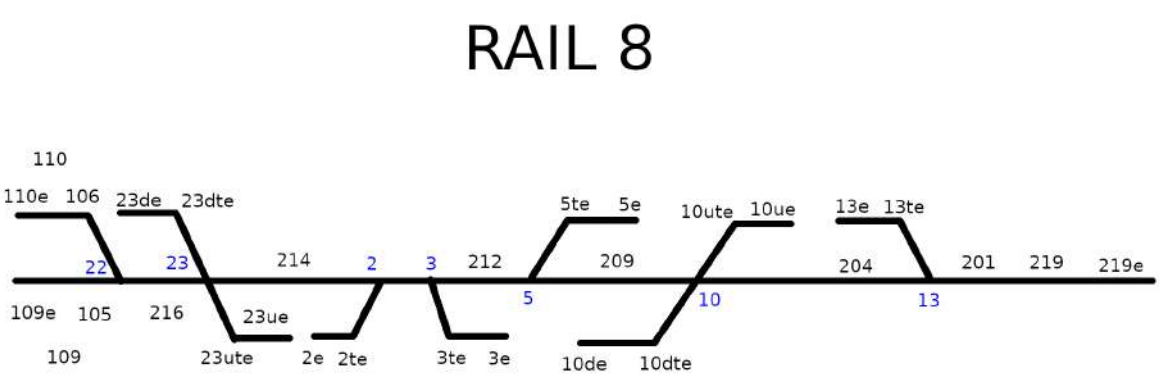

Abbildung 25: Rail 8

ID	SRC	DST	PATH	CONFLICTS
1	214	5te	214,212,5te	2,3,6,7,46,52,53,54,55,56,57,61,62,74
2	214	204	214,212,209,204	1,3,6,7,10,47,48,52,53,54,55,56,57,61,62,74
3	214	3te	214,3te	1,2,6,7,10,48,52,53,54,55,56,57,59,60,61,62
6	2e	5te	2e,2te,212,5te	1,2,3,7,10,46,52,53,54,55,56,57,59,60,61,62,74
7	2e	204	2e,2te,212,209,204	1,2,3,6,10,47,48,52,53,54,55,56,57,59,60,61,62,74
10	10de	204	10de,10dte,204	2,3,6,7,47,48,54,57,59,60,61,62,74
13	13e	219	13e,13te,201,219	14,46,47,48,89
14	204	219	204,201,219	13,46,47,48,89
19	109e	105	109e,109,105	20,86,89
20	110e	106	110e,110,106	19,87,89
27	105	23ute	105,216,23ute	28,29,30,34,52,53,54,55,56,57,59,60,61,62,86,87
28	105	214	105,216,214	27,29,30,34,52,53,55,56,59,60,61,62,86,87
29	106	23ute	106,216,23ute	27,28,30,34,52,53,54,55,56,57,59,60,61,62,86,87
30	106	214	106,216,214	27,28,29,34,52,53,555,56,59,60,61,62,86,87
34	23de	214	23de,23dte,214	27,28,29,30,52,53,55,56,59,60,61,62
46	201	13te	201,13te	1,6,13,14,47,48,74
47	201	209	201,204,209	2,7,10,13,14,46,48,74
48	201	10dte	201,204,10dte	2,3,7,10,13,14,46,47,74
52	5e	216	5e,5te,212,214,216	1,2,3,6,7,27,28,29,30,34,53,54,55,56,57,59,60,61,62
53	5e	23dte	5e,5te,212,214,23dte	1,2,3,6,7,27,28,29,30,34,52,54,55,56,57,59,60,61,62
54	5e	2te	5e,5te,212,2te	1,2,3,6,7,10,27,29,52,53,55,56,57,59,60,61,62
55	209	216	209,212,214,216	1,2,3,6,7,27,28,29,30,34,52,53,54,56,57,59,60,61,62
56	209	23dte	209,212,214,23dte	1,2,3,6,7,27,28,29,30,34,52,53,54,55,57,59,60,61,62
57	209	2te	209,212,2te	1,2,3,6,7,10,27,29,52,53,54,55,56,59,60,61,62
59	23ue	216	23ue,23ute,216	3,6,7,10,27,28,29,30,34,52,53,54,55,56,57,60,61,62
60	23ue	23dte	23ue,23ute,23dte	3,6,7,10,27,28,29,30,34,52,53,54,55,56,57,59,61,62
61	3e	216	3e,3te,214,216	1,2,3,6,7,10,27,28,29,30,34,52,53,54,55,56,57,59,60,62
62	3e	23dte	3e,3te,214,23dte	1,2,3,6,7,10,27,28,29,30,34,52,53,54,55,56,57,59,60,61
74	10ue	209	10ue,10ute,209	1,2,6,7,10,46,47,48
86	216	109	216,105,109	19,27,28,29,30,87
87	216	110	216,106,110	20,27,28,29,30,86
89	219e	201	219e,219,201	13,14,19,20

Tabelle 14: Rail 8 Routen

B. CD-Inhalt

```
.
├── bachelorarbeit.pdf
├── interlocking_table.csv
├── mil_tests_wrong_conflict
│   ├── rail_1_r_16_and_92_not_in_conflict.txt
│   ├── rail_2_r_14_and_91_not_in_conflict.txt
│   ├── rail_3_r_31_and_68_not_in_conflict.txt
│   ├── rail_4_r_31_and_68_not_in_conflict.txt
│   ├── rail_5_r_38_and_43_not_in_conflict.txt
│   └── rail_6_r_5_and_74_not_in_conflict.txt
├── mil_tests_wrong_point
│   ├── rail_1_r_17_p_20_wrong.txt
│   ├── rail_2_r_22_p_15_wrong.txt
│   ├── rail_3_r_69_p_26_wrong.txt
│   ├── rail_4_r_32_p_25_wrong.txt
│   ├── rail_5_r_44_p_28_wrong.txt
│   └── rail_6_r_5_p_12_wrong.txt
├── rails
│   ├── rail_1.smv
│   ├── rail_2.smv
│   ├── rail_3.smv
│   ├── rail_4.smv
│   ├── rail_5.smv
│   ├── rail_6.smv
│   ├── rail_7.smv
│   └── rail_8.smv
├── readme.txt
└── verification
    ├── rail_1.txt
    ├── rail_2.txt
    ├── rail_3.txt
    ├── rail_4.txt
    ├── rail_5.txt
    └── rail_6.txt
```

Literaturverzeichnis

[BCC+16] Bozzano, Marco ; Cavada, Roberto ; Cimatti, Alessandro ; Dorigatti, Michele ; Griggio, Alberto ; Mariotti, Alessandro ; Micheli, Andrea ; Mover, Sergio ; Roveri, Marco ; Tonetta, Stefano: *nuXmv 1.1.1 User Manual*, 2016. https://es-static.fbk.eu/tools/nuxmv/index.php?n=Documentation.Home

[BLH+19] Brüning, Felix ; Leuschner, Jan ; Hölting, Raven ; Kropp, Jan R. ; Lange, Matthias ; Forquignon, Lars ; Niewöhner, Tom ; Lipps, Thomas ; Kohlhase, Felix ; Kandsorra, Niklas: *Projektbericht TEAMOD.* Version: 2019. https://gitlab.informatik.uni-bremen.de/kohlhase/teamod/tree/master/bericht

[CCD+14] Cavada, Roberto ; Cimatti, Alessandro ; Dorigatti, Michele ; Griggio, Alberto ; Mariotti, Alessandro ; Micheli, Andrea ; Mover, Sergio ; Roveri, Marco ; Tonetta, Stefano: The nuXmv Symbolic Model Checker. In: Biere, Armin (Hrsg.) ; Bloem, Roderick (Hrsg.): *CAV* Bd. 8559, Springer, 2014 (Lecture Notes in Computer Science). – ISBN 978–3–319–08866–2, S. 334–342

[FHM17] Fantechi, Alessandro ; Haxthausen, Anne E. ; Macedo, Hugo D.: Compositional Verification of Interlocking Systems for Large Stations. In: Cimatti, Alessandro (Hrsg.) ; Sirjani, Marjan (Hrsg.): *Software Engineering and Formal Methods - 15th International Conference, SEFM 2017, Trento, Italy, September 4-8, 2017, Proceedings* Bd. 10469, Springer, 2017 (Lecture Notes in Computer Science). – ISBN 978–3–319–66196–4, 236–252

[HHP17] Hong, Linh V. ; Haxthausen, Anne E. ; Peleska, Jan: Formal modelling and verification of interlocking systems featuring sequential release. In: *Sci. Comput. Program.* 133 (2017), 91–115. http://dx.doi.org/10.1016/j.scico.2016.05.010. – DOI 10.1016/j.scico.2016.05.010

[MFH16] Macedo, Hugo D. ; Fantechi, Alessandro ; Haxthausen, Anne E.: Compositional Verification of Multi-station Interlocking Systems. In: Margaria, Tiziana (Hrsg.) ; Steffen, Bernhard (Hrsg.): *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part II* Bd. 9953, 2016 (Lecture Notes in Computer Science). – ISBN 978–3–319–47168–6, 279–293

[MFH17] Macedo, Hugo D. ; Fantechi, Alessandro ; Haxthausen, Anne E.: Compositional Model Checking of Interlocking Systems for Lines with Multiple Stations. In: Barrett, Clark W. (Hrsg.) ; Davies, Misty (Hrsg.) ; Kahsai, Temesghen (Hrsg.): *NASA Formal Methods - 9th International Symposium, NFM 2017, Moffett Field, CA, USA, May 16-18, 2017, Proceedings* Bd. 10227, 2017 (Lecture Notes in Computer Science). – ISBN 978–3–319–57287–1, 146–162

[TGR13] Thomas Gibson-Robinson, Alexandre Boulgakov A.W. R. Philip Armstrong A. Philip Armstrong: *Failures Divergences Refinement (FDR) Version 3*, 2013. https://www.cs.ox.ac.uk/projects/fdr/

[TGR14] Thomas Gibson-Robinson, Alexandre Boulgakov A.W. R. Philip Armstrong A. Philip Armstrong: FDR3 — A Modern Refinement Checker for CSP. In: Ábrahám, Erika (Hrsg.) ; Havelund, Klaus (Hrsg.): *Tools and Algorithms for the Construction and Analysis of Systems* Bd. 8413, 2014 (Lecture Notes in Computer Science), S. 187–201

Eidesstattliche Erklärung

Eidesstattliche Erklärung zur Bachelorarbeit

Ich versichere, die von mir vorgelegte Arbeit selbstständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

$Unterschrift:$ $Ort, Datum:$