Master Thesis
zur Erlangung des akademischen Grades

Master of Science

Modellgetriebene Generierung von CSP-Code zur Verifikation von Gleisnetzen

Felix Brüning

Erstgutachter:
Prof. Dr. Jan Peleska

Zweitgutachter:
Prof. Dr. Christoph Lüth

Bremen, den 2. August 2021

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

KAPITEL 1

Einführung

Der heutige Alltag wird zunehmend von automatisierten Techniken unterstützt. Gerade wenn es darum geht, den Alltag für Menschen sicherer und einfacher zu machen, kommen häufig Computer zum Einsatz die Aufgaben übernehmen, für die vor einigen Jahrzehnten noch Menschen eingesetzt wurden. Diese Computer, auch eingebettete Systeme genannt, reichen dabei so weit in unsern heutigen Alltag hinein, dass ohne diese Technik das Leben kaum vorstellbar ist. Von Unterhaltungsmedien über Haushaltsgeräte bis hin zum täglichen Reisen sind computergestützte Steuerungen in allen Bereichen stark vertreten. Oft werden auch da Computer eingesetzt, wo man es kaum vermutet, wie in Eisenbahnen oder Flugzeugen. Aufgrund dieser Tatsache übernehmen Computer auch sicherheitskritische Aufgaben, wodurch sich Menschen zunehmend auf die korrekte Arbeitsweise dieser Systeme verlassen müssen. Denn diese Aufgaben werden nicht mehr vom Menschen, sondern vom Computers sowie dessen Software übernommen, die immer zur richtigen Zeit richtig entscheiden muss. Dabei rückt das Thema Verifikation und Validation immer weiter in den Fokus. Um die Korrektheit der seit Dekaden an Komplexität wachsenden Computersysteme zu verifizieren, ist ein hoher Grad an Verifikationsstärke notwendig. Jedes Fehlverhalten und jedes unerwartete Ereignis muss vor Inbetriebnahme identifiziert und beseitigt werden. Um das sicherzustellen, wird ein hoher Aufwand betrieben, um die Sicherheit und die korrekte Arbeitsweise eingebetteter Systeme zu garantieren, die den Entwicklungsaufwand maßgeblich überschreiten. Der lange Verifikationsprozess zieht sich über mehrere Stufen. Erst nach erfolgreichen Bestehen der Tests auf einer Stufe darf das System in der nächsten Stufe verifiziert werden. Von beginnender Systemmodellierung zur Verifikation des Modells gegen die formalen Sicherheitsaspekte durch Modellprüfung über modellbasierter Verifikation des implementierten Systems sowie Integrationstests bis hin zu Hardware-/Software-in-the-Loop-Tests und anschließender Inbetriebnahme vergeht ein langer Weg.
Alle diese Stufen garantieren die Sicherheit des Systems in jeden Entwicklungszustand. Das Resultat davon ist ein Computersystem, welches Alltagsaufgaben sicher und effizient bewältigt.

1.1 Praktischer Hintergrund

Vom Wintersemester 2018/19 bis zum Sommersemester 2019 wurde im Kontext eines Bachelor-Projektes ein prototypisches autonomes Eisenbahnsystem auf Basis einer Märklin-Anlage entwickelt. Dieses macht es möglich, dass Züge fahrerlos Ziele auf einem Gleisnetz ansteuern können. [BHL+19] Das entwickelte autonome Bahnsystem wurde anschließend durch zwei Bachelorarbeiten modell-geprüft, wodurch Fehler identifiziert und beseitigt wurden. Danach wurde das System dadurch erfolgreich in Anbetracht der Sicherheitskriterien verifiziert. [Brü20] [Lan19] Das anschließende Masterprojekt im Wintersemester 2019/20 bis zum Sommersemester 2020 nahm sich als Aufgabe, dass Züge in Verklemmungssituationen sich eigenständig befreien können, indem diese eine mögliche Lösung untereinander aushandeln. Um die korrekte Arbeitsweise des Systems während der Ausführung zu garantieren, wurde das System des digitalen Zwillings eingeführt. Ebenfalls wurde das Gleisnetz soweit modifiziert, dass Weichenzustände in Echtzeit an das Stellwerk gesendet und verarbeitet werden können. Zudem wurde hier das System modellbasiert getestet. Anschließend wurde die korrekte Arbeitsweise des autonomen Systems durch Software-/Hardware-in-the-Loop Tests sichergestellt. [LBK+20]

1.2 Zielsetzung

Ziel dieser Arbeit ist die Entwicklung eines Tools, welches aus einer Gleisnetzspezifikation äquivalenten CSP_M-Gleisnetz-Code generiert, der zur formalen Verifikation von Eisenbahnanlagen genutzt werden kann.
Der im Bachelor- sowie im Masterprojekt genutzte Prototyp in Form einer Märklin-Gleisanlage wurde in der Bachelor Thesis [Brü20] bereits händisch modelliert und anschließend verifiziert. Dieses Gleisnetz soll nun als Anwendungsbeispiel durch eine XML-Repräsentation dargestellt werden. Der zu entwickelnde CSP_M-Code-Generator erhält diese XML-Repräsentation als Eingabe und erzeugt daraus das Modell des Gleisnetzes in Form von CSP_M-Code. Daraus ergibt sich ein schnelleres Erstellen des CSP_M-Modells, welches händisch ziemlich zeitaufwendig ist. Zudem sind kurzfristige Änderungen schnell umsetzbar, da der Code-Generator einfach die neue XML-Repräsentation einlesen muss und daraus direkt den Code erzeugt.
Weiter wird der CSP_M-Code-Generator durch geeignete Methoden getestet und verifiziert. Dabei dient das CSP_M-Gleisnetz-Modell aus der oben genannten Bachelor-Thesis als Referenzmodell.

1.3 Verwandte Arbeiten

Automatisiert Code aus einem Modell zu generieren ist ein aktuelles Thema, wodurch in diesem Bereich viele Publikationen existieren. In [Bas20] wird beschrieben, wie durch künstlicher Intelligenz Quellcode für verschieden Plattformen und Zielsprachen aus einem Modell generiert werden kann. Das Paper [RBP19] zeigt, wie

aus einem Modell Simulink-Code generiert werden kann. Gerade hinsichtlich Code-Generatoren gibt es zudem viele proprietäre Lösungen wie IBM Rational Rhapsody. [IBM21] Ebenfalls gibt es Open-Source Lösungen wie Eclipse Papyrus. [Pro21] Weiterhin wird ein Kurs zum Thema "Spezifikation eingebetteter Systeme" an der Universität Bremen von Jan Peleska gehalten, indem es darum geht aus SysML-Modellen C++-Code zu generieren [Pel20]. In [VHP17] wird eine domänentypische Sprache vorgestellt, um generische Modelle von Stellwerken zu erzeugen. Ähnlich wie in dieser Arbeit wird eine IDL-Sprache genutzt, um Modelle zu beschreiben. Das ähnelt dem in dieser Arbeit vorgestellten Ansatz, Gleisnetze mithilfe einer XML-Spezifikation zu beschreiben. In [BDF+17] wird ein weiterer Ansatz geliefert, wie präzises Modellieren von Eisenbahn Komponenten durch Event-B und eine erweiterte Version der UML, iUML-B in Kombination mit Event-B durchgeführt werden kann. Im Bereich der Modellierung von Systemen existieren weitere Sprachen wie in [Hax14] vorgestellt. In diesem Paper vorgestellte vereinfachte Sprache RSL wird zur einfacheren Modellierung von Systemen genutzt. Gerade im Bereich der Eisenbahn Domäne lassen sich schon Spezifikationen aus der Interlocking Table automatisiert extrahieren, wie in [Hax12] gezeigt. Im Bereich des Modell-Basierten Testens nimmt die Prozessalgembra CSP ebenfalls eine bedeutende Stellung ein. Nach neusten Veröffentlichungen lässt sich aus endlichen, nicht-terminierenden oder sogar nicht-deterministischen CSP-Prozessen minimale und endliche Testmenge zum Modell-Basierten-Testen generieren. [PHC19]

KAPITEL 2

Grundlagen

In diesem Kapitel werden die für diese Arbeit benötigten Grundlagen eingeführt. Zu den Grundlagen gehören zentral die erweiterte Auszeichnungssprache XML (Kapitel 2.1), die Modellierungssprache CSP$_M$ (Kapitel 2.2), der Model Checker FDR4 (Kapitel 2.3) und nötige Implementierungen aus der Bachelorarbeit [Brü20], die auch hier in dieser Arbeit verwendet werden (Kapitel 2.4). Diese Grundlagen sind ein Bestandteil der darauffolgenden Kapitel.

2.1 XML

Die Extensible Markup Language, kurz XML, ist eine Sprache, die zur standardisierten Darstellung von hierarchischen Daten in Textform genutzt wird. [TB08] Diese 1998 vom World Wide Web Consortium (kurz: W3C) veröffentlichte Sprache erlaubt es Daten, als hierarchische Struktur darzustellen und plattformunabhängig zwischen Computern zu transferieren. XML beruht dabei auf dem zuvor weit verbreiteten Vorgänger SGML. Ebenfalls wird XML auch häufig verwendet, um aktuelle Programmzustände in Form von Objektzuständen zu speichern. Dadurch ist erneutes Wiederherstellen schnell und einfach möglich. Als Editor für XML-Dateien wird der oXygen XML-Editor empfohlen, der intuitive Funktionen bereitstellt. [SRL21] Der Standard umfasst weit aus mehr als das, was hier beschrieben wird. Damit das jedoch nicht den Rahmen dieser Arbeit überschreitet, wird nur die in dieser Arbeit benötigte Teilmenge des XML-Sprachstandards beschrieben.

2.1.1 Struktur

Eine XML-Datei gilt als valide, wenn es nach dem Standard formatiert wurde. Dieser Standard sieht dabei die Struktur vor:

```
xml_document := prolog element(s) Misc
```

Dies ist folgendermaßen zu interpretieren: Zunächst umfasst jede XML-Datei einen XML-Header, den Prolog, der den Sprachstandard, die Zeichencodierung sowie mögliche Namensräume angibt. Dieser ist zwingend erforderlich für die Korrektheit des

Dokumentes. Der Header hat dabei die Struktur, wie sie beispielsweise in Listing 1 gezeigt ist.

```
1 <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
```

Listing 1: XML-Header

Hierbei ist die XML-Version 1.0, sowie die Codierung UTF-8 angegeben zur korrekten Parsierung dieser Datei. Das Attribut `standalone=yes` sagt hierbei aus, dass keine externen Namensräume verwendet werden und hier Standard-XML verwendet wird. In dieser Datei wird also keine Form von DTD (Dokumenttyp Definitionen) verwendet (siehe Kapitel 2.1.2).
Nach dem XML-Header folgt der eigentliche Inhalt der Datei. Die Daten werden dann in Form von XML-Elementen hierarchisch in Form einer Baumstruktur dargestellt. Ein XML-Element ist immer gekennzeichnet durch einen Element-Typen eingeschlossen in spitzen Klammern. Ein sogenannter XML-Start-Tag zeigt, dass nun ein neues XML-Element beginnt. Ausgedrückt wird dies durch `<ELEMENT>`, wie in Listing 2 zu sehen. Danach werden beliebige Unterelemente deklariert. Start-Tags können ebenfalls Element-Attribute beinhalten. Wie im Listing 2 zu sehen, wird die ID des Elements auf ”elem1” gesetzt. Hierbei können beliebig viele weitere Attribute folgen.

```
1 <ELEMENT id="elem1">
2   .. Unterlemente ...
3 </ELEMENT>
```

Listing 2: Struktur von XML-Elementen

Nachdem alle Unterelemente genannt wurden, schließt ein Element immer mit dem XML-End-Tag ab. Dieser beschreibt textuell, dass das beschriebene Element nun zu Ende ist. Um den End-Tag vom Start-Tag zu unterscheiden, ist nach der ersten spitzen Klammer des Tags eine Slash eingefügt. Eine Beispielimplementierung ist im Listing 3 zu sehen.

```
1 <section id="t201">
2   <connection ref="t219" side="up" />
3   <connection ref="p13" side="down" />
4 </section>
```

Listing 3: XML-Element

Hier wird ein Element `section` beschrieben, welches jeweils zwei Unterelemente vom Typ `connection` besitzt. Wie in Zeile zwei und drei im Listing 3 zu sehen, können XML-Elemente auch direkt im Start-Tag ein End-Tag enthalten. Die beiden connection-Elemente werden am Ende direkt von einem Slash und einer spitzen Klammer geschlossen. Dies gilt dann als End-Tag und ist nach XML-Standard valide.

2.1.2 Datentyp Definitionen

Datentyp Definitionen (Data-Type-Definitions), kurz DTD, ist ein wesentlicher Bestandteil des XML-Standards. Datentyp Definitionen haben die wesentliche Aufgabe, das Sprachmodell von XML so weit zu abstrahieren, dass XML-Dateien auf jeden spezifischen Anwendungsfall angepasst werden können. Dabei ist es durch DTD möglich, XML-Typen aus anderen Dateien zu referenzieren und dann zu verwenden oder eigene Element-Tags zu definieren. Die Nutzung von DTD ist dadurch möglich, da der XML-Standard XML-Dokumente als valide ansieht, wenn die abstrakte Umsetzung, wie in Kapitel 2.1.1 gezeigt, umgesetzt wurde. Wie Datentyp Definitionen umgesetzt werden, ist in Listing 3 zu sehen. Hier haben XML-Elemente direkte Anwendungsfall-spezifische Namen, wie `section` oder `connection`.
Eigene XML-Datentyp Definitionen können nach dem XML-Header neu angelegt werden.

```
1 <!ELEMENT #name #attr>
```

Listing 4: DTD Element Typendefinition

Wie im Listing 4 zu sehen, beschreibt `#name` den Namen des neuen Elements. `#attr` beschreibt dabei die Attribute des neu angelegten Elementes. Nach der Definition des Elements kann dies als DTD-Tag in dem XML-Dokument verwendet werden.

2.2 CSP_M

CSP, Communicating Sequential Processes, ist eine Prozessalgebra, die 1985 von C.A.R. Hoare an der Oxford Universität veröffentlicht wurde. [Hoa85] Diese Sprache dient zum Modellieren und zur Verifikation von Verhaltensmodellen von Softwaresystemen. So lässt sich das Verhalten von verteilten Systemen optimal mit CSP beschreiben. Klassische Einsatzgebiete von CSP ist der Bereich der parallelen sicherheitskritischen, eingebetteten Systeme, deren Verhalten streng auf Einhaltung von Sicherheitsaspekten verifiziert werden muss. Ein Anwendungsbeispiel ist die Bachelorarbeit [Brü20], in der eine prototypische Eisenbahnanlage mithilfe von CSP zuerst modelliert und anschließend erfolgreich verifiziert wurde. Um CSP-Implementierungen automatisiert zu prüfen, existieren gewisse Refinement-Checker. Der bekannteste ist FDR, der den gesamten CSP-Sprachstandard unterstützt. Ein weiterer Refinement-Checker ist ProB, der jedoch CSP nicht vollständig unterstützt. In dieser Arbeit wird der Refinement-Checker FDR4 benutzt (siehe Abschnitt 2.3) [GABR16].
Um Systeme in CSP zu modellieren, muss die Funktionsweise und den Aufbau dieser Prozessalgebra verständlich sein. Im Folgenden wird hierbei CSP mit dem FDR-Spezifischen Dialekt CSP_M genauer erläutert.

2.2.1 CSP_M-Channel

Eines der Kernbestandteile dieser Prozessalgebra ist der sogenannte Channel-Mechanismus. Ein Channel ist im Kontext von CSP ein Kanal, über den CSP-Prozesse miteinander kommunizieren können. Eine CSP-Channel-Deklaration ist in Listing 5 zu sehen.

```
1 channel c_0,..,c_n : t_1. .. .t_n
```

Listing 5: CSP Channel Deklaration

Jeder Channel hat dabei einen Namen $(c_0, ..., c_n)$, sowie einen Typen $(t_1.....t_n)$. Ein Typ kann dabei `Int` oder `Bool` sein, jedoch auch einen Typen bestehend aus einer Liste von Werten wie $\{0..2\}$ sein. Die Liste als Typ besagt, dass auf dem Kanal die Zahlenwerte der Menge $EV = \{0, 1, 2\}$ als Event auftreten dürfen.
Der Kanal

```
1 channel chan_b : {0..2}.Bool
```

erzeugt demnach folgende mögliche Event-Menge:

$$\alpha \text{ chan\_b} = \{\langle 0.true\rangle, \langle 0.false\rangle, \langle 1.true\rangle, \langle 1.false\rangle, \langle 2.true\rangle, \langle 2.false\rangle\}$$

Damit Prozesse mithilfe von Events über Kanäle kommunizieren können, gibt es in CSP drei mögliche Operatoren.

- $c?n$: Dieser Ausdruck bedeutet, dass auf jedes Event des Kanals c gehört wird und in die Variable n gespeichert wird.
- $c!1$: Mit diesem Operator wird auf den Kanal c eine ”1” als Event gesendet.
- $c.1$: Der Punktoperator ist zentral für die Synchronisierung zuständig. Prozesse, die dieses Event beinhalten, machen gleichzeitig einen Schritt, wenn dieses Event von einem dieser Prozesse erzeugt wird.

2.2.2 CSP_M-Prozesse

Um Systeme in CSP_M darzustellen, werden CSP_M-Prozesse verwendet. CSP_M-Prozesse sind Abfolgen von Events, die auf Channel erzeugt werden. Prozesse enden nach Erzeugen der Events auf Channels in einem anderen Prozess, der auch wieder der Prozess selbst sein kann. Der einfachste aller Prozesse ist der Prozess $STOP$, der einen Prozess stoppt.
Ein Beispiel ist der Prozess $Q = a \rightarrow P$: Erst gilt a und dann verhält sich Q wie der Prozess P.

External Choice Ein Prozess-Operator ist der External-Choice Operator `[]` . Dieser dient als Pattern-Matching-Operator. Ein Beispiel zum External-Choice ist in Listing 6 dargestellt.

```
channel a,b

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

Listing 6: CSP External Choice

Der Prozess P evaluiert zunächst die Channel a und b. Je nachdem auf welchem Channel zuerst ein Event erzeugt wird, wird reagiert. Wenn zuerst auf dem Channel a ein Event auftritt, wird sich verhalten wie P, wenn jedoch auf b ein Event erzeugt wird, wird gestoppt durch *STOP*.
Zudem ist es möglich, auf bestimmte Events eines Kanals zu hören. Listing 7 zeigt dieses Verhalten.

```
channel a : {1..2}
channel b : Bool
channel c

Q = a.1 -> Q
    []
    b.True -> STOP
    []
    c -> Q
```

Listing 7: Synchronisieren auf ein bestimmtes Event

Tritt auf dem Kanal `a` ein Event `1` auf, so verhält sich Q danach wieder wie Q. Tritt auf dem Kanal `b` der Wert `True` auf, wird gestoppt und tritt ein Event auf dem Kanal `c` auf wird sich danach verhalten wie Q.

2.2.3 Synchronisation von CSP_M-Prozessen

Die Hauptaufgabe bei der Verifikation ist das Verifizieren eines Systems, wie beispielsweise ein Modell eines Stellwerkes. Ein Stellwerk besteht aus mehreren einzelnen Komponenten, die gewöhnlich nie als nur einen CSP-Prozess dargestellt und implementiert werden. So werden einzelne Funktionen als CSP-Prozesse dargestellt, die in Verbindung miteinander das gesamte Stellwerk modellieren.
Damit aus mehreren einzelnen CSP-Prozessen ein System wie ein Stellwerk entstehen kann, gibt es in CSP die Funktion, Prozesse miteinander zu synchronisieren. Die Folge dessen ist, dass die Kommunikation über CSP_M-Channels erst dann nützlich ist.
Wie im Listing 8 zu sehen, können Prozesse auf zwei Art und Weisen synchronisiert werden. SYSTEM_0 beschreibt hierbei, dass Prozess P und Prozess Q sich auf alle Events des Kanals a und b synchronisieren. SYSTEM_1 hingegen ist die Synchronisation von P und Q nur auf das Event a.0 des Kanals a.

```
1  channel a : {0,1}
2  channel b, c
3
4  P = a.0 -> b -> P
5      []
6      a.1 -> STOP
7      []
8      b -> P
9      []
10     c -> STOP
11
12 Q = a.0 -> Q
13     []
14     a.1 -> STOP
15     []
16     b -> STOP
17
18 SYSTEM_0 = P [| {| a, b |} |] Q
19 SYSTEM_1 = P [| { a.0 } |] Q
```

Listing 8: Synchronisation von Prozessen

2.2.4 CSP_M-Fehlermodelle und Verifikation

Um in CSP_M implementierte Systeme auf Eigenschaften zu prüfen, beinhaltet CSP_M einige Assertions, die Prozesse auf Refinement prüfen. Dazu gibt es drei grundsätzliche Type wie im Listing 9 zu sehen.

```
1 assert SPEC [T=  IMPL
2 assert SPEC [F=  IMPL
3 assert SPEC [FD= IMPL
```

Listing 9: CSP-Refinement Behauptung

Die Behauptung aus Zeile eins beweist hierbei die Trace-Refinement-Behauptung zwischen SPEC und IMPL. Genauer beschrieben sagt das aus, dass alle erzeugbaren Traces aus IMPL Teilmenge der erzeugbaren Traces von SPEC sind. Formal beschrieben bedeutet das dies:

$$(1) \quad SPEC \sqsubseteq_{\mathrm{T}} IMPL \quad \widehat{=} \quad traces(IMPL) \subseteq traces(SPEC)$$

Die Refinement-Behauptung aus Zeile zwei sowie aus Zeile drei zeigen einmal eine die Failures Refinement (Zeile zwei) und einmal die Failures-Divergence (Zeile drei). Die Failures Refinement ist Formal folgendermaßen definiert:

$$(2) \quad SPEC \sqsubseteq_{\mathrm{F}} IMPL \quad \widehat{=} \quad failures(IMPL) \subseteq failures(SPEC)$$

Diese sagt aus, dass zum selben Zeitpunkt IMPL und SPEC einen Fehler machen. Genauer gesagt wird hierbei geprüft, ob IMPL und SPEC dieselbe Menge Events als nicht erzeugbar ablehnen. Führt beispielsweise ein Prozess eine Menge von Events

aus und gelangt dann zu einem Punkt, an dem ein gewisses Event nicht ausgeführt werden darf, gehört dieses Event zur Fehlermenge. Ist die Fehlermenge von IMPL Teilmenge der von SPEC, dann gilt die Behauptung.
Prüfen auf Failures-Divergence hingegen untersucht SPEC und IMPL, ob diese möglicherweise in einem Lifelock-Zustand enden und sind dann äquivalent verhalten.

$$(3) \quad SPEC \sqsubseteq_{\text{FD}} IMPL \quad \widehat{=} \quad \begin{aligned} &\mathit{failures}(IMPL) \subseteq \mathit{failures}(SPEC) \wedge \\ &\mathit{divergences}(IMPL) \subseteq \mathit{divergences}(SPEC) \end{aligned}$$

Oft ist jedoch das Ziel auf nur ein bestimmtes Verhalten zu prüfen. Gerade das ist nützlich, wenn Trace-Refinement angewendet werden soll. Wenn dabei nur Trace mit bestimmten Events betrachtet werden sollen, können andere Events, die nicht betrachtet werden sollen, für den Test ausgeblendet werden. Intern werden diese Events jedoch noch erzeugt, aber in ein unsichtbares Event τ umgewandelt. Um das umzusetzen, gibt es in CSP_M den sogenannten hiding-Operator.

```
1  channel a, b, c
2
3  IMPL = a -> STOP
4         []
5         b -> IMPL
6         []
7         c -> IMPL
8
9  SPEC = a -> STOP
10
11 assert SPEC [T= IMPL \ {| b, c |}
```

Listing 10: Test auf Trace-Refinement mit dem Hiding-Operator

Listing 10 zeigt den Einsatz des hiding-Operators. Hierbei wird geprüft, ob die Trace-Refinement-Behauptung zwischen SPEC und IMPL gilt. Jedoch werden die Events b und c von IMPL verborgen, da SPEC diese nicht erzeugt.

2.3 FDR4

FDR4 ist ein Refinement Checker der ein gegebenes Modell in CSP_M gegen eine Eigenschaft auf Refinement prüft. FDR in der vierten Version wurde 2016 veröffentlicht. [GABR16] Ursprünglich wurde dieser jedoch 1995 von Formal Systems Ltd. und der Universität Oxford veröffentlicht.
Abbildung 1 zeigt die grafische Benutzeroberfläche von FDR4, eine CLI-Option existiert ebenfalls. Mit FDR4 lässt sich eine CSP_M-Datei laden und die in ihr enthaltenen Assertions prüfen. Da FDR4 keinen SAT- oder SMT-Solver zum Prüfen von Behauptungen verwendet, wird eine spezielle interne Repräsentation des Modells verwendet. Dadurch ist es möglich, dass Probleme wie State-Explosion erst bei sehr großen Modellen auftreten. Wie FDR4 ein Modell in CSP_M in eine interne Repräsentation umwandelt und dann gegen Behauptungen prüft, wird im Folgenden beschrieben.

Abbildung 1: Benutzeroberfläche FDR4

FDR4 übersetzt zunächst CSP_M in reines CSP. Reines CSP hingegen unterscheidet sich dadurch von CSP_M, dass CSP_M gegenüber CSP funktionale Eigenschaften besitzt. Diese eingebetteten Funktionen in CSP_M werden dadurch zuerst eliminiert. Anschließend wird der CSP-Code dann minimiert, sodass jeder CSP-Prozess immer nur ein Event erzeugt oder auf ein Event wartet und dann zu einem anderen Zustand übergeht. Das Minimieren ist die Voraussetzung dafür, dass der CSP-Code in ein Labelled-Transition-System umgewandelt werden kann. Dieser Vorgang wird für jedes in CSP_M implementierte Modell sowie jede in CSP_M implementierte Spezifikation durchgeführt. Das Resultat ist dann eine Menge von Labelled-Transition-Systems. [GABR16] Folgendes Beispiel soll diesen Algorithmus verdeutlichen. Zunächst wird händisch bewiesen, ob die in Listing 11 gilt, dann, wie dies in FDR4 bewiesen wird.

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

Listing 11: Muster CSP-Code

Dieses Code-Beispiel soll zeigen, wie der Test auf Trace-Refinement in FDR4 durchgeführt wird. Durch die Assertion in Zeile 13 ist zu erkennen, dass getestet werden soll, ob

$$(4) \quad traces(M0) \subseteq traces(S0)$$

gilt. Das Modell $M0$ implementiert hier das Verhalten, dass $M0$ zuerst ein Event a erzeugt und dann zum Zustand $M1$ wechselt. In $M1$ wird dann entschieden, ob ein b erzeugt wurde und zu $M0$ zurückgekehrt wird oder ein a Event erzeugt wurde und dann zu $M2$ wechselt. Im letzteren Fall wird dann ein b erzeugt. Danach folgt die Transition zum Initialzustand $M0$. Betrachtet man keine Deadlocks, ist die Menge an Traces, die von $M0$ erzeugt werden folgende:

$$\alpha M0 = \{\langle ab\rangle, \langle aab\rangle\}$$

$S0$ hingegen erzeugt ein Event a und wechselt zum Zustand $S1$, der dann wiederum ein Event a erzeugt und zum Zustand $S2$ wechselt. Der Zustand $S2$ erzeugt ein Event b und wechselt dann zum Initialzustand $S0$. Dadurch ergibt sich folgender Trace, erzeugt von $S0$:

$$\alpha S0 = \{\langle aab\rangle\}$$

Folglich kann bewiesen werden, dass die Behauptung (4) nicht gilt:

$$\{\langle ab\rangle, \langle aab\rangle\} \not\subseteq \{\langle aab\rangle\} \Rightarrow traces(M0) \not\subseteq traces(S0) \Rightarrow SPEC \not\sqsubseteq_{\mathrm{T}} IMPL \Rightarrow Failed$$

FDR4 hingegen nutzt einen etwas anderen Ansatz. Nachdem CSP$_M$-Modellierungen in ein Labelled-Transition-System (LTS) umgeformt wurden, sind zwei LTS entstanden; einmal für das Modell und einmal für die Spezifikation. Abbildung 2 zeigt hierbei einmal $M0$, sowie $S0$ aus Listing 11.

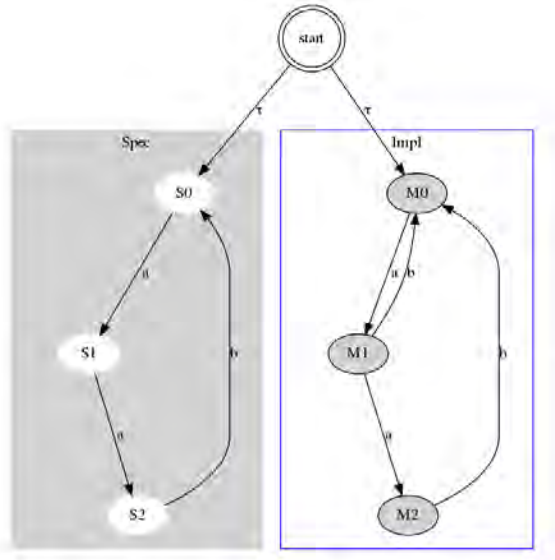

Abbildung 2: FDR4-interne CSP$_M$-Repräsentation als LTS

FDR4 vergleicht immer Zustände, ausgehend von einem Zustandspaar. Ein Zustandspaar ist hierbei ein Zustand aus der Spezifikation $S0$ sowie ein Zustand aus dem Modell $M0$. FDR4 startet im Zustand start. Da jede Folge Transition ein τ ist,

wird direkt in $S0$ sowie $M0$ gewechselt. Daraus ergibt sich das erste Zustandspaar ($S0$, $M0$). Von jedem Zustand des Zustandspaares wird nun jede ausgehende Transition beobachtet. Von $S0$ ausgehend kann eine Transition zu $S1$ genommen werden, die das Event `a` erzeugt. Gleiches gilt für die von $M0$ ausgehende Transition zu $M1$, die ebenfalls ein Event `a` erzeugt. Da jeder Zustand die gleichen Ausgehenden Transitionen hat sind diese bis hierher äquivalent. Das nächste Zustandspaar sind die Folgezustände ($S1$,$M1$). Von $S1$ gibt es eine Transition zu $S2$, welche ein Event `a` erzeugt. Gleiche existiert von $M1$ zu $M2$, die ebenfalls ein Event `a` erzeugt. $M1$ hingegen hat noch eine weitere Transition zu $M0$, die ein `b` erzeugt. Diese existiert in $S1$ nicht, wodurch ein Fehler sowie dieses Gegenbeispiel zu der Behauptung in Listing 11 ausgegeben wird.

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

Listing 12: Ausgabe von FDR4

In Listing 12 ist zu sehen, dass FDR4 die Behauptung als Failed ausgibt. Zudem wird das Fehler-Event `b` angegeben durch Error Event: `b`. [GABR16]

2.4 Abhängige CSP$_M$-Implementierungen

Wesentliche Bestandteile eines Gleisnetzes sind Kreuzweichen, Weichen sowie Streckenabschnitte, auf denen Züge detektiert werden können. Der in dieser Arbeit entwickelte CSP$_M$-Code-Generator beschränkt sich hierbei nur auf das Erzeugen eines Gleisnetzes aus einer Spezifikation, verbindet quasi einzelne Gleiselemente miteinander, sodass das spezifizierte Gleisnetz erzeugt wird. Die dafür nötigen Gleiselemente werden daher lediglich aus der Bachelor-Thesis [Brü20] übernommen, in der das gewünschte Verhalten dieser schon verifiziert wurde.

Kreuzweichen haben dabei die abstrakte Funktion, wie beschrieben in Abbildung 3. Kreuzweichen können wie normale Weichen zwei Positionen einnehmen. In der Position straight fahren Züge geradeaus durch die Weiche. In Abbildung 3 ist das der Weg von S_2 zu S_0 oder andersherum, oder von S_1 zu S_3 oder andersherum. Biegt ein Zug jedoch anders ab, entgleist dieser. Die zweite Weichen Position ist

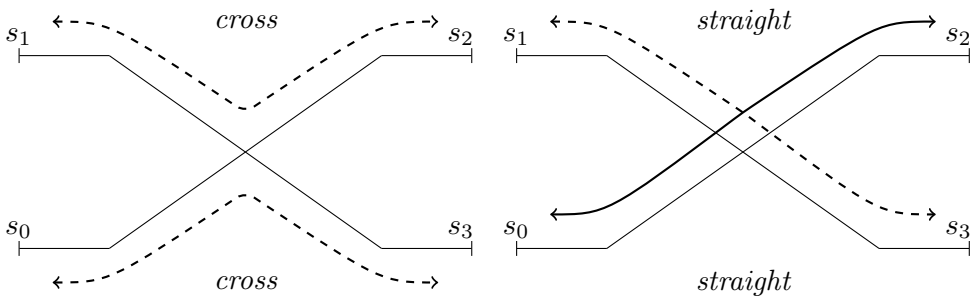

Abbildung 3: Fahrtrichtung nach Stellung der Kreuzweiche

die Cross-Position. In dieser biegt der Zug nach dem Hineinfahren in die Weiche ab. Konkret bedeutet das für die in Abbildung 3 dargestellte Weiche, dass Züge von S_2 kommend nach S_1 abbiegen und anders herum. Kommt ein Zug jedoch von S_0, biegt dieser in Richtung S_3 ab. Von S_3 kommend biegt dieser dann zu S_0 ab.
Die in CSP$_M$ modellierte Kreuzweiche, beschrieben in Abbildung 4, besitzt ähnliches Verhalten, unterscheidet sich jedoch in einigen Punkten. Der Initialzustand ist der Zustand CROSSING. Ist die Weiche in diesem Zustand und wird vom Zug befahren, wird ein Crash-Event erzeugt, da die Weiche noch nicht gestellt wurde. Damit das nicht passiert, muss die Weiche zunächst durch ein request-Befehl angefragt werden. Wenn nun ein Zug die Weiche befährt, wird wieder ein crash-Event ausgelöst, da die Weiche noch nicht gestellt wurde. Erst wenn diese nach dem request-Event durch ein set-Event gestellt wurde, darf diese befahren werden. Danach verhält sich die Weiche wie in Abbildung 3. Um die Weiche wieder für andere Züge freizugeben, gelangt man durch Erzeugen des release-Events wieder zum Initialzustand.

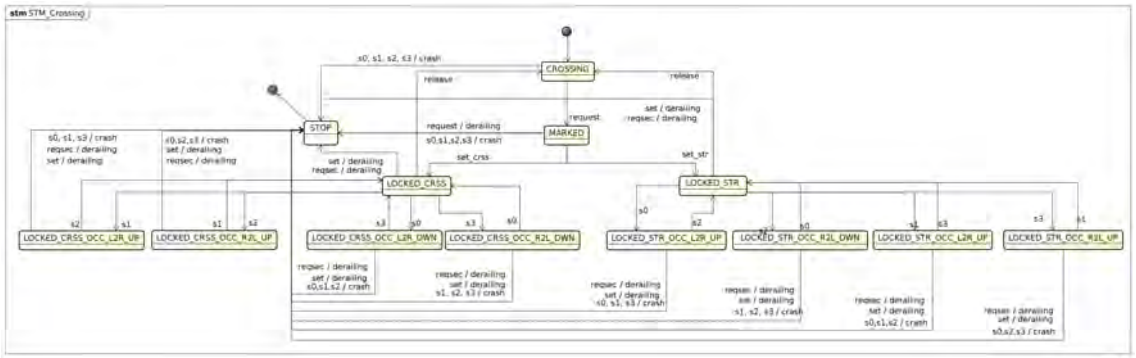

Abbildung 4: Zustandsautomat einer Kreuzweiche

Der vollständige CSP$_M$-Code ist im Anhang unter Listing 77 bis Listing 80 zu sehen. Instanziiert wird eine Kreuzweiche über den Aufruf dargestellt in Listing 13.

```
1 CROSSING(id, s0, s1, s2, s3)
```

Listing 13: Instanziierung einer Kreuzweiche

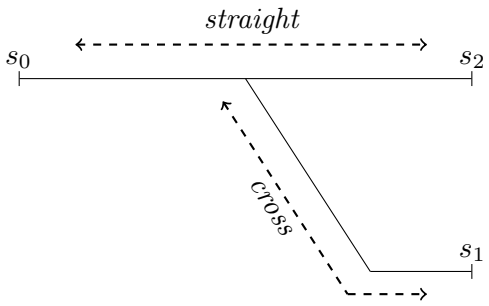

Abbildung 5: Modell einer Weiche

Wurde die Kreuzweiche angefragt und danach auch in die korrekte Position versetzt, so kann erst dann wieder die Weiche gestellt werden, wenn die Weiche wieder im Initial-, oder Marked-Zustand ist. In jedem anderen Fall wird ein Event vom Typ `derailing` ausgelöst, da durch erneutes Stellen der Weiche der darauf befindliche Zug entgleichen kann. Das gilt sowohl für die Kreuzweiche als auch für die normale Weiche.

Weichen verhalten sich ähnlich wie Kreuzweichen, haben jedoch anstatt vier nur drei Ein- und Ausfahrten. Abbildung 5 zeigt eine Weiche. Auch diese kann in zwei unterschiedlichen Zuständen gestellt werden, in straight und cross Position. In Straight-Position kann von S_0 in die Weiche hinein- und aus S_2 hinausgefahren werden oder andersherum. In Cross-Position darf ebenfalls von S_0 hinein gefahren werden, jedoch fährt der Zug dann durch S_1 hinaus. Gleiches gilt, wenn der Zug durch S_1 hinein fährt, fährt dieser durch S_0 wieder hinaus.
Die Implementierung dieser Weiche ist durch die SysML-FSM in Abbildung 6 gezeigt.

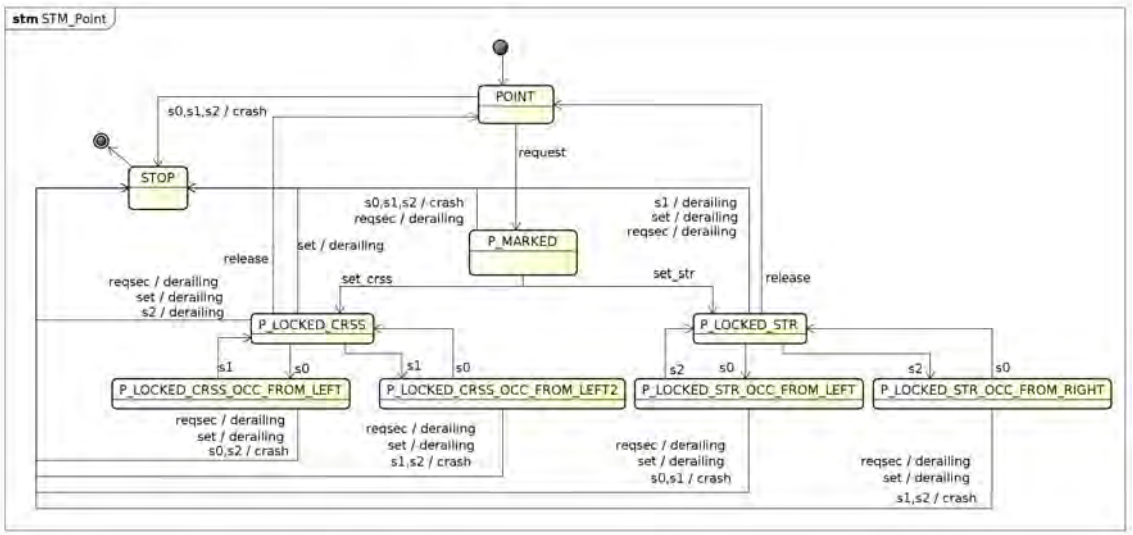

Abbildung 6: Zustandsautomat einer Weiche

Ist die Weiche im Initialzustand POINT, darf die Weiche nicht befahren werden, da

sonst ein Crash-Event ausgelöst wird, da der Zug entgleisen würde, weil die Weiche noch nicht gestellt wurde. Erst durch Anfragen der Weiche gelangt man in dem Weichen-Zustand P_MARKED. Auch in diesem Zustand ist die Weiche noch nicht gestellt, wodurch bei jedem Befahren der Weiche ein Crash-Event erzeugt werden würde. Erst durch Setzen der Weiche in die richtige Weichen-Position darf diese befahren werden. Dann verhält sich die Weiche wie in Abbildung 5 gezeigt. Um diese für andere Züge wieder freizugeben, muss ein Release-Event erzeugt werden, welches die Weiche wieder in den Initialzustand versetzt. Der CSP_M-Code befindet sich im Anhang unter Listing 81 und 82. Eine Weiche kann folglich durch den Aufruf in Listing 14 instanziiert werden.

```
1 POINT(id, s0, s1, s2)
```

Listing 14: Instanziierung einer Weiche

Track-Elemente sind die dritte Art Elemente in einem Gleisnetz. Diese haben die Aufgabe, dem Stellwerk eine Belegtmeldung mitzuteilen, wenn dieses von einem Zug befahren wird. So dienen Track-Elemente zur genauen Lokalisierung von Zügen im Gleisnetz und tragen dadurch einen großen Teil zur Sicherheit bei.
Abbildung 7 zeigt den Zustandsautomaten eines Trackelements. Zunächst ist dieses im Zustand TRACK_ELEMENT. Wenn dann ein Zug von einer Richtung hineinfährt, wird ein Occupied-Event erzeugt. Dieses teilt dem Stellwerk mit, dass ein Zug dieses gerade befährt. Folgt danach aus derselben Richtung ein Zug wird von dem Zustand TRAIN_FROM_LEFT oder TRAIN_FROM_RIGHT ein crash-Event ausgelöst. Fährt ein Zug korrekt wieder hinaus, gelangt man wieder in den Initialzustand. Der vollständige CSP_M-Code ist im Anhang unter 83 zu sehen. In Listing 15 ist die Instanziierung gezeigt.

```
1 TRACK_ELEMENT(id, s0, s1)
```

Listing 15: Instanziierung eines Zug-Detektions-Gleiselemetes

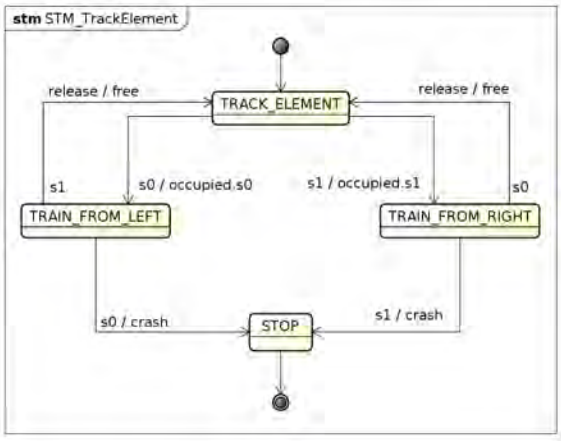

Abbildung 7: Zustandsautomat eines Trackelements

Durch das `release`-Event kann ebenfalls das Track-Element wieder freigegeben werden, wodurch ein Event `free` erzeugt wird.

KAPITEL 3

CSP$_M$-Code-Generator

Die Modellierung von sicherheitskritischen Systemen nimmt einen hohen Stellenwert in der Entwicklung ein. Dabei ist eine sehr detaillierte Beschreibung des Systemverhaltens erforderlich, damit im späteren Realbetrieb kein Fehlverhalten auftritt. Da die Modellierungen von sicherheitskritischen Systemen als Grundlage für die Verifikation nach der eigentlichen Entwicklung benötigt werden, ist ein hohes Maß an Exaktheit nötig. Nach dem Stand der Technik werden Testfälle nicht mehr händisch erzeugt, sondern automatisiert über ausgereifte Algorithmen aus dem Modell und einer Spezifikation generiert. Kommt es hier zu Fehlern im Modell, so hat das erhebliche Folgen für das Testergebnis und daraufhin für den sicheren Betrieb. Umso wichtiger ist es, dass Modelle so einfach wie möglich beschrieben werden können und trotzdem sehr aussagekräftig sind. CSP$_M$ bietet hierbei eine Prozessalgebra, die für das Modellieren von Systemen verwendet werden kann. Jedoch muss ein Modell in CSP$_M$ sehr detailliert beschrieben werden, wodurch schnell menschengemachte Fehler entstehen können. Damit die Korrektheit der Modellierung sowie zudem ein schnelles Erzeugen des Modells erreicht werden kann, wird in diesem Kapitel der CSP$_M$-Code-Generator vorgestellt. Dieser ermöglicht es, Gleisnetze, die in Form einer XML-Datei beschrieben werden, zu verarbeiten und äquivalenten CSP$_M$-Code zu erzeugen.
Im Abschnitt 3.1 wird zunächst beschrieben, in welcher Form die XML-Repräsentation vorliegen muss und welcher XML-Standard verwendet wird. Dieser Standard ist nötig, damit der CSP$_M$-Code-Generator korrekten und äquivalenten CSP$_M$-Code zum Modell beschrieben in der XML-Datei erzeugen kann. Danach wird die Analyse der XML-Datei erläutert (Abschnitt 3.2), die das in der XML-Datei dargestellte Gleisnetz auf Validität prüft. Der eigentliche Code-Generierungsprozess wird in Abschnitt 3.3 und Abschnitt 3.4 vorgestellt. Zunächst wird das Erzeugen von Gleisnetz Gruppen erläutert (Abschnitt 3.3), anschließend die Generierung der CSP$_M$ typischen Systeme des Gleisnetzes, sodass ein vollständig modelliertes Gleisnetz erzeugt werden kann.

3.1 XML - Modell - Konventionen

Der CSP_M-Code-Generator nutzt als Eingabe eine strukturierte Darstellung des Gleisnetzes in Form einer XML-Datei. Dabei ist die XML-Datei eine Art Auflistung der einzelnen Gleiselemente. Jedes Gleiselement ist dabei dargestellt als Objekt mit Verbindungen zu anderen Gleiselementen. Die Darstellungsart des Gleisnetzes orientiert sich dabei an die Gleisnetz-Darstellung von dem Projekt RobustRailS der DTU Dänemark und Anne Haxthausen. [Hax21]
Zur genauen Darstellung werden XML-Datentyp Definitionen verwendet. Ein Beispiel Gleisnetz ist in Listing 16 gezeigt.

```
1  <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2  <!DOCTYPE rail_decl [
3    <!ELEMENT network (#PCDATA)>
4    <!ELEMENT section (#PCDATA)>
5    <!ELEMENT point (#PCDATA)>
6    <!ELEMENT crossover (#PCDATA)>
7    <!ELEMENT connection (#PCDATA)>
8  ]>
9  <network id="sample">
10   <section id="s0">
11     <connection ref="c0" side="down"/>
12   </section>
13   <section id="s1">
14     <connection ref="p1" side="down"/>
15   </section>
16   <section id="s2">
17     <connection ref="p0" side="up"/>
18   </section>
19   <point id="p0">
20     <connection ref="s2" side="minus"/>
21     <connection ref="c0" side="plus"/>
22   </point>
23   <point id="p1" >
24     <connection ref="s1" side="minus"/>
25     <connection ref="c0" side="plus"/>
26   </point>
27   <point id="p2" >
28     <connection ref="c0" side="stem"/>
29   </point>
30   <crossover id="c0">
31     <connection ref="s0" side="up right"/>
32     <connection ref="p0" side="up left"/>
33     <connection ref="p1" side="down right"/>
34     <connection ref="p2" side="down left"/>
35   </crossover>
36 </network>
```

Listing 16: Beispieldarstelllung eines Gleisnetzes

Ein Gleisnetz wird deklariert durch den XML-DTD-Tag `network`, welches durch das Attribut `id` eine eindeutige ID zugewiesen bekommt, wie in Zeile 9 Listing 16 zu sehen. Mögliche Gleiselemente in einem Gleisnetz sind Section, Point und Crossing, die in dem XML-Kontext als `crossover` deklariert werden. Sections sind gerade Gleiselemente, die eine Belegtmeldung an das Stellwerk schicken, sobald sich darauf

ein Zug befindet. Points sind Weichen mit drei Ausgängen, eine Crossing ist eine Kreuzweiche mit vier Ausgängen.
Jedes Gleiselement besitzt eine eindeutige ID als Attribut. Zusätzlich hat jedes Gleiselement Verbindungen zu anderen Elementen. Eine Section (Track-Element) kann bis zu zwei Verbindungen zu anderen Gleiselementen haben. Einmal in up- und einmal in Down-Richtung. Eine Verbindung (Connection) hat dabei als Attribut einmal einen Wert `ref`, welches die Referenz zu einem anderen Gleiselement ist. Der Wert repräsentiert die ID des verbundenen Gleiselementes. Das Attribut `side` spezifiziert die Seite des Ausgangs, mit dem das Gleiselement mit der ID in ref verbunden ist. Jede Section kann dabei bis zu zwei und mindestens eine Verbindung besitzen. Ist nur eine Verbindung angegeben wie bei Section s0 in Listing 16, so ist dieses Gleiselement ein Ausgang des Gleisnetzes und stellt somit eine Verbindung zu anderen, hier nicht beschriebenen Gleiselementen dar. Weichen (Points) können bis zu drei und mindestens eine Verbindung besitzen, Kreuzweichen besitzen mindestens eine und maximal vier Verbindungen. Weichen mit drei Ausgängen haben dabei die Seite stem, plus und minus. Plus ist dabei die Fahrtrichtung geradeaus, in Position minus biegt der Zug ab. Stem ist folglich die gerade Eingangsseite. Eine Kreuzweiche hat vier Verbindungen. Up left, up right, sowie down left und down right. Betrachtet man Abbildung 8, so ist die Verbindung von c_0 zu p_1 die Seite down right, die Verbindung von c_0 zu p_2 down left. c_0 und s_0 sind über up right, c_0 und p_0 sind über up left verbunden. Die up-(left/right) Seite ist hierbei links und die down-(left/right) rechts im Bild. In der Praxis ist die Bedeutung gleich, wenn diese getauscht wären, jedoch ist es nicht erlaubt, wenn an einer Seite jeweils eine up und eine down Seite existiert. Dabei orientiert sich die Seitenbenennung an der Schaltlogik und dem Zustandsautomaten der Kreuzweiche (siehe Abbildung 4).
Das in der XML-Datei in Listing 16 dargestellte Gleisnetz ist als Bild in Abbildung 8 dargestellt. Die Track-Elemente s_1, sowie s_2 können mit der down- und up-Seite jeweils mit andere Gleiselemente verbunden sein, die hier nicht von Relevanz sind.

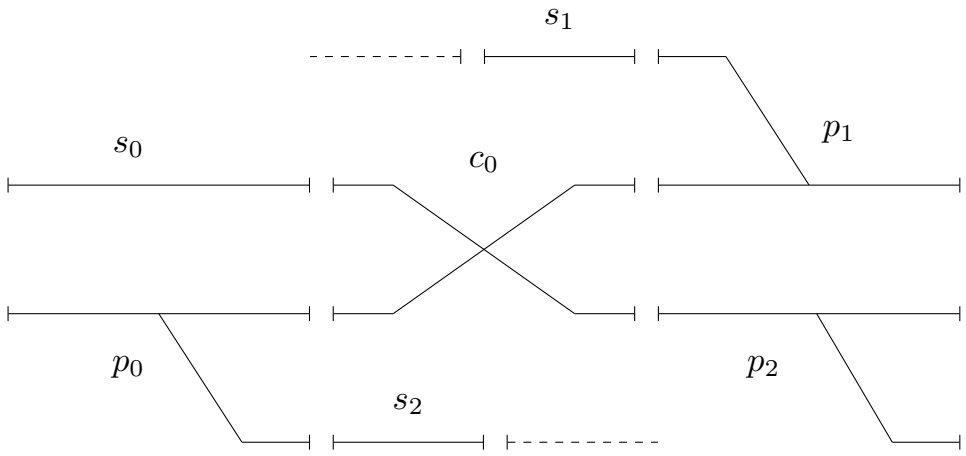

Abbildung 8: Abbildung des Beispielsgleisnetzes aus Listing 16

Dabei sind die gestrichelten Linien die Verbindungen zu Gleiselementen, die hierbei

nicht relevant sind. p_0, p_1, p_2, sowie s_0 sind verbunden mit der Kreuzweiche c_0. Section s_1 ist mit p_1 verbunden, Section s_0 ist mit c_0 verbunden.

3.2 Einlesen und Analyse der XML-Datei

Zunächst muss die Gleisnetz-beschreibende XML-Datei eingelesen (parsing) und analysiert werden. Zum Einlesen des Gleisnetz-Modells wird die C-Bibliothek LibXml2 herangezogen. Die LibXml2 Bibliothek beinhaltet einen in C geschriebenen Parser für die Metasprache XML. LibXml2 wurde ursprünglich für das GNOME-Projekt und somit für die GNOME-Desktopumgebung entwickelt. [GP21] Dabei bietet Libxml2 eine einfache API und eine stabile Implementierung. Diese implementiert zudem DTD sowie Namespaces. Somit ist diese Bibliothek für den Code-Generator ideal.
Durch wenige Funktionsaufrufe kann schnell eine XML-Datei eingelesen werden. Wie in Listing 17 zu sehen, erhält man durch einen Aufruf von `xmlParseFile(filename)` einen C-Pointer zu dem eingelesenen XML-Dokument in Form eines `xmlDocPtr`, der einen Pointer zu der XML-C-Struktur darstellt. Diese C-Struktur stellt das Dokument als Baumstruktur wie XML-Typisch dar.

```
1  static xmlDocPtr readXml(const char* filename)
2  {
3    xmlDocPtr document;
4
5    document = xmlParseFile(filename);
6
7    if(document == NULL)
8    {
9      fprintf(stderr, "File %s is empty!", filename);
10     return NULL;
11   }
12
13   return document;
14 }
15
16 dl_list_t* parseXmlToAbstractTrackElements(const char* filename)
17 {
18   Trackelements = dl_create_list();
19
20   xmlDocPtr document = readXml(filename);
21   xmlNodePtr ptr = xmlDocGetRootElement(document);
22   rootNodeName = xmlGetProp(ptr, (xmlChar*)"id");
23
24   traverseNodes(ptr);
25
26   return Trackelements;
27 }
```

Listing 17: Parsieren der XML-Datei

Wenn ein Pointer zu dem eingelesenen Dokument zurückgegeben wurde, ist das eingelesene Dokument nach der LibXml2 valide hinsichtlich der XML-Spezifikation. Danach erhält man den Pointer des ersten XML-Knotens `xmlNodePtr` mit dem

Aufruf `xmlDocGetRootElement(document)`. Dieser Knoten entspricht dem Knoten `network` aus Listing 16 (siehe Zeile 1). Mit diesem Wurzelknoten der XML-Datei existiert nun ein Pointer zum Wurzelknoten der Gleisnetze-XML-Datei. Der nächste Schritt ist nun, alle Kind-Knoten des Wurzelknotens zu iterieren.

```
1  static void traverseNodes(xmlNodePtr root)
2  {
3    int valid;
4    xmlNodePtr ptr;
5
6    for(ptr = root->children; ptr; ptr = ptr->next)
7    {
8      valid = 0;
9      const xmlChar* name = ptr->name;
10     if(!name) continue;
11
12     xmlChar* id = xmlGetProp(ptr, (xmlChar*) "id");
13     if(!id) continue;
14
15     abstractTrackElem_t trackElement;
16
17     if( !xmlStrcmp(name, (xmlChar*)"section") )
18     {
19       trackElement = newAbstractTrackElement(SECTION, id);
20       valid = 1;
21     }
22
23     if( !xmlStrcmp(name, (xmlChar*)"point") )
24     {
25       trackElement = newAbstractTrackElement(POINT, id);
26       valid = 1;
27     }
28
29     if( !xmlStrcmp(name, (xmlChar*)"crossover") )
30     {
31       trackElement = newAbstractTrackElement(CROSSING, id);
32       valid = 1;
33     }
34
35     traverseReferences(&trackElement, ptr);
36
37     if(valid)
38       dl_insert(Trackelements, &trackElement, sizeof(abstractTrackElem_t));
39   }
40 }
```

Listing 18: Parsieren der XML-Knoten

Wie in Listing 18 zu sehen, werden alle Kind-Knoten des Wurzelknotens iteriert und jeweils verglichen, um welchen Typ Gleiselement es sich handelt.
Der Aufrufgraph aus Abbildung 9 zeigt die Aufrufreihenfolge der jeweiligen C-Funktionen zum Parsieren der XML-Datei. Die Funktion
`parseXmlToAbstractTrackElements` wird dabei von der `main()`-Funktion aufgerufen und liest dabei die XML-Datei mit hilfe der XmlLib2 ein, wie in Abbildung 9 zu sehen. Danach werden alle XML-Knoten durch die Funktion `traverseNodes` traversiert und als `abstractTrackElement_t` gespeichert. Diese Funktion traversiert

wiederum ebenfalls alle Referenzen zu anderen Gleiselementen.

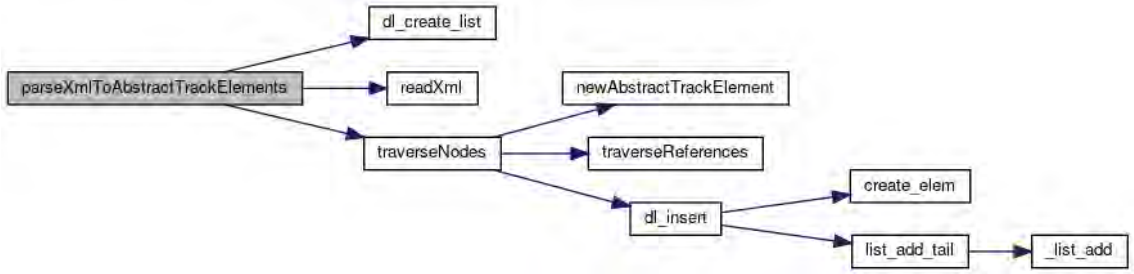

Abbildung 9: Aufrufgraph des XML-Parsers

Jedes Gleiselement wird dabei zunächst im CSP$_M$-Code-Generator abstrakt behandelt. Sections, Weichen (Point) und Kreuzweichen (Crossings) werden als C-Struktur `abstractTrackElem_t` gespeichert. So ist jedes `abstractTrackElem_t` durch eine C-Struktur in Listing 19 dargestellt. Jedes `abstractTrackElem_t` besitzt dabei ein Attribut type, welches den Gleistypen `Weiche`, `Kreuzweiche` oder Zug-Detektions-Gleis (`section`) beschreibt. Dann werden die jeweiligen Objekte erzeugt, die die Gleiselemente repräsentieren.
Mithilfe der for-Schleife aus Zeile 6 werden alle Kind-Knoten des Wurzelknotens iteriert. Mit der Operation aus Zeile 9 wird der Name, also der DTD des Knotens ermittelt. Ist dieser nicht-leer und nicht null, so wird hinterher entschieden, um welchen Typen es sich handelt (Zeile 17 bis 33). Je nachdem, um welchen Typen es sich handelt, wird ein entsprechendes `abstractTrackElem_t` erzeugt und der Liste aller eingelesenen Gleiselemente hinzugefügt. Anschließend werden alle Verbindungen (Referenzen) eingelesen.
Jedes `abstractTrackElem_t` besitzt ein Attribut `id`, welches die ID des Gleiselementes aus der XML-Datei repräsentiert. Wie in Listing 16 zu sehen, hat das `abstractTrackElem_t` der Weiche p0 die id p0. Das Attribut `isGenerated` beschreibt, ob das Gleiselement schon als CSP$_M$-Code generiert wurde. Das ist wichtig bei der Generierung der Gruppen in der ersten Phase.

```
1  typedef struct abstractTrackElement
2  {
3    enum trackElementType    type;
4    BOOL                     isGenerated;
5    xmlChar*                 isBindTo;
6    int                      bindingLayer;
7    xmlChar*                 id;
8    int                      cspId;
9    absCon_t                 connectors[MAX_CONNECTORS];
10 } abstractTrackElem_t;
```

Listing 19: C-Struktur des abstractTrackElem_t

Die Attribute `isBindTo` und `bindingLayer` beschreiben, zu welchem Gleiselement dieses verbunden ist und letzteres auf welcher Ebene (direkt an eine Weiche oder zu

einer Gruppe) dieses zu anderen Gleiselementen gebunden ist. Die `cspId` ist die ID, die das Gleiselement in der CSP$_M$-Implementierung besitzt. Da CSP$_M$ nur Zahlen als ID akzeptiert und keine Zeichen wie xmlChar (Attribut id), wird hierbei explizit eine CSP$_M$-ID erzeugt. Weiterhin besitzt jedes abstrakte Gleiselement mehrere gleistyp abhängige Verbindungen zu anderen Gleiselementen. Diese Verbindungen sind vom Typ `absCon_t` und werden im Array `connectors` gespeichert. Dabei beschreibt jede Verbindung in diesem Array eine Verbindung (Connection) aus der XML-Datei. Eine Verbindung zwischen zwei Gleiselement vom Typ `abstractTrackElem_t` ist ein C-Typ `absCon_t` im Listing 20 zu sehen.

```
1 typedef struct abstractConnection
2 {
3   xmlChar*     ref;
4   xmlChar*     side;
5   BOOL         isConnectedToGroup;
6   int          channelId;
7 } absCon_t;
```

Listing 20: C-Struktur einer abstrakten Verbindung absCon_t

Das Attribut `ref` entspricht die ID des `abstractTrackElem_t`, mit dem es verbunden ist. Das Attribut `side` beschreibt dabei, über welche Seite diese miteinander verbunden sind. Ist eine Weiche p0 mit einem Track-Element t0 über die Seite stem von p0 und up von t0 verbunden, so besitzt p0 eine Verbindung `absCon_t` mit ref = t0 und side = stem, sowie t0 eine Verbindung `absCon_t` mit ref = p0 und side = up. Die `channelId` ist dabei das CSP$_M$-Channel-Event, über dem die zwei Gleiselemente kommunizieren. Diese Channel-ID ist das CSP$_M$-Channel-Event dieser spezifischen Verbindung zwischen zwei Gleiselementen.
Um später aus dem Gleisnetz als Graphen validen CSP$_M$-Code zu erzeugen, muss die Gleisnetzbeschreibung syntaktisch korrekt sein. Bei der Generierung von CSP$_M$-Code ist es nötig, dass das Model sehr genau beschrieben wird. Kleinste Abweichungen erzeugen einen syntaktisch unterschiedlichen CSP$_M$-Code. Referenziert beispielsweise eine Weiche ein Gleiselement, das so gar nicht in der XML-Datei existiert, so entspricht der generierte CSP$_M$-Code nicht der eigentlichen Spezifikation. Dadurch können Tests fälschlicherweise bestehen, die eigentlich fehlschlagen würden. Das hat eine fatale Wirkung auf das Testergebnis sowie auf das reale Systemverhalten. Um vollständig korrekten CSP$_M$-Code zu generieren, ist es zwingend nötig, dass das Modell bestimmte Anforderungen erfüllt.

Die ID eines Gleiselementes ist eindeutig. Damit das in der XML-Datei spezifizierte Gleisnetz von dem Code-Generator bei der Codegenerierung richtig interpretiert werden kann, darf eine ID eines Gleiselementes in der XML-Datei nur einmal vergeben werden. Wenn IDs nicht eindeutig sind, können Verbindungen zwischen Gleiselementen falsch interpretiert werden, wodurch kein semantisch korrekter CSP$_M$-Code entstehen kann.

Keine Selbstreferenzen in einem Gleisnetz. Wenn Gleiselemente an den Ausgängen eines Gleiselementes sich selbst referenzieren, kann ein Zug das Gleiselement mehrmals zur selben Zeit befahren. Zur Modellierung in CSP$_M$ ist dies jedoch nicht zulässig, da nicht sichergestellt werden kann, ob eine Weiche für den Zug korrekt gestellt ist, sodass kein Zug entgleist. Besteht ein Gleisnetz nur aus einer Weiche und ein Zug soll von s_2 nach s_3 fahren (siehe Abbildung 5), so muss die Weiche von Straight-Stellung in Cross-Stellung versetzt werden. Dabei muss jedoch sichergestellt werden, dass der Zug nicht entgleist. In diesem Fall ist das nicht möglich.

Jedes Gleiselement muss erreichbar sein. Damit ein Gleisnetz erzeugt werden kann, muss jedes Gleiselement von mindestens einem anderen Gleiselement erreichbar sein. Falls es ein freies, zu keinem weiteren Gleiselement gebundenes Gleiselement gibt, schlägt der Algorithmus im Code-Generieren aus Abschnitt 3.4 fehl. Gibt es jedoch ungebundene Gleiselemente, muss vorher abgebrochen werden. Diese Funktion iteriert durch alle eingelesenen Gleiselemente und prüft, ob jedes Element der Liste von mindestens einem anderen Gleiselement verbunden ist.

Jede Referenz muss erreichbar sein. Nicht nur jedes Gleiselement muss erreichbar sein, sondern auch die Gleiselemente, die von einem Gleiselement referenziert werden. Wie im Abschnitt 3.1 beschrieben, ist ein Gleiselement über Referenzen mit anderen Gleiselementen verbunden. Die Referenz wird beschrieben durch die ID des Gleiselementes, mit dem es verbunden ist. Diese ID muss auch im Gleisnetz existieren.

Durch `abstractTrackElem_t` und `absCon_t` wird das eingelesene Gleisnetz intern in einer Graphenstruktur gespeichert. Nach dem Erstellen der Gleisnetzspezifikation wird für jedes `abstractTrackElem_t` eine eindeutige `cspId` vergeben. Diese ist nötig für die spätere Code-Generierung des CSP$_M$-Codes. Anschließend werden die Verbindungen zwischen zwei abstrakten Gleiselementen genauer betrachtet (Listing 21): Zunächst werden für alle Verbindungen eines `abstractTrackElem_t` eine neue, eindeutige Kanal-ID vergeben. Diese ist für die Kommunikation der beiden Gleisnetze im CSP$_M$-Code nötig. Ebenso gibt es eine Verbindung in entgegengesetzter Richtung, ausgehend von einem anderen `abstractTrackElem_t`. Diese Verbindung wird gesucht und dessen Kanal ID zu diesem Kanal ID gesetzt. Diese beiden Verbindungen sind identisch, referenzieren jedoch in umgekehrter Richtung. Das Setzen der ID übernimmt die Funktion `updateAbstractTEChannelID()`.
Die for-Schleife in Zeile 4 iteriert durch alle im Gleisnetz befindlichen Gleiselemente. Die zweite for-Schleife iteriert durch alle Verbindungen innerhalb des aktuellen `abstractTrackElem_t`. Ist eine Verbindung einer Seite ungebunden (if-Bedingung Zeile 13 bis 20), so wird eine Kanal-ID gesetzt, die undefiniert ist. Im Fall, dass eine Verbindung an der aktuellen Seite existiert (Zeile 24 bis 28), so wird die nächst-freie Kanal-ID gesetzt (Zeile 24) und die Kanal-ID in entgegengesetzter Richtung mit der

Funktion `updateAbstractTEChannelId` gleich zu diesem Wert gesetzt.

```
1  void generateChannelIds()
2  {
3    struct list_head* ptr;
4    for_each(ptr, allTrackElements->head)
5    {
6      abstractTrackElem_t* cr = (abstractTrackElem_t*)ptr->data;
7      cr->cspId = getNextFreeCspId();
8
9      int i;
10     for(i = 0; i < MAX_CONNECTORS; i++)
11     {
12       // If a connector is not set
13       if(! cr->connectors[i].ref)
14       {
15         xmlChar buf[20];
16         cr->connectors[i].ref = calloc(20, sizeof(char));
17         sprintf((char*)buf, "%d", getNextUndefinedId());
18         xmlStrcat(cr->connectors[i].ref, buf);
19         cr->connectors[i].side = (xmlChar*)"NULL";
20         cr->connectors[i].channelId = getNextFreeChannelId();
21
22       } else if( cr->connectors[i].channelId == -1 )
23       {
24         cr->connectors[i].channelId = getNextFreeChannelId();
25         updateAbstractTEChannelId(allTrackElements,
26                         cr->connectors[i].ref,
27                         cr->id,
28                         cr->connectors[i].channelId);
29       }
30     }
31   }
32 }
```

Listing 21: Funktion zum Generieren der CSP-Channel IDs

3.3 Abbildung von XML - Elementen auf CSP_M

Zur Generierung von CSP_M-Modellen gibt es zahlreiche Herangehensweisen. Um jedoch aus der verarbeiteten XML-Datei in Form eines Graphen äquivalenten CSP_M-Code zu generieren, ist strukturiertes Vorgehen nötig. Ein Graph mit vielen Abhängigkeiten ist hierbei so zu verarbeiten, dass schließlich ein CSP_M-Prozess das gesamte semantische Gleisnetz-Modell beschreibt. Der in dieser Arbeit vorgestellte Algorithmus arbeitet nach dem Prinzip des Erstellens von Gruppen im Graphen. Zunächst werden einzelne Gleiselemente mit Verbindungen zueinander zu Gruppen hinzugefügt. Diese Gruppen werden schließlich miteinander verknüpft, sodass ein einziger Gleisnetz-CSP_M-Prozess entsteht. Dafür unterteilt der Algorithmus die im Gleisnetz befindlichen Gleiselemente in sogenannte High-Level und Low-Level Gleiselemente. Dabei sind High-Level Gleiselemente Weichen und Kreuzweichen. Diese gelten als High-Level Gleiselemente, da diese aktiv die Richtung des Zuges und die Folgegleise bestimmen, die der darauf befindliche Zug als nächstes befährt. Low-Level Gleiselemente hingegen werden von Zügen befahren, um dem Stellwerk ein Belegt-Signal

mitzuteilen. Typische Gleiselemente sind dabei Zug-Detektions-Gleise, die die einzige Aufgabe haben, dass Züge diese passieren und nur in einer Richtung hinaus fahren können. Anschließend melden diese dem Stellwerk eine Belegtmeldung, falls sich ein Zug darauf befindet.
Eine Gleisnetz-Gruppe besteht aus miteinander verbundenen Gleiselementen, die sich über Verbindungen referenzieren (siehe Listing 19). Zwei Gleiselemente bilden eine Gruppe, wenn diese mindestens über eine Seite miteinander verbunden sind, folglich sich also über mindestens eine Verbindung referenzieren (Listing 20).

```
<section id="s2">
  <connection ref="p0" side="up"/>
</section>
<point id="p0">
  <connection ref="c0" side="plus"/>
  <connection ref="s2" side="minus"/>
</point>
```

Listing 22: Point und Track-Element verbunden in der XML-Datei

Das Gleisnetz, dargestellt in Listing 22, zeigt hierbei eine Zweiergruppe. Die Weiche p0 bildet mit s2 jeweils eine Gruppe. Wäre nun p0 mit einem weiteren Track-Element verbunden, so würden diese dann eine Dreiergruppe bilden. Das gilt, da Gruppen ebenfalls selber aus Gruppen bestehen können.
Dieser Teil des Gleisnetzes ist durch Abbildung 10 gezeigt. Zu sehen ist die Weiche p_0 verbunden mit dem Track-Element s_2.

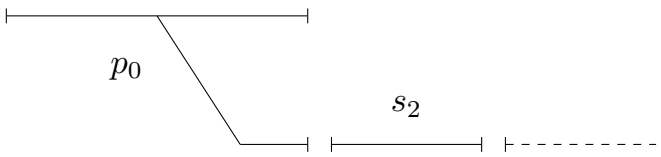

Abbildung 10: Teilnetz des Gleisnetzes

In der ersten Phase werden zunächst alle High-Level Gleiselemente betrachtet. Diese bilden in diesem Kontext den Kern der initial zu erstellenden Gruppen, die nur aus Gleiselementen bestehen.

```
typedef struct network_group
{
  char*                   name;
  enum trackElementType   type;
  dl_list_t*              track_elements;
  dl_list_t*              outerRefs;
} network_group_t;
```

Listing 23: C-Struktur einer Gleisnetz-Gruppe

Der Algorithmus sucht dann alle um diesem High-Level Gleiselement befindlichen Low-Level Gleiselementen und versucht diese, soweit noch nicht gebunden, zu dieser Gruppe hinzuzufügen.

Eine Gleisnetz-Gruppe wird dargestellt durch die C-Struktur `networkGroup_t`, wie in Listing 23 zu sehen. Der Name der Gruppe entspricht dem Namen sowie der Typ dem Typen des High-Level-Gleiselementes. In der Double-Linked-List `track_elements` sind alle Gleiselemente der Gruppe gespeichert, sowohl High-Level, als auch Low-Level Gleiselemente. Die Liste `outerRefs` speichert alle Verbindungen zu anderen Gleiselementen, die sich nicht in dieser Gruppe befinden. Sind wie in Listing 22 p0 und s2 in dieser Gruppe, so ist in `outerRefs` die Verbindung zu c0.

Das grundsätzliche Vorgehen wird hierbei mithilfe des Listings 22 beschrieben. Zunächst wählt der Algorithmus die Weiche p0 aus. Die neue Gruppe bekommt folglich den Namen p0 und die Weiche p0 wird zu den Gleiselementen der Gruppe hinzugefügt. Um nun weitere Elemente für diese Gruppe zu suchen, wird durch alle Verbindungen der Weiche p0 iteriert. Dabei werden alle Gleiselemente betrachtet, die den Typ SECTION haben, also ein Gleiselement zur Detektion von Zügen und folglich Low-Level Gleiselemente sind. Wenn diese noch zu keiner Gruppe gebunden sind (`isBindTo` $\equiv$ `NULL`, Listing 19), werden diese zu den Elementen dieser Gruppe hinzugefügt. Danach wird das Attribut `isBindTo` von diesen Gleiselementen auf den Namen dieser Gruppe und `bindingLayer` auf den Wert eins gesetzt. Der Wert eins im Binding-Layer sagt in diesem Kontext aus, dass diese Low-Level Gleiselemente direkt an ein High-Level Gleiselement gebunden sind. Nachfolgend besteht die Gruppe aus der Weiche p0, sowie aus dem Zug-Detektions-Gleis s2.

Der Folgeschritt ist dann alle Verbindungen der Gruppenelemente zu finden, die noch zu keiner Gruppe gebunden sind. Das ist nötig, damit später Gruppen miteinander verbunden und über genau diese Verbindungen miteinander verknüpft werden können. In diesem Fall ist das genau die Verbindung von p0 zu c0. Diese ist die Verbindung über plus zu c0. Das Gleiselement c0 befindet sich nicht in dieser Gruppe, daher muss diese Verbindung zu einem späteren Zeitpunkt aufgelöst werden. An Section s0 hingegen sind schon alle möglichen Verbindungen gebunden. Die Verbindung zu c_0 wird zu den `outerRefs` hinzugefügt.

Nun werden nicht mehr die einzelnen Elemente, sondern die Gruppe als Ganzes betrachtet. Es existiert eine unaufgelöste Verbindung in dieser Gruppe, die die Kreuzweiche c0 referenziert. Es kann jedoch der Fall auftreten, dass eine Gruppe äußere Verbindungen hat, die jeweils dieselbe Weiche referenzieren. Obwohl diese äquivalent scheinen, werden trotzdem beide zu der Liste der `outerRefs` hinzugefügt. Dieses Vorgehen ist wichtig für die Codegenerierung, was im Abschnitt 3.4 genauer beschrieben wird.

Nachdem Gruppen gebildet wurden sowie alle benachbarten Low-Level Gleiselemente zu High-Level Gleiselementen hinzugefügt wurden, kann es vorkommen, dass noch ungebundene Low-Level Gleiselemente existieren. Das ist genau dann der Fall, wenn sich im Gleisnetz zwischen zwei High-Level Gleiselementen über eine Verbindung drei Low-Level Gleiselemente befinden. Existieren zwei Weichen, die in einer Route von einem Zug befahren werden können und existieren zwischen den beiden stem-Seiten drei Track-Elemente, so werden die der Weiche nächsten Track-Elemente

der Gruppe der Weiche hinzugefügt und das mittlere bleibt ungebunden. Um ungebundene Gleiselemente zu einer Gruppe zu binden, wählt der Algorithmus eine Gruppe aus, zu der dieses Low-Level Gleiselement gebunden werden kann. Dabei wird die erste Gruppe ausgewählt, die infrage kommt. Der Binding-Layer des Track-Elementes wird dann auf den Wert gesetzt, wie viele Elemente bereits in der Gruppe existieren.
Erst nachdem alle Gleiselemente zu einer Gruppe gebunden wurden, wird mit der zweiten Phase fortgesetzt. Die Syntaxprüfung der Modellierung stellt sicher, dass alle Gleiselemente zu Gruppen gebunden werden können und keine freien Gleiselemente mehr existieren.

In der zweiten Phase wird der CSP$_M$-Code für alle erstellten Gruppen der ersten Phase generiert, da diese Gruppen nicht weiter verändert werden. Zur Generierung werden die vordefinierten Gleiselemente aus [Brü20] verwendet. Die jeweiligen Gleiselemente werden, wie im Abschnitt 2.4 gezeigt, verwendet. In der ersten Phase wurden die in einer Gruppe befindlichen Elemente miteinander verbunden. Der Name sowie der Typ der Gruppe definieren den Namen des CSP$_M$-Prozesses, der diese beschreibt. Da die Gruppe vom Typ Point ist und den Namen p0 trägt, wird folglich der CSP$_M$-Prozess POINT_p0 erstellt. Danach werden die einzelnen Elemente generiert und in die Ausgabedatei geschrieben. Begonnen wird dabei mit den Low-Level Gleiselementen mit dem höchsten Binding-Layer, sodass diese sich nach der CSP$_M$-Syntax mit dem High-Level Gleiselement verbinden. Bei der Instanziierung eines Gleiselementes in CSP$_M$ bekommt die Instanz die in der C-Struktur gespeicherte und zuvor generierte CSP-ID (Listing 19). Innerhalb der Verbindungen wird dann für jede Seite des Gleiselementes die richtige zuvor vergebene CSP-Channel ID gesucht und der Instanz übergeben. Ein Zug-Detektion-Gleiselement mit der CSP-ID 5 und den Verbindungen über up zu p0 mit Channel-ID 2 und Verbindung über down zu c0 mit der CSP-Channel-ID 3 wird instanziiert als

```
POINT_p0  =  ( TRACK_ELEMENT(5, 2, 3).
```

Danach wird das `isGenerted`-Flag innerhalb der C-Struktur `abstractTrackElem_t` des instanziierten Gleiselementes auf true gesetzt. Im nächsten Schritt wird der CSP$_M$-Kanal ermittelt, über welchen dieses zuvor instanziierte Gleiselement mit dem in der Gruppe befindlichen High-Level Gleiselement verbunden ist. In diesem Fall ist das der Channel mit der ID 2. So wird nun das Synchronisationsevent erzeugt:

```
POINT_p0  =  ( TRACK_ELEMENT(5, 2, 3) [|{ rail.2 }|]
```

Nach diesem Vorgehen werden alle Low-Level Gleiselemente der Gruppe instanziiert. Danach folgt das Erstellen des High-Level-Gleiselementes. Dieses wird dann wie die Low-Level Gleiselement instanziiert, jedoch wird kein Synchronisationsevent erzeugt. Der erzeugte CSP$_M$-Code der Gruppe ist dann folgender:

```
POINT_p0  =  ( TRACK_ELEMENT(5, 2, 3) [|{ rail.2 }|]
               POINT(0, 30, 4, 2) )
```

Hierbei ist zu sehen, dass in der Instanz von Weiche p0 diese über den CSP$_M$-Kanal 2 mit dem Gleiselement 5 verbunden ist. Die Kanalnummer 30 am stem-Ausgang der Weiche deutet darauf hin, dass hierbei kein Element verbunden ist. Ungebundenen Ausgänge in einem Gleisnetz Modell werden durch hohe Kanal IDs dargestellt. In dem Gleisnetz aus Abbildung 8 existieren drei Weichen, eine Kreuzweiche und zwei Track-Elemente. Insgesamt sind also nur 6 Verbindungen nötig, um das Gleisnetz darzustellen. Die Weiche p0 ist in diesem Gleisnetz über die stem-Seite nicht an einem anderen Gleisnetz gebunden. Zur Vollständigkeit erhält diese Seite eine viel höhere ID, als es im Gleisnetz benötigt wird. Zu erkennen ist ebenso, dass Point 0 noch nicht über Kanal ID 4 mit einem Gleiselement verbunden ist, Track-Element 5 ist über Kanal ID 3 noch nicht mit einem anderen verbunden.

3.4 Generierung des Gleisnetzes aus Gleiselementen

Wie in Abschnitt 3.3 beschrieben, wurde das Gleisnetz in sogenannte Gleisnetz-Gruppen unterteilt. Der korrespondierende CSP$_M$-Code wurde ebenfalls erzeugt. Der nächste nötige Schritt ist, die erstellten Gruppen weiter zu größeren Gruppen zusammenzufassen mit dem Ziel, dass das Gleisnetz schlussendlich durch einen einzigen CSP$_M$-Prozess beschrieben werden kann. Durch die Komposition der einzelnen Gruppen wird dieser Schritt durchgeführt.
Der Algorithmus terminiert, wenn alle Gruppen soweit verknüpft wurden, dass ein einzelner CSP$_M$-Prozess erzeugt wurde, der das gesamte Gleisnetz beschreibt. Dazu wird im ersten Schritt zunächst eine beliebige initiale Gruppe ausgewählt. Anschließend wird nach einer zweiten Gruppe gesucht, die mit dieser verbunden ist. Das ist die Gruppe, deren ungebundene Verbindungen Elemente aus dieser Gruppe referenzieren. Wurde eine Gruppe gefunden, wird das erste System erzeugt. Dieses erhält den Namen SYSTEM_N, wobei N eine Zahl zwischen 1 und $k-1$ ist, wobei k die Anzahl der existierenden Gruppen ist. Begonnen wird dabei bei dem höchsten Wert für N und in jedem Schritt dekrementiert, sodass das finale System die Nummer 1, folglich den Namen SYSTEM_1 hat.
Algorithmus 1 zeigt die Implementierung. Im ersten Fall gibt es nur Gruppen und noch keine Systeme. Systeme sind dabei miteinander verbundene Gruppen. Nachdem die erste Gruppe ausgewählt wurde, wird eine Gruppe gesucht, die mit dieser verbunden ist. Diese bilden dann das erste System, welches anschließend in die Ausgabedatei geschrieben wird. Diese beiden Gruppen bilden eine neue Gruppe, also das erste System. Somit werden diese beiden Gruppen verbunden durch `merge(last_group, g)` und ergeben eine neue Gruppe vom Typ `networkGroup_t`. Anschließend ist die last_group die zuvor neu erstellte `networkGroup_t`-Gruppe. Im nächsten Schritt wird die nächste freie Gruppe ausgewählt und versucht, diese mit der in der vorherigen Iteration erstellten Gruppe zu verbinden.

Algorithm 1: GENERATE_SYSTEMS

```
last_group = NULL;
while still groups exist do
  next:
  for ∀ g ∈ groups do
    if first_element then
      strongestConnected = getStrongestConnectedGroup();
      first_element = false;
    else
      strongestConnected = last_group;
    end if
    if strongestConnected not null then
      LIST channels = getChannelsBetweenGroups(g, strongestConnected);
      if channels not empty then
        printNextSystem(g, last_group, channels);
        last_group = merge(last_group, g);
        goto next;
      end if
    end if
  end for
end while
```

Falls diese Gruppen über äußere Verbindungen miteinander verbunden sind, wird das neue System in die Ausgabedatei geschrieben und diese beiden Gruppen zu einer neuen Gruppe zusammengefasst. Falls jedoch keine Abhängigkeiten zwischen diesen Gruppen existiert, wird der Vorgang erneut mit der nächsten freien Gruppe wiederholt. Aus dem Beispiel-Gleisnetz aus Listing 16 erstellt der Algorithmus den CSP_M-Code aus Listing 24. SYSTEM_3 beschreibt das System in erster Iteration. SYSTEM_3 ist die Verbindung der Gruppen CROSSING_c0 und POINT_p0.
SYSTEM_2 fügt die Gruppe POINT_p1 zu dem SYSTEM_3 hinzu. SYSTEM_1 fügt die Gruppe POINT_p2 zu dem SYSTEM_2 hinzu. SYSTEM_1 ergibt schließlich das `NETWORK`, also das Gleisnetz aus Abbildung 8 und Listing 16.
Terminiert der Algorithmus, so wird die Ausgabedatei geschlossen und gespeichert. Der erstellte CSP_M-Prozess `NETWORK` repräsentiert das Gleisnetz aus der XML-Spezifikation. Das somit generierte Gleisnetz kann nun für Verifikation genutzt werden.

```
1  ------------------------------------------------------------
2  -- This file was generated by the 'XML to CSP' Code Generator
3  -- Version 0.5.4
4  --
5  -- by Felix Brüning, University of Bremen 2021
6  ------------------------------------------------------------
7
8  channel rail : {0 .. 22}
9
10 include "channel.csp"
11 include "track_elem.csp"
12 include "crossing.csp"
13 include "point.csp"
14
15 -- Group elements:
16 -- CSP-ID 1 -> 's0'
17 -- CSP-ID 7 -> 'c0'
18 CROSSING_c0 = ( TRACK_ELEMENT(1, 1, 0) [| { rail.0 } |]
19           CROSSING(7, 12, 0, 15, 18) )
20
21 -- Group elements:
22 -- CSP-ID 3 -> 's2'
23 -- CSP-ID 4 -> 'p0'
24 POINT_p0 = ( TRACK_ELEMENT(3, 8, 9) [| { rail.8 } |]
25           POINT(4, 13, 8, 12) )
26
27 -- Group elements:
28 -- CSP-ID 2 -> 's1'
29 -- CSP-ID 5 -> 'p1'
30 POINT_p1 = ( TRACK_ELEMENT(2, 5, 4) [| { rail.4 } |]
31           POINT(5, 16, 4, 15) )
32
33 -- Group elements:
34 -- CSP-ID 6 -> 'p2'
35 POINT_p2 = POINT(6, 18, 19, 19)
36
37 SYSTEM_3 = CROSSING_c0 [| { rail.12 } |] POINT_p0
38 SYSTEM_2 = POINT_p1 [| { rail.15 } |] SYSTEM_3
39 SYSTEM_1 = POINT_p2 [| { rail.18 } |] SYSTEM_2
40
41 NETWORK = SYSTEM_1
```

Listing 24: CSP_M-Code des Gleisnetzes

KAPITEL 4

Verifikation

4.1 Äquivalenzprüfung mit der Referenzimplementierung mithilfe von FDR4

Um die Korrektheit des CSP_M-Code Generators zu zeigen, werden Äquivalenztest und Äquivalenzbeweise zwischen generierten Gleisnetzen und den jeweiligen Referenzmodellen durchgeführt. Zur Verifikation des CSP_M-Code-Generators werden Referenzimplementierungen aus [Brü20] verwendet. Dazu zählen einmal das kleine Gleisnetz, welches einen vereinfachten Teil des TEAMOD-Gleisnetzes darstellt und das TEAMOD-Gleisnetz selbst. Mithilfe von FDR4 soll bewiesen werden, ob das Referenzmodell äquivalent zu dem generierten Gleisnetz ist. FDR4 stellt dazu hilfreiche Funktionen zum Prüfen von Assertions bereit, die das Prüfen auf Äquivalenz zweier CSP_M-Implementierungen ermöglichen. Wie im Abschnitt 2.2.4 beschrieben, kann durch Prüfen auf Failures Refinement die Äquivalenz zweier CSP_M-Prozesse bewiesen werden. Mithilfe dieser Methode wird in diesem Kapitel die Äquivalenz der generierten Gleisnetze mit den Referenzimplementierungen aus der Bachelor-Thesis [Brü20] gezeigt. Im Abschnitt 4.1.2 wird durch Prüfen auf Failures Refinement die Äquivalenz des generierten kleinen Gleisnetzes zum Referenzmodell gezeigt. Im Abschnitt 4.1.3 werden verschiedene Ansätze geliefert, wie die Korrektheit des Code-Generators in Bezug zu sehr großen Modellen gezeigt werden kann. Dazu wird das Aufstellen und Durchführen von Äquivalenztests mithilfe eines Property-Katalogs und das Verifizieren durch Äquivalenzbeweise mithilfe kompositioneller Verifikation erläutert.

4.1.1 Anpassen der Modellierungen

Vor der Äquivalenzprüfung müssen die CSP_M-Schnittstellen beider Gleisnetze so umgeformt werden, dass diese vergleichbar sind. Die Referenzimplementierung stellt zwar das semantisch gleiche Gleisnetz wie das generierte Gleisnetz dar, unterscheidet sich jedoch hinsichtlich CSP_M-Events, mit denen Gleiselemente in CSP_M kommunizieren. Zudem unterscheiden sich Gleiselemente hinsichtlich ihrer ID, obwohl diese

das semantisch selbe Gleiselement implementieren. Somit müssen diese Werte vorerst angepasst werden, da sonst die Äquivalenzprüfung von vornherein fehlschlägt. Diese Umformung kann einerseits händisch erfolgen oder mit dem Renaming-Operator von CSP_M. Dieser Operator implementiert die Funktionalität, dass bestimmte Events eines Prozesses umbenannt werden können. Da diese Methode für große Modelle mit einer großen Eventmenge sehr aufwendig ist, wird in diesem Fall die händische Variante gewählt. Modelliert ein CSP_M-Prozess beispielsweise das Track-Element t0 ein Weiterer die Weiche p0, sind diese in der Referenzimplementierung über `rail.5` verbunden in dem generierten Gleisnetz jedoch über `rail.9` so müssen diese soweit umgeformt werden, dass diese über dasselbe Event synchronisiert werden. Beide müssen sowohl in der Referenzimplementierung als auch in dem generierten Gleisnetz über dasselbe Event synchronisiert werden. Jedes Gleiselement sowohl Weiche als auch Track-Element besitzen eine eindeutige ID, über der diese adressiert werden. Modellieren zwei CSP_M-Prozesse das semantisch selbe Gleiselement besitzen jedoch unterschiedliche IDs so müssen diese ebenfalls auf denselben Wert gesetzt werden. Beide Gleisnetze sind anschließend vergleichbar und für die Verifikation vorbereitet.

4.1.2 Kleines Gleisnetz

In der Bachelor-Thesis [Brü20] wurde zur Verifikation vorerst ein Teil des zu testenden Gleisnetzes benutzt, um die Modellprüfung mit CSP_M anhand dessen zu erläutern. Dieses Gleisnetz ist der vereinfachte inneren Ring des TEAMOD-Gleisnetzes und wird zur Äquivalenzprüfung hierbei verwendet. Das genaue Vorgehen wird dabei anhand dieses Gleisnetzes aus Abbildung 11 gezeigt.

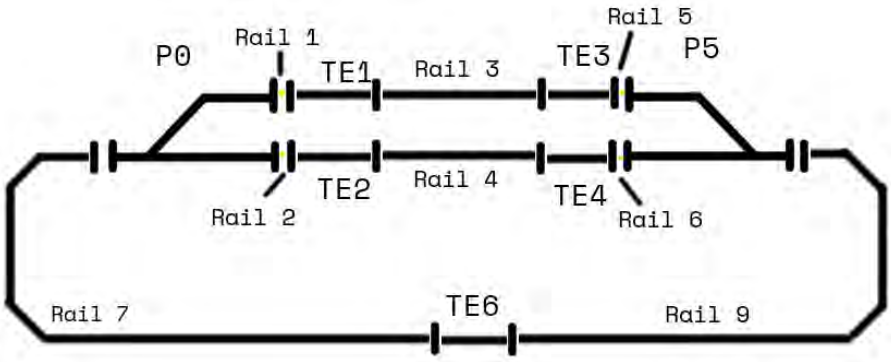

Abbildung 11: Kleines Gleisnetz

Dieses Gleisnetz wird durch den Referenz-CSP_M-Code in Listing 25 repräsentiert. Die Weiche mit der ID 0 ist über `rail.1` mit dem Track-Element eins und über `rail.2` mit Track-Element zwei verbunden. Die Weiche fünf ist mit `rail.5` mit Track-Element drei sowie mit `rail.6` mit dem Track-Element vier verbunden. Weiche null, Track-Element eins und zwei sind über `rail.3` und `rail.4` mit Weiche fünf Track-Element zwei und Track-Element vier verbunden. Mit diesen Gleiselementen ist schließlich Track-Element sechs über `rail.7` und `rail.6` verbunden. Somit mo-

delliert der CSP$_M$-Prozess `RAILWAY_NETWORK` das Gleisnetz aus Abbildung 11.

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

Listing 25: CSP$_M$-Referenzimplementerung vom kleinen Gleisnetz

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!DOCTYPE rail_decl [
  <!ELEMENT network (#PCDATA)>
  <!ELEMENT section (#PCDATA)>
  <!ELEMENT point (#PCDATA)>
  <!ELEMENT crossover (#PCDATA)>
  <!ELEMENT connection (#PCDATA)>
]>
<network id="kleines_gleisnetz">
<section id="t1">
  <connection ref="p0" side="up" />
  <connection ref="t3" side="down" />
</section>
<section id="t2">
  <connection ref="p0" side="up" />
  <connection ref="t4" side="down" />
</section>
<section id="t3">
  <connection ref="t1" side="up" />
  <connection ref="p5" side="down" />
</section>
<section id="t4">
  <connection ref="t2" side="up" />
  <connection ref="p5" side="down" />
</section>
<section id="t6">
  <connection ref="p0" side="up" />
  <connection ref="p5" side="down" />
</section>
<point id="p0">
  <connection ref="t2" side="plus"/>
  <connection ref="t1" side="minus"/>
  <connection ref="t6" side="stem"/>
</point>
<point id="p5">
  <connection ref="t4" side="plus"/>
  <connection ref="t3" side="minus"/>
  <connection ref="t6" side="stem"/>
</point>
</network>
```

Listing 26: XML-Repräsentation des kleinen Gleisnetzes

Wie im Abschnitt 3.1 erläutert, wird nun dieses Gleisnetz durch eine XML-Datei beschrieben. Listing 26 zeigt die XML-Darstellung von dem Gleisnetz. Wie in Abbildung 11 zu sehen, ist Track-Element eins über up mit der Weiche p0 und über down

mit Track-Element t3 verbunden. Dargestellt wird das durch den Section-XML-Knoten in Zeile 10 bis 13 Listing 26. Weiche p0 ist über stem mit Track-Element t6, über plus mit Track-Element t2 und über minus mit Track-Element t1. Weiche p5 ist über stem mit Track-Element t6, über plus mit Track-Element t4, sowie über minus mit Track-Element t3 verbunden. Track-Element t2 ist via up mit der Weiche p0, sowie über down mit t4 verbunden. Track-Element t3, sowie t4 sind beide über die down-Seite mit der Weiche p5 verbunden. Track-Element t3 ist über up mit t1, Track-Element t4 mit t2 über up verbunden. Track-Element t6 verbindet schließlich Weiche p0 und p5 miteinander, über up mit p0, über down mit p5. Diese XML-Datei dient als Eingabedatei für den CSP_M-Code-Generator. Mit dem Aufruf und Flag `-f` wird die Datei eingelesen.

```
$ ./cspCodeGenerator -f kleines_gleisnetz.xml.
```

Der Code-Generator erzeugt dann eine .csp-Datei mit dem Namen des Gleisnetzes, wie in der XML-Datei angegeben (kleines_gleisnetz.csp). Listing 27 zeigt die modifizierte Ausgabe des Code-Generators. Diese wurde wie im Abschnitt 4.1.1 beschrieben bearbeitet, sodass beide CSP_M-Modellierungen dieselben CSP_M-Channel-IDs und Gleiselement ID nutzen, sodass diese vergleichbar sind. Der Code-Generator hat, anders als in der händischen Modellierung aus Listing 25, zwei Gruppen gebildet, die einmal aus Weiche p0, Track Elementen t1, t2 und t6, sowie aus Weiche p5, Track-Element t3 und t4 bestehen.
Diese ergeben über `rail.3`, `rail.4`, `rail.9` den CSP_M-Prozess `SYSTEM_1`, der den Gleisnetz-CSP_M-Prozess `RAILWAY_NETWORK_AUTOGEN` darstellt. Die Verbindungen der Gleiselemente sind in der XML-Datei über Referenzierungen beschrieben. Die Gruppe POINT_p0 stellt die Verbindung von Track-Element t1, t2 und t6 zur Weiche p0 dar. Track-Element t1 ist über `rail.1`, t2 über `rail.2` und t6 über `rail.6` mit Weiche p0 verbunden. Diese Verbindungen passen mit denen aus Abbildung 11 überein. Die zweite Gruppe POINT_p5 repräsentiert die Weiche p5 in Verbindung mit Track-Element t3 und t4. Track-Element t3 ist über `rail.5`, t4 über `rail.6` mit Weiche p5 verbunden. Damit diese beiden Gruppen das Gleisnetz aus Abbildung 11 modellieren, sind diese über `rail.3`, `rail.4`, `rail.9` verbunden und bilden das `SYSTEM_1`. Der nächste Schritt ist die Verifikation des Code-Generators hinsichtlich der Äquivalenz dieser beiden Modellierungen aus Listing 25 und Listing 27. Zur Verifikation werden beide Gleisnetze, einmal die Referenzmodellierung aus Listing 25 und die generierte Version aus Listing 27 auf Failures Refinement getestet. Die Aufstellung der Assertions ist in Listing 28 zu sehen.
Zunächst werden alle nötigen Implementierungen inkludiert, wie die Channel-Definitionen, vordefinierte Modellierungen für Track-Element und Point sowie die beiden Gleisnetz-Modellierungen `RAILWAY_NETWORK` (Referenzmodell) sowie `RAILWAY_NETWORK_AUTOGEN` (generiertes Gleisnetz).

```
1  -------------------------------------------------------------
2  -- This file was generated by the 'XML to CSP' Code Generator
3  -- Version 0.5.2
4  --
5  -- by Felix Brüning, University of Bremen 2020
6  -------------------------------------------------------------
7
8  -- Group elements:
9  -- CSP-ID 1 -> 't1'
10 -- CSP-ID 2 -> 't2'
11 -- CSP-ID 5 -> 't6'
12 -- CSP-ID 0 -> 'p0'
13 POINT_p0 = ( TRACK_ELEMENT(1, 1, 3) [| { rail.1 } |]
14       ( TRACK_ELEMENT(2, 2, 4) [| { rail.2 } |]
15       ( TRACK_ELEMENT(6, 7, 9) [| { rail.7 } |]
16       POINT(0, 7, 1, 2) )))
17
18 -- Group elements:
19 -- CSP-ID 3 -> 't3'
20 -- CSP-ID 4 -> 't4'
21 -- CSP-ID 5 -> 'p5'
22 POINT_p5 = ( TRACK_ELEMENT(3, 3, 5) [| { rail.5 } |]
23       ( TRACK_ELEMENT(4, 4, 6) [| { rail.6 } |]
24       POINT(5, 9, 5, 6) ))
25
26 SYSTEM_1 = POINT_p0 [| { rail.3, rail.4, rail.9 } |] POINT_p5
27
28 RAILWAY_NETWORK_AUTOGEN = SYSTEM_1
```

Listing 27: CSP_M-Code des generierten kleinen Gleisnetzes

Um einen vollständigen Äquivalenzbeweis durch Modellprüfung durchzuführen, wird in beiden Richtungen getestet. Zunächst wird auf Failures Refinement vom generierten Gleisnetz zum Referenzmodell geprüft, danach vom Referenzmodell zum generierten Gleisnetz geprüft. Geprüft wird dies mithilfe von FDR4 in der Version 4.2.7 auf einem 64-Bit Linux Debian 10 System mit einem Intel Core i5-8250U mit vier physischen und acht logischen Prozessor-Kernen sowie 16GB RAM.

```
1 include "channel.csp"
2 include "point.csp"
3 include "track_elem.csp"
4
5 include "autogen_gleisnetz_kl.csp" --RAILWAY_NETWORK_AUTOGEN
6 include "ba_REF_MODEL.csp"         --RAILWAY_NETWORK
7
8 assert RAILWAY_NETWORK_AUTOGEN  [F= RAILWAY_NETWORK
9 assert RAILWAY_NETWORK         [F= RAILWAY_NETWORK_AUTOGEN
```

Listing 28: Äquivalenzbeweis

Das Ergebnis des Äquivalenzbeweises ist in Listing 29 zu sehen. Beide Äquivalenzbeweise endeten erfolgreich (passed). Dadurch ist bewiesen, dass die Referenzmodellierung und die generierten Modellierung äquivalent sind. Für dieses kleine Gleisnetz erzeugt der Code-Generator einen äquivalenten CSP_M-Code, der somit zu weiteren Verifikation genutzt werden kann.

```
RAILWAY_NETWORK_AUTOGEN [F= RAILWAY_NETWORK: Passed
RAILWAY_NETWORK [F= RAILWAY_NETWORK_AUTOGEN: Passed
```

Listing 29: Ergebnis des Äquivalenzbeweises

4.1.3 TEAMOD Gleisnetz

Nach dem Vorgehen aus Abschnitt 4.1.2 soll nun das TEAMOD-Gleisnetz des gleichnamigen Master-, sowie Bachelor-Projektes modelliert werden und der CSP_M-Code automatisch mithilfe des CSP_M-Code-Generators generiert werden. Das Gleisnetz aus Abbildung 12 und Abbildung 22 im Anhang zeigt das Gleisnetz.

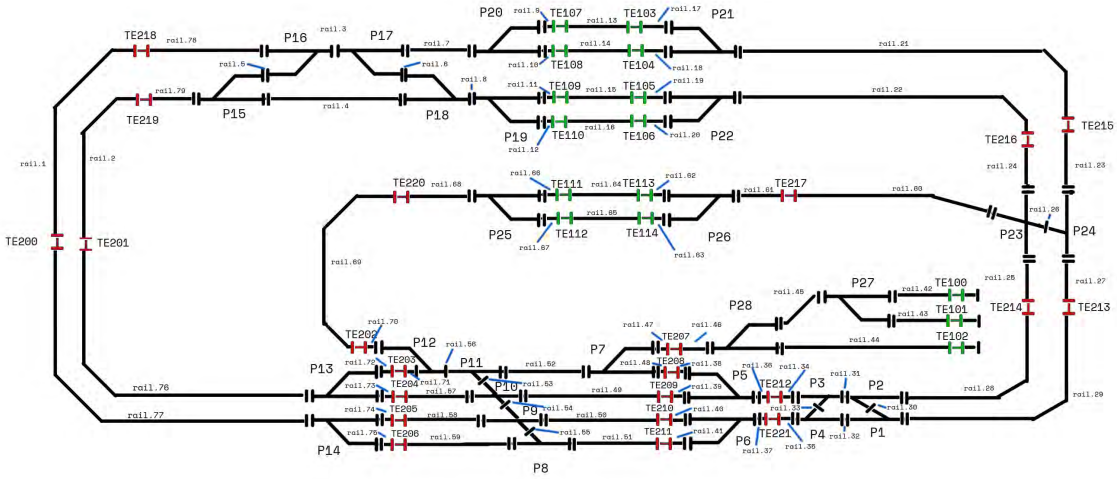

Abbildung 12: TEAMOD-Gleisnetz

Dieses Gleisnetz wurde ebenfalls zuvor in [Brü20] händisch in CSP_M modelliert und dient hierbei als Referenzmodell. Um zu diesem Gleisnetz äquivalenten CSP_M-Code zu generieren, wird dieses Gleisnetz als XML-Datei beschrieben. Dazu wurde händisch eine XML-Datei erstellt, die das Gleisnetz beschreibt. Diese XML-Datei dient im nächsten Schritt als Eingabedatei für den Code-Generator und befindet sich im Anhang unter Listing 84 bis Listing 89. Der Code-Generator erstellt daraus schließlich CSP_M-Code, der dieses Gleisnetz modelliert. Dieser befindet sich im Anhang unter Listing 90 bis Listing 94.

Test auf Failures Refinement. Die Äquivalenz beider Gleisnetze soll nun wie im Abschnitt 4.1.2 durch Failures Refinement und FDR4 bewiesen werden. Zunächst wird das generierte Gleisnetz, wie in Abschnitt 4.1.1 gezeigt, umgeformt, sodass diese vergleichbar sind. Listing 30 zeigt das Aufstellen der Behauptungen für den Äquivalenzbeweis.
RAILWAY_NETWORK_AUTOGEN ist der CSP_M-Prozess, der das generierte Gleisnetz-Modell vom CSP_M-Code-Generator darstellt. RAILWAY_NETWORK ist

das Referenzgleisnetz aus [Brü20]. Jeweils zwei Behauptungen werden von FDR4 ausgeführt, um die Äquivalenz in beiden Richtungen zu zeigen.

```
1 include "channel.csp"
2 include "point.csp"
3 include "track_elem.csp"
4
5 include "autogen_gleisnetz_kl.csp" --RAILWAY_NETWORK_AUTOGEN
6 include "ba_REF_MODEL.csp"         --RAILWAY_NETWORK
7
8 assert RAILWAY_NETWORK_AUTOGEN  [F= RAILWAY_NETWORK
9 assert RAILWAY_NETWORK         [F= RAILWAY_NETWORK_AUTOGEN
```

Listing 30: Äquivalenzbeweis generiertes TEAMOD-Gleisnetz mit dem Referenzgleisnetz

Geprüft werden diese Beweise auf einem Server der Google Cloud Plattform mit einem Intel XEON mit 16 physikalischen CPU-Kernen und 128GB RAM. Nach einiger Zeit bricht FDR4 jedoch aufgrund von ausgeschöpftem RAM ab. Ausgelöst wird das durch Zustandsexplosion. Gelöst werden kann das, indem entweder die Hardwareressourcen aufgestockt werden oder eine andere Verifikationsstrategie herangezogen wird. Da beide Gleisnetze sehr große Modelle darstellen, schaft eine Aufstockung der Hardware wenig Aussicht auf Erfolg. Die Erfolgschancen bei einem größeren Server sind aufgrund der hohen Komplexität der Modellprüfung sehr gering. Um die Äquivalenz zu zeigen, wird daher eine andere Strategie herangezogen. Diese basiert auf die kompositionelle Verifikation.

Kompositionelle Verifikation. Eine weitere Möglichkeit die Äquivalenz großer Modelle zu beweisen ist die kompositionellen Verifikation. [MFH17] Diese Technik beruht auf dem Konzept, große Modelle schrittweise aus kleineren Bausteinen heraus zu verifizieren. Anstatt dabei ein sehr großes Modell wie das des TEAMOD-Gleisnetzes als Ganzes auf Äquivalenz zu prüfen, werden einzelne Sub-Komponenten der beiden Gleisnetze auf Äquivalenz geprüft. Dieser Ansatz beruht auf dem Prinzip des *assume-guarantee reasoning*, auf dem Prinzip der Garantie, dass Sub-Prozesse jeweils eine bestimmte Behauptung erfüllen, wodurch schließlich weiter das Erfüllen einer Behauptung auf höherer Ebene abgeleitet werden kann.

Kompositionelle Verifikation mit einer Prozess-Algebra. Sind zwei große CSP-Modelle gegeben, deren Äquivalenz aufgrund von Zustandsexplosionen nicht bewiesen werden kann, kann hierbei mithilfe der kompositionellen Verifikation die Äquivalenz trotzdem gezeigt werden. [Bur] [WW09] Die kompositionelle Verifikation ist dann besonders einfach, wenn eine eins-zu-eins Korrespondenz zwischen den jeweiligen Sub-Prozessen existiert. Seien beispielsweise S und P zwei sehr große Modelle in Form von CSP-Prozessen, die aufgrund von Zustandsexplosionen nicht im vertretbaren Aufwand verifiziert werden können. Dabei ist S die parallele Komposition von S_1 und S_2, sowie P die parallele Komposition von P_1 und P_2. Die Sub-Prozesse von S und P befinden sich in diesem Fall in einer eins-zu-eins Korrespondenz.

$$S = S_1 \parallel S_2$$
$$P = P_1 \parallel P_2$$

In diesem Fall kann die kompositionelle Verifikation angewendet werden. Zu beweisen ist dann, dass jede Sub-Komponente von S und P zur jeweiligen eins-zu-eins korrespondierenden Komponente äquivalenz im Bezug zur Failures-Refinement sind.

$$assert\, P_1 \sqsubseteq_{\mathrm{F}} S_1$$
$$assert\, S_1 \sqsubseteq_{\mathrm{F}} P_1$$
$$assert\, P_2 \sqsubseteq_{\mathrm{F}} S_2$$
$$assert\, S_2 \sqsubseteq_{\mathrm{F}} P_2$$

Sind diese Behauptungen erfüllt, so kann daraus abgeleitet werden, dass ebenso die folgenden Behauptungen gelten.

$$assert\, P \sqsubseteq_{\mathrm{F}} S$$
$$assert\, S \sqsubseteq_{\mathrm{F}} P$$

Das Prinzip soll nun zum Beweisen der Äquivalenz der Referenzimplementierung des TEAMOD-Gleisnetzes mit dem generierten TEAMOD-Gleisnetz genutzt werden. Zunächst ist festzustellen, dass sowohl das Referenzmodell als auch das generierte Modell in einzelnen Sub-Komponenten untergliedert sind. Diese Tatsache ist wichtig für die kompositionelle Verifikation, da diese sonst nicht anwendbar ist. Im nächsten Schritt werden nun Paare von eins-zu-eins korrespondierenden Sub-Komponenten gesucht, sodass jeweils eine Sub-Komponente des Referenzmodells mit einer Sub-Komponente des generierten Modells und in anderer Richtung auf Failures Refinement geprüft werden kann. Dabei stehen die Sub-Komponenten aus Listing 31 der beiden Gleisnetz-Modelle in eins-zu-eins Korrespondenz.

```
1 assert GRP_11 [F= POINT_p13
2 assert GRP_15 [F= POINT_p14
3 assert GRP_08 [F= POINT_p15
4 assert GRP_05 [F= POINT_p16
5 assert GRP_20 [F= POINT_p25
6 assert GRP_23 [F= POINT_p20
7 assert GRP_25 [F= POINT_p19
8 assert GRP_26 [F= POINT_p22
9 assert GRP_13 [F= POINT_p7
```

Listing 31: Sub-Prozesse von dem Referenzmodell und dem generierten Modell in eins-zu-eins Korrespondenz

Zwei in eins-zu-eins Korrespondenz stehende Sub-Komponenten aus Listing 31 sind in Listing 32 zu sehen. Hierbei ist einmal der CSP_M-Prozess `GRP_23` des Referenzgleisnetzes und `POINT_p20` aus dem generierten Gleisnetz zu sehen. Die Äquivalenz der beiden Sub-Komponenten des TEAMOD-Gleisnetzes konnte mithilfe von FDR4 bewiesen werden.

```
GRP_03 = (POINT(20, 7, 9, 10)
          [| { rail.9  } |] TRACK_ELEMENT(107, 9, 13))

GRP_23 = (GRP_03 [| { rail.10 } |] TRACK_ELEMENT(108, 10, 14))

POINT_p20 = ( TRACK_ELEMENT(107, 9, 13) [| { rail.9 } |]
              ( TRACK_ELEMENT(108, 10, 14) [| { rail.10 } |]
              POINT(20, 7, 9, 10) ))
```

Listing 32: Sub-Komponenten der Gleisnetze

Insgesamt konnten alle Behauptungen aus Listing 31 mithilfe von FDR4 bewiesen werden. Der vollständige Verifikations-Code ist im Anhang unter Listing 95 zu sehen. Die Sub-Komponenten, die erfolgreich auf Äquivalenz geprüft werden konnten, sind gelb in Abbildung 13 gefärbt.

Abbildung 13: Erfolgreich auf Äquivalenz geprüfte Gleisabschnitte

Jedoch war es nicht möglich, das gesamte Gleisnetz durch kompositionelle Verifikation zu verifizieren. Zum Einen stehen nicht alle Sub-Prozesse des Referenzgleisnetzes in eins-zu-eins Korrespondenz zu den Sub-Prozessen des generierten Gleisnetzes. Aufgrund der alphabetisierten parallelen Komposition der Sub-Prozesse war es zudem nicht weiter möglich, das Gleisnetz in Sub-Prozessen zu unterteilen und schließlich zu verifizeren. Da nicht jede Sub-Komponente in eins-zu-eins Korrepondenz exisiteren und aufgrund des Problems, dass aus der Korrektheit der Einzelelemente nicht direkt die Korrektheit der alphabetisierten parallelen Komposition folgt, wird im nächsten Schritt versucht, durch Aufstellen geeigneter Properties die Äquivalenz dennoch zu zeigen.

Property Katalog Eine weitere Herangehensweise ist das Aufstellen eines Property-Kataloges. Ein Property-Katalog besteht aus einer Menge von Eigenschaften, die für den Äquivalenztest benötigt werden. Dabei besteht dieser aus so vielen Eigenschaften (Properties), dass die Anzahl dessen die Äquivalenz annähern kann. Das in

dieser Arbeit angewendete Verfahren konzentriert sich dabei auf kritische Sektionen im Gleisnetz und nutzt diese als Grundlage zum Aufstellen von Properties. Kritische Sektionen sind in dem Gleisnetz genau die Stellen, die dazu führen können, dass die Äquivalenzbehauptungen wiederlegt oder bewiesen werden können.
Damit sowohl das generierte als auch das Referenzgleisnetz äquivalent sein können, müssen jeweils alle Gleiselemente korrekt miteinander verbunden sein, sodass diese das reale Gleisnetz widerspiegeln. In diesem Fall sind das genau die Verbindungen zwischen Gleiselementen, die dafür sorgen, dass der Zug den korrekten Weg nimmt. Kritische Stellen im Gleisnetz sind somit genau die Punkte, an denen der Folgepfad des Zuges im Gleisnetz bestimmt wird. Weichen im Gleisnetz sind demnach kritische Punkte, an denen sich beide Gleisnetze unterscheiden können.
Zur Verifikation wird dabei der Ansatz gewählt, dass jede Weiche befahren wird und der Zug schließlich das richtige Zielgleis erreichen soll. Dazu werden Routen im Gleisnetz ausgewählt, die mehrere Weichen abdecken. Danach werden Testfälle in Form von Zug-Traces, Pfad von der Startposition zur Endposition, erzeugt, die diese Route wiederspiegeln. Zunächst befährt ein Zug eine Route mit korrekten Weichenstellungen. Das Ergebnis des Tests sollte erfolgreich sein. Danach wird an jeder Weiche ein Fehler erzeugt und beobachtet, ob der Zug entgleisen kann. Wenn dann genau das generierte Gleisnetz und das Referenzgleisnetz exakt gleich reagieren, so ist diese Route in beiden Gleisnetzen in diesem Fall äquivalent.

```
-- # ====================================
-- # ===== T218 -> T215 via T107, T103  =
-- # = P16, P17, P20, P21               =
-- # ====================================
-----------------------------------------------------
TRAIN_0 = movement_auth.0 -> rail.1 -> rail.78 -> rail.3 -> rail.7
      -> rail.9 -> rail.13 -> rail.17 -> rail.21 -> STOP

IXL_0 = reqsec.16 -> set.16.str -> reqsec.17 -> set.17.str -> reqsec.20
      -> set.20.crss -> reqsec.21 -> set.21.crss -> movement_auth.0 -> STOP

TEST0 = TRAIN_0 [| { movement_auth.0 } |] IXL_0
SYSTEM0 = RAILWAY_NETWORK [| {| rail, reqsec, set, release |} |] TEST0
assert STOP [T= SYSTEM0 \ diff(Events, {| derailing |})
-----------------------------------------------------
```

Listing 33: Testfall bei korrekten Weichenstellungen

Listing 33 zeigt hierbei den Testfall mit korrekten Weichenstellungen. TRAIN_0 ist der CSP_M-Prozess, der den zu befahrenden Pfad der Route in Form eines Zuges beschreibt. IXL_0 ist das Stellwerk, welches die Weichen in die richtige Stellung versetzt. Sobald alle Weichen gestellt wurden, wird mit dem Event `movement_auth.0` die Fahrterlaubnis erteilt, woraufhin der Zug die Route befährt. Der CSP_M-Prozess TEST0 beschreibt diesen Testfall. IXL_0 und TRAIN_0 werden über `movement_auth.0` miteinander synchronisiert. Das System dieses Testfalls wird mit dem CSP_M-Prozess SYSTEM0 ausgedrückt, der den Testfall TEST0 mit dem Gleisnetz RAILWAY_NETWORK synchronisiert, sodass der Testfall auf das Gleis-

netz angewendet werden kann. Die finale Verifikationsbehauptung prüft, ob STOP von dem Prozess SYSTEM0 Trace-Verfeinert wird, wodurch alle CSP_M-Events verborgen werden außer `derailing`. Dieses Signal wird genau dann ausgelöst, wenn der Zug die Weiche entgegengesetzt der Weichenstellung befährt und somit entgleisen kann.
Wenn ein Zug entgleist, dann folgt, dass die Weichenstellungen nicht korrekt für den Zugpfad sind. Reagieren beide Gleisnetze identisch, erzeugen also dasselbe Fehler-Event an derselben Stelle, so sind diese in dem Fall äquivalent. Das Ergebnis bei beiden Gleisnetzen war erfolgreich, wodurch in diesem Fall gezeigt wurde, dass diese Route in beiden Gleisnetzen äquivalent ist.

```
1  -- # Fehler P16
2  ------------------------------------------------
3  TRAIN_2 = movement_auth.2 -> rail.1 -> rail.78 -> rail.3 -> rail.7
4        -> rail.9 -> rail.13 -> rail.17 -> rail.21 -> STOP
5
6  IXL_2 = reqsec.16 -> set.16.crss -> reqsec.17 -> set.17.str -> reqsec.20
7        -> set.20.crss -> reqsec.21 -> set.21.crss -> movement_auth.2 -> STOP
8
9  TEST2 = TRAIN_2 [| { movement_auth.2 } |] IXL_2
10 SYSTEM2 = RAILWAY_NETWORK [| {| rail, reqsec, set, release |} |] TEST2
11 assert STOP [T= SYSTEM2 \ diff(Events, {| derailing |})
12 ------------------------------------------------
```

Listing 34: Testfall mit falschen Weichenstellungen

Um nun Tests bei falschen Weichenstellungen auszuführen, ist es nötig zu prüfen, ob sich beide Gleisnetze in Ausnahmesituationen gleich verhalten. Dazu muss geprüft werden, ob bei falscher Weichenstellung sich beide Gleisnetze gleich verhalten. Dazu ist zu prüfen, ob jeder Weichenfehler in beiden Gleisnetzen erkannt werden kann und an derselben Stelle auftreten. Für diese Tests wird der Testfall aus Listing 33 herangezogen und für jede Weiche in falscher Position ein neuer Testfall erzeugt.
Listing 34 zeigt den Testfall, bei dem die Weiche P16 falsch gestellt wurde. Hierbei wird diese vom Stellwerk in Position cross versetzt (abbiegend), korrekt wäre in Position straight (geradeaus). Wie in Abbildung 12 zu sehen, würde der Zug nun entgleisen, wodurch ein Event `derailing.16` erwartet wird.
Bei beiden Gleisnetzen wurde wie erwartet das Fehler-Event `derailing.16` erzeugt, wie in Listing 35 zu sehen. Der Testfall

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

beschreibt dabei den Testfall mit dem Referenzmodell,

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

mit dem generierten Gleisnetz. Durch den Hiding-Operator werden hierbei alle Events außer `derailing` verborgen. In diesem Fall reagieren beide Gleisnetze äquivalent. Die Tests für jede Weiche dieser Route wurden nach diesem Schema ausgeführt. Alle Testfälle endeten bei jeden der beiden Gleisnetze identisch. Somit ist diese Route in diesem Fall bei beiden Gleisnetzen äquivalent.

```
1 Welcome to FDR Version 4.2.7 copyright 2016 Oxford University Innovation
    Ltd. All Rights Reserved.
2 License: Academic license for non-commercial use only
3 STOP [T= REF_SYSTEM \ diff(Events, {|derailing|}):
4   Log:
5     Result: Failed
6     Trace: <...>
7     Error Event: derailing.16
8 STOP [T= SUT_SYSTEM \ diff(Events, {|derailing|}):
9   Log:
10    Result: Failed
11    Trace: <...>
12    Error Event: derailing.16
```

Listing 35: Testergebnis des Fehlers an Weiche P16

Für den Property-Katalog werden insgesamt 346 Testfälle erzeugt. Es wurden jeweils Routen getestet, die nicht identisch sind und sich jeweils nur teilweise überschneiden. Zudem wird jede Weiche mindestens einmal befahren und dabei aus allen Richtungen angefahren, sodass jeder Ausgang getestet werden kann. Die Tests sind ebenfalls so konzipiert, dass es zudem Testfälle gibt, die je zwei Routen halb überdeckt. Dadurch werden auch transitive Fälle abgedeckt. Die Testfälle als CSP$_M$-Datei befindet sich auf dem mitgelieferten Speichermedium.

Durch die hohe Anzahl an Testfällen und durch die Testabdeckung jeder kritischen Sektion im Gleisnetz kann daraus gefolgert werden, dass sowohl das generierte Gleisnetz als auch das Referenzgleisnetz äquivalent sind. Durch das Abdecken der Randfälle im Gleisnetz durch die Tests des Property-Kataloges kann sichergestellt werden, dass diese äquivalent sind. Um diese schon hohe Testgenauigkeit weiter zu erhöhen, können weitere Testfälle erzeugt werden. Da jedoch jede kritische Sektion mit mehreren Testfällen abgedeckt ist, ist dies nicht mehr nötig und die Äquivalenz ist gezeigt.

4.2 CSP$_M$-Trace-Generator und Gleisnetz-Interpreter

Als zusätzliche Verifikationsmethode für den CSP$_M$-Code-Generator wird in diesem Abschnitt der CSP$_M$-Trace-Generator und CSP$_M$-Gleisnetz Interpreter vorgestellt. Diese ergänzen die zuvor vorgestellten Verifikationsmethoden mit einer alternativen Methode. Der CSP$_M$-Trace-Generator und Gleisnetz-Interpreter werden genutzt, um Traces aus einer CSP$_M$-Spezifikation zu generieren und auf Äquivalenz mit einem zu testenden Gleisnetz zu prüfen. Dieses in Java implementierte Programm nimmt einen zusätzlichen Bestandteil der Verifikation des Code-Generators ein. Dadurch ist es möglich, große Gleisnetze auf Äquivalenz zu prüfen, ohne vollständiges Model-Checking auf einen sehr großen Zustandsraum auszuführen. Dieses Tool verspricht dabei schnelle Laufzeiten und ein schnelles, aussagekräftiges Ergebnis.

Die Grundlegende Softwarearchitektur ist in Abbildung 14 gezeigt.

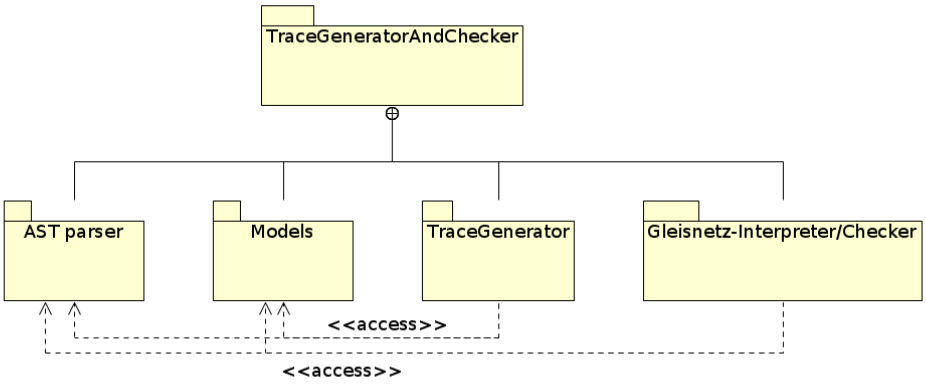

Abbildung 14: Softwarearchitektur CSP_M-Trace-Generator und Gleisnetz-Interpreter

In den folgenden Abschnitten wird das Programm genauer eingeführt. In Abschnitt 4.2.1 werden die Konventionen erläutert, in welcher Form die CSP_M-Datei vorliegen muss, damit der Trace-Generator diese verarbeiten kann. Ebenso wird die Struktur der Trace-Datei vorgestellt.In Abschnitt 4.2.2 wird das Tool CSP_{MF} genauer erläutert, welches das Einlesen und verarbeiten der CSP_M-Spezifikation ermöglicht und den Syntaxbaum in Form einer XML-Datei ausgibt. Der eigentliche Trace-Generator wird schließlich in Abschnitt 4.2.4 erläutert. Der Gleisnetz-Interpreter wird dann in Abschnitt 4.2.5 erläutert, sowie die Verifikation der Gleisnetze wird in Abschnitt 4.2.6 beschrieben.

4.2.1 Konventionen

CSP_M-Datei. Damit der CSP_M-Trace-Generator und Gleisnetz-Interpreter zwei Gleisnetze korrekt auf Äquivalenz prüfen kann, muss die CSP_M-Datei in einer bestimmten Form vorliegen. Dabei ist es wichtig, dass diese Datei alle nötigen Modelle inkludiert und vollständig ist, dass auch FDR4 diese Datei verarbeiten könnte. Der CSP_M-Parser CSP_{MF} verlangt jedoch für das Einlesen, dass es einen Main-CSP_M-Prozess gibt. Somit muss der Gleisnetz-Hauptprozess als MAIN deklariert werden, wie in der umgeformten Spezifikation des kleinen Gleisnetzes in Listing 36 zu sehen.

```
1 include "point.csp"
2 include "track_elem.csp"
3
4 MAIN = (( ((POINT(0,7,1,2) [|{rail.1}|] TRACK_ELEMENT(1,1,3))
5         [|{rail.2}|] TRACK_ELEMENT(2,2,4))
6         [|{ rail.3, rail.4 }|]
7         ((POINT(5,9,5,6) [|{rail.5}|] TRACK_ELEMENT(3,3,5))
8         [|{rail.6}|] TRACK_ELEMENT(4,4,6)) )
9         [|{rail.7, rail.9}|] TRACK_ELEMENT(6,7,9))
```

Listing 36: Umgeformte Spezifikation des kleines Gleisnetzes

Tracefile. Die vom CSP_M-Trace-Generator generierte `tracefile.txt` beschriebt alle möglichen Traces in dem Gleisnetz. Die `tracefile.txt` des Gleisnetzes aus Listing 36 ist gezeigt in Listing 37.

```
5:c;0:c;1;3;5;9;7;
5:s;0:s;2;4;6;9;7;
5:c;0:c;7;9;5;3;1;
5:s;0:c;7;9;6;4;2;
```

Listing 37: Trace-Datei des kleinen Gleisnetzes

Jede Zeile entspricht einem Trace in dem Gleisnetz. Unterteilt sind Elemente in dem Trace mit einem Semikolon. Dabei werden die Elemente in zwei Arten unterschieden. Der Ausdruck `5:c` bedeutet, dass die Weiche mit der ID 5 in die Cross-Richtung für den Trace versetzt werden muss. Der Ausdruck `5:s` bedeutet somit, dass Weiche 5 in die Straight-Position versetzt werden muss. Die Elemente, die nur aus einem Element bestehen, wie `9`, beschreiben ein `rail`-Event auf dem Gleisnetz, wobei die Reihenfolge der aufgezählten `rail`-Elemente wichtig für die Bedeutung des Traces ist. Diese beschreiben den eigentlichen Pfad des Zuges auf dem Gleisnetz.

4.2.2 CSP_{MF}-Tool

Das CSP_{MF}-Tool ist als Frontend für den ProB Model Checker und Animator an der Universität Düsseldorf entwickelt worden. Diese in Haskell implementierte Programm bietet die Möglichkeit, CSP_M-Modelle einzulesen und den Syntaxbaum in Form einer XML-Datei auszugeben. [LF08]

```
<?xml version="1.0"?>
<Module>
  <moduleDecls>
    <list>
      <FunBind>
        <Ident unIdent="POINT"/>
        <list>
          <FunCase>
            <list>
            </list>
          </FunCase>
        </list>
      </FunBind>
      <FunBind>
        <Ident unIdent="MARKED_P"/>
        <list>
          <FunCase>
            <list>
            </list>
          </FunCase>
        </list>
      </FunBind>
    </list>
  </moduleDecls>
</Module>
```

Listing 38: Struktur des Syntaxbaums von CSP_{MF}

Der Syntaxbaum von CSP_{MF} hat die Struktur, wie in Listing 38 zu sehen. Durch den XML-Tag `Module` und `moduleDecls` werden die einzelnen sogenannten CSP-Module, die CSP_M-Implementierungen, beschrieben. Diese sind in diesem Fall die einzelnen CSP-Module, die speziell in CSP_M modelliert werden können. Die sich darin befindliche `list` listet dabei alle CSP_M-Prozesse auf, die in diesem Modul existieren. Der XML-Tag `FunBind` deklariert eine CSP_M-Prozess-Deklaration. Der sich darin befindliche Tag `Ident` beschreibt den Namen des Prozesses. Im Listing 38 sind die Prozesse POINT und MARKED_P dargestellt. Innerhalb einer `FunBind`-Umgebung werden alle Pattern der CSP_M-Funktion unter dem Tag `FunCase` beschrieben. Die innerhalb dieses Tags beschriebene Pattern sind in Listing 39 gezeigt. Wie in Listing 39 zu sehen, wird hier eine Prefix-Expression, also Events und ein Zustandswechsel beschrieben. Die Informationen, die nur ProB dienen, wurden hierbei entfernt, da diese dafür nicht relevant sind. Diese sind Parser-Anweisungen und Informationen über Zeilen im CSP_M-Code. Über `DotTuple` werden Events wie hier `reqsec` beschrieben. Der tag `CallFunction` sagt hierbei aus, dass nach dem Auftreten des Events in den Zustand deklariert in `CallFunction` gewechselt wird.

```
<PrefixExp>
  <DotTuple>
    <list>
      <Var>
        <Ident unIdent="reqsec"/>
      </Var>
      <Var>
        <Ident unIdent="id"/>
      </Var>
    </list>
  </DotTuple>
  <list/>
  <CallFunction>
    <Var>
      <Ident unIdent="MARKED_P"/>
    </Var>
    <list>
      <list>
        <Var>
          <Ident unIdent="id"/>
        </Var>
        <Var>
          <Ident unIdent="s0"/>
        </Var>
        <Var>
          <Ident unIdent="s1"/>
        </Var>
        <Var>
          <Ident unIdent="s2"/>
        </Var>
      </list>
    </list>
  </CallFunction>
</PrefixExp>
```

Listing 39: CSP-Prozess-Pattern als Syntaxbaum in der XML-Datei

Mithilfe dieser Darstellung werden die Modellierungen von Point, Crossing und Track_Element dargestellt. Neben den Spezifikationen werden ebenfalls die synchronisierten Prozesse eingelesen, die ebenfalls im Syntaxbaum beschrieben werden. Dazu zählt das Gleisnetz, welches als CSP_M-Prozess beschrieben wird, der einzelne Gleiselemente synchronisiert.
Eine Gleisnetzspezifikation wird innerhalb der XML-Datei wie folgt dargestellt. Listing 40 zeigt die Struktur.

```
<PatBind>
  <VarPat>
    <Ident unIdent="NETWORK"/>
  </VarPat>
    <Parens>
      <ProcSharing>
        <!---- Declaration of synchronisations----->
      </ProcSharing>
    </Parens>
</PatBind>
```

Listing 40: Darstellung des Gleisnetzes durch die XML-Datei

Innerhalb von `moduleDecls` wird das synchronisierte Gleisnetz innerhalb einer `PatBind` Umgebung definiert. Zunächst wird über `Ident` der Name des Gleisnetzes angegeben. Innerhalb des `Parens`- und `ProcSharing`-Blocks werden nun die Synchronisationsbedingungen deklariert. Ein Beispiel für eine Synchronisationsbedingung ist in Listing 42 dargestellt.
Diese ist in drei Teilen, der `SetExp`, sowie zwei CallFunction unterteilt. Dabei repräsentiert `SetExp` die Synchronisationsbedingung. In diesem Fall werden zwei Gleiselemente auf das Event `rail.1` synchronisiert. In der `SetExp` ist der CSP_M-Kanal-Name durch `Ident` angegeben und das explizite Event auf dem Kanal durch `IntExp`. Dieses Event synchronisiert dabei zwei CSP_M-Prozesse, die durch die beiden nachfolgenden CallFunction-Abschnitte deklariert werden. Durch `Ident` wird der Name des Prozesses, hier einmal POINT und TRACK_ELEMENT, deklariert. Jede Deklaration besitzt zusätzlich eine Liste von Parametern, die durch `IntExp` ausgedrückt werden (siehe Weichen-Modellierung Listing 81 und 82 sowie Track-Element Implementierung in Listing 83). Dadurch wird eine Synchronisationsbehauptung aufgestellt sowie die jeweiligen Prozesse direkt implementiert durch die Liste der expliziten Parameter. In CSP_M entspricht der Ausschnitt aus der XML-Datei dem CSP_M-Prozess `P` aus Listing 41.

```
P = POINT(0, 7, 1, 2) [| { rail.1 } |] TRACK_ELEMENT(1, 1, 3)
```

Listing 41: CSP_M-Code des Ausschnittes aus Listing 42

In der XML-Datei, die den Syntaxbaum beschreibt, gibt es mehrere Sektionen von `ProcSharing`, jedoch werden diese nicht als Liste gespeichert. Diese werden verschachtelt in einer Baumstruktur gespeichert, sodass die Struktur dieser Sektionen

der Reihenfolge der aufeinander synchronisierten Prozesse innerhalb der CSP-Datei entspricht.

```
<ProcSharing>
  <SetExp>
    <Var>
      <Ident unIdent="rail"/>
    </Var>
    <IntExp>
      <Integer val="1"/>
    </IntExp>
  </SetExp>
  <CallFunction>
    <Var>
      <Ident unIdent="POINT"/>
    </Var>
    <list>
      <list>
        <IntExp>
          <Integer val="0"/>
        </IntExp>
        <IntExp>
          <Integer val="7"/>
        </IntExp>
        <IntExp>
          <Integer val="1"/>
        </IntExp>
        <IntExp>
          <Integer val="2"/>
        </IntExp>
      </list>
    </list>
  </CallFunction>
  <CallFunction>
    <Var>
      <Ident unIdent="TRACK_ELEMENT"/>
    </Var>
    <list>
      <list>
        <IntExp>
          <Integer val="1"/>
        </IntExp>
        <IntExp>
          <Integer val="1"/>
        </IntExp>
        <IntExp>
          <Integer val="3"/>
        </IntExp>
      </list>
    </list>
  </CallFunction>
</ProcSharing>
```

Listing 42: Deklaration einer Synchronisationsbedingung

4.2.3 Parsieren des Syntaxbaums

Eine zentrale Aufgabe sowohl des CSP_M-Trace-Generators, als auch des Gleisnetz-Interpreters ist es, die von dem CSP_{MF}-Tool erzeugte XML-Datei einzulesen und zu verarbeiten. Dazu wird der DOM-Parser von der W3C als Programmbibliothek verwendet, die, ähnlich wie die LibXml2, durch wenige Funktionsaufrufe eine XML-Datei parsiert. Der DOM-Parser stammt aus der Programmbibliothek `javax.xml.parsers`.

```
1  public class CSPProcess implements Serializable
2  {
3    // ----------------------------------------
4    // Private Attributes
5    // ----------------------------------------
6    private String                  name;
7    private String                  rename;
8    private CSPPattern              pattern;
9
10   // ----------------------------------------
11   // Public Attributes
12   // ----------------------------------------
13   public int                      level = -1;
14   public List<String>             params;
15   public Map<String, Integer>     paramsMap;
16   public boolean                  instanceWasMade;
17   public boolean                  visited;
18   public List<Transition>         transitions;
19 }
```

Listing 43: CSPProcess Java Klasse

Die einzulesene XML-Datei des Syntaxbaums wird intern in geeignete Datenstrukturen gespeichert. Grundsätzlich wird jeder CSP_M-Prozess als Java CSPProcess-Objekt gespeichert. Listing 43 zeigt die CSPProcess-Java-Klasse. Jeder CSP_M-Prozess besitzt einen Namen, der diesen beschreibt, wie CROSSING, POINT oder MARKED_P. Das Attribut `pattern` vom Typ CSPPattern beschreibt alle in dem Prozess existierenden Pattern. Gibt es beispielsweise eine Fallunterscheidung durch External Choice innerhalb eines CSP_M-Prozesses, so wird das durch das CSPPattern-Objekt beschrieben. Die Liste vom Typ String `params` sind alle Parameter eines CSP_M-Prozesses, wie diese definiert wurden. Ein CSP_M-Prozess `POINT(id, s0, s1, s2)` hat die Parameter id, s0, s1 und s2, die in dieser Liste gespeichert werden. Folglich speichert `paramsMap` die Implementierung des Prozesses und ordnet den Parametern realen Werten zu. In diesem Anwendungsfall sind die Parameter immer vom Typ integer, wodurch hierbei die Map dementsprechend Strings auf Integer abbildet. Andere Typen als Parameter sind auch möglich, jedoch für diesen Anwendungsfall irrelevant.

Durch `instanceWasMade` wird gespeichert, ob diese Instanz bereits erstellt wurde, sodass beispielsweise Instanzen von CSP_M-Prozessen nicht mehrmals erstellt werden. Das Attribut `visited` und `transitions` sind für den Trace-Generator und

Gleisnetz-Interpreter nötig, was im Folgenden genauer beschrieben wird. Das Attribut `level` ist später wichtig für das Traversieren der einzelnen Zustandsautomaten. Die Java-Klasse CSPPattern beschreibt die Pattern eines CSP_M-Prozesses. Diese besteht aus den einzelnen Funktionen eines CSP_M-Prozesses. Listing 44 beschreibt diese Klasse.

```
1 public class CSPPattern implements Serializable
2 {
3   public List<CSPFunction>    functions;
4   public Choice               choice = Choice.NONE;
5   public List<Transition>     transitions;
6 }
```

Listing 44: CSPPattern Java-Klasse

Die Liste der CSPFunction, `functions` deklariert eine Liste, die alle CSP_M-Funktionen innerhalb eines Patterns von einem CSP_M-Prozess speichert. Das Attribut `choice` deklariert den Typ der Fallunterscheidung. Der Typ Choice ist hierbei ein Enumerations-Typ mit den möglichen Werten `Choice.None`, der keine Fallunterscheidung beschreibt, `Choice.EXTERNAL_CHOICE`, der denn external choice und `Choice.INTERNAL_CHOICE`, der den internal-choice Fall beschreibt, mit dem der Fall zwischen allen Funktionen unterschieden wird. Für diesen Anwendungsfall ist das ausreichend, da innerhalb eines CSP_M-Prozesses auch verschiedene Fallunterscheidungen existieren können. Da es für diesen Anwendungsfall genügt, wurde sich jedoch nur auf eine mögliche Variante festgelegt. Real kann es auch sein, dass es mehrere Fallunterscheidungen in einem Pattern gibt. Die Liste der Transition, `transitions`, wird später genauer erläutert.
Listing 45 zeigt die Java Klasse CSPFunction, die eine CSP_M-Funktion innerhalb eines CSP_M-Prozesses beschreibt. Eine CSP_M-Funktion ist im CSP_M-Kontext eine Folge von Events und anschließendem Zustandswechsel.

```
1 public class CSPFunction implements Serializable
2 {
3   public LinkedList<CSPEvent> trace;
4   public CSPFunctionCall      targetCall;
5   public CSPProcess           next;
6   public CSPProcess           prev;
7   public Transition           transition;
8 }
```

Listing 45: CSPFunction Java-Klasse

Die Java-LinkedList `trace` vom Typ CSPEvent beschreibt geordnet die Folge aller CSP_M-Events innerhalb einer Funktion, wie in Listing 47 zu sehen. Diese enthält die Liste der Events, die von der Funktion erzeugt oder darauf gehört werden. Das Attribut `next` und `prev` beschreiben den Folge- beziehungsweise Ursprungs-CSP_M-Prozess der Funktion. Das Attribut `transition` vom Typ Transition wird später genauer erläutert.

Das Attribut `targetCall` vom Typ CSPFunctionCall speichert den Namen des Folge-CSP_M-Prozesses wie STOP oder POINT. Listing 46 zeigt die Klasse.

```
1 public class CSPFunctionCall implements Serializable
2 {
3   public String           targetFunctionName = "";
4   public List<String>     params;
5   public List<Integer>    intParams;
6 }
```

Listing 46: CSPFunctionCall Java-Klasse

Hierbei beschreibt `targetFunctionName` den Namen sowie `params` die geordnete Liste der Parameter des Folgeprozesses. Für diesen Anwendungsfall hingegen beschreibt `intParams` die geordnete Liste der Parameter der CSP_M-Prozess-Implementierung.
Die Trace-Events einer Funktion wird durch eine LinkedList (Listing 45) vom Typ CSPEvent beschrieben, wie in Listing 47 gezeigt.

```
1 public class CSPEvent implements Serializable
2 {
3   public String       channel;
4   public List<String> params;
5 }
```

Listing 47: CSPEvent Java-Klasse

Das Attribut `channel` beschreibt den Kanal sowie `params` die Liste der Parameter des Events. Bei dem Event `rail.5` ist rail der Kanalname und die Fünf wird in der List der `params` gespeichert.
Das Parsieren der XML-Datei erfolgt danach in zwei Schritten: Zunächst werden alle Prozesse parsiert, wie POINT, CROSSING und TRACK_ELEMENT. Diese sind in diesem Anwendungsfall zunächst die Modellspezifikationen. Diese werden dann als CSPProcess-Java-Klasse gespeichert. Danach erfolgt das Einlesen und Verarbeiten der CSP_M-Prozess-Implementierungen. Der Ablauf und Aufrufhierarchie während des Parsierungs-Prozesses ist durch Abbildung 20 dargestellt. Diese Abbildung befindet sich aufgrund der Größe im Anhang.

Einlesen der Modelle. Üblicherweise werden zu Beginn einer Spezifikation Modellspezifikationen inkludiert. Im ersten Schritt werden somit alle durch `include` inkludierten Modelle parsiert. In diesem Fall sind das genau die in CSP_M modellierten Modelle von Point, Crossing und Track-Element. Dazu dient die Java-Methode `parseFunBind()`. XML-Schlüsselwörter mit `FunBind` in der XML-Datei dienen zur Deklaration eines CSP_M-Prozesses und entspricht der Klassendefinition aus Listing 43. Ein `FunBind` Abschnitt in der XML-Datei beschreibt zunächst den Namen des Prozesses mit `Ident` sowie alle Pattern und Parameter des Prozesses mit `FunCase`. Nachdem der Name des CSP_M-Prozesses gelesen und als Name des neu erstellten CSPProcess gesetzt wurde, werden alle Parameter des Prozesses parsiert. Dieser Schritt wird mithilfe der Methoden `parseParams()` durchgeführt.

Danach werden alle Pattern des Prozesses parsiert. Die Pattern werden dann als Java-Klasse CSPPattern gespeichert. Somit werden nun alle Kindknoten des `Fun2` XML-Knotens unterhalb von `FunCases` mit der Methode `parseFunctions()` gelesen. Durch das Schlüsselwort `Fun2` wird zunächst die Art des Pattern-Matchings angegeben, wie external oder internal choice. Nun können entweder Events erzeugt und ein Zustandswechsel im klassischen Sinne erfolgen, oder es existiert ein Lambda-Ausdruck. Im Fall, dass ein Lambda-Ausdruck gegeben ist, wird mit der Methode `parseLambdaExpression()` diese gelesen. Diese Methode ist in mehreren Teilen gegliedert und wird anhand des folgenden Listings 48 genauer erläutert.

```
1 ([] x : {s0, s1, s2} @ rail.x -> crash.id -> STOP)
```

Listing 48: Parsieren eines Lambda-Ausrucks

Zu sehen ist ein Lambda-Ausdruck in CSP_M. Dieser ist unterteilt in einer Prefix-Expression `rail.x -> crash.id -> STOP`, einer Menge, über die der Lamda-Ausdruck definiert ist, das Range-Enum (`{s0, s1, s2}`), sowie den Choice-Reprica-Operator `[]`, hier external choice. Nun wird zunächst die Definitionsmenge des Ausdrucks mithilfe der Methode `parseLambdaRangeEnums` parsiert, die die genauen Elemente dieser Menge mit `parseVar` ermittelt. Danach wird die Prefix-Expression parsiert mit der Methode `createCSPFunctionFromPrefixExp()`, die wiederum alle Prefix-Expression mithilfe der Methode `parsePrefixExpression()` parsiert. Letztere Methode ist nötig, da es später bei Ausdrücken ohne Lambda-Ausdrücken vorkommen kann, dass es mehrere Prefix-Expressions geben kann. Danach wird eine Prefix-Expression genauer untersucht. Diese besteht aus Events sowie aus einem Folgezustand (Folgeprozess). Ein Event auf einem Kanal kann dabei über den Punktoperator genauer spezifiziert werden, wodurch nur ein bestimmter Wert wie `rail.2`, auf einem Kanal erzeugt wird. Diese Deklaration ist spezifiziert durch das XML-Schlüsselwort `DotTuple`. Somit parsiert die Methode `parseDotTuple()` die Events mit Punktoperator der Prefix-Expression. Die Methode `parseVar()` verarbeitet schließlich den genauen Wert, der auf dem Kanal erzeugt wird. Der Folgezustand oder Folgeprozess der Prefix-Expression wird mithilfe der Methode `parseCallFunction()` gelesen. Ferner ist der parsierte Lambda-Ausdruck eine Liste von CSPFunction, siehe Listing 45. Dabei speichert das Attribut `trace` die Prefix-Expression, sowie `targetCall` den Folgeprozess. Diese Liste wird dann dem CSPPatter-Objekt zum Attribut `functions` hinzugefügt, wie in Listing 44 zu sehen. Dadurch, dass in dieser Arbeit nur der External Choice-Operator genutzt wird, wird dann das Attribut `choice` naiv auf `Choice.EXTERNAL_CHOICE` gesetzt.
Nachdem existierende Lambda-Ausdrücke eingelesen wurden, werden nun alle weiteren CSP_M-Funktionen eingelesen. Zuerst wird der Choice-Operator mit der Methode `parseChoice()` parsiert, der in diesem Fall immer der External-Choice Operator ist. Ähnlich wie beim Parsieren der Lambda-Audrücke werden zunächst alle Prefix-Expression mithilfe der Methode `createCSPFunctionFromPrefixExp()` eingelesen. Diese wiederum parsiert jeden Ausdruck, was von der Methode

`parsePrefixExpression()` übernommen wird. Hierbei werden wieder alle Events (`DotTuple`) und der Folgeprozess mithilfe der Methode `parseCallFunction()` parsiert. Somit ergibt sich eine Liste von CSPFunction, die dem CSPPattern-Objekt hinzugefügt werden.
Wurden alle inkludierten Modelle parsiert, kann es vorkommen, dass es weitere CSP$_M$-Prozesse gibt, die nicht zu Modellen, sondern schon zu der Gleisnetzspezifikation gehören. In diesem Fall werden diese ebenfalls parsiert.

Parsieren der synchronisierten Prozesse. Nachdem alle CSP$_M$-Prozesse eingelesen wurden, werden im nächsten Schritt alle Synchronisationsbedingungen verarbeitet. Listing 49 zeigt als Beispiel der Synchronisationsbedingung des generierten kleinen Gleisnetzes.

```
1 POINT_p5 = ( TRACK_ELEMENT(3, 3, 5) [| { rail.5 } |]
2            ( TRACK_ELEMENT(4, 4, 6) [| { rail.6 } |]
3            POINT(5, 9, 5, 6) ))
```

Listing 49: Beispiel für eine Synchronisationsbedingung

Der Prozess `POINT_p5` beschreibt, dass `TRACK_ELEMENT(3, 3, 5)` mit `rail.5` und `TRACK_ELEMENT(4, 4, 6)` mit `rail.6` mit der Weiche `POINT(5, 9, 5, 6)` synchronisiert sind. Der Prozess `POINT_p5` wird im weiteren Verlauf als Synchronisationsgruppe betrachtet.
Der Prozess `POINT_p5` wird schließlich durch die Java-Klasse `CSPBindingGroup` beschrieben, wie in Listing 50 zu sehen. Diese besitzt einen Namen `name`, der den Prozessnamen speichert. Das Attribut `alternativeName` ist für das Prozess-Renaming gedacht. Die Liste der CSPBindingElement, `elements`, speichert alle sich in der Gruppe befindlichen Elemente. So wird zunächst mithilfe der Methode `parsePatBind()` alle Synchronisationsgruppen aus der XML-Datei mit dem Tag `PatBind` gelesen.

```
1 public class CSPBindingGroup
2 {
3   private String                    name;
4   private String                    alternativeName;
5
6   public List<CSPBindingElement>  elements;
7 }
```

Listing 50: CSPBindingGroup Java-Klasse

Der Bereich mit dem Schlüsselwort `PatBind` deklariert einen Prozess, der Prozesse synchronisiert. Der Name des Prozesses wird mithilfe der Methode `getName()` parsiert. Durch das XML-Schlüsselwort `Parens` werden schließlich die einzelnen deklarierten Point-, Crossing- und Track-Element-Prozesse deklariert und deren Synchronisationsbedingungen zu einem anderen CSP$_M$-Prozess dieser Gruppe angegeben. Die Methode `parseParensBinding()` wird dieser XML-Block verarbeitet. Die Deklaration einer Point-, Crossing- und Track-Element Deklaration ist in dem XML-Dokument unter dem Schlüsselwort `ProcSharing` angegeben. Ein `ProcSharing`-Instanz beschreibt beispielsweise `TRACK_ELEMENT(3, 3, 5)`. Diese Deklaration wird

dann mit der Methode `parseCallFunction()` parsiert. Die jeweilige Synchronisationsbedingung zu einem anderen Gleiselement-Prozess wird mit der Methode `parseSynchronizationRails()` eingelesen. Dadurch, dass es in CSP$_M$ möglich ist, dass zwei Prozesse sich synchronisieren sowie rekursiv sich weitere auf die jeweils vorhandenen synchronisieren können, rufen sich `parseProcSharing()` und `parseParensBinding()` sich gegenseitig rekursiv auf.
Das Resultat ist ein Objekt vom Typ `CSPBindingElement`, zu sehen in Listing 51. Die Synchronisationsbedinungen werden in der Liste der Integer, `connector`, gespeichert. Die Deklaration des CSP$_M$-Prozesses ist das Attribut `functionCall` vom Typ `CSPFunctionCall`.

```
1 public class CSPBindingElement
2 {
3   public CSPFunctionCall  functionCall;
4   public List<Integer>    connector;
5   public List<String>     groupCall;
6 }
```

Listing 51: CSPBindingElement Java-Klasse

Die eingelesenen Elemente vom Typ CSPBindingElement werden zu dem Attribut `elements` von der CSPBindingGroup hinzugefügt.
Nach dem Prinzip wird jeder CSP$_M$-Prozess, der Prozesse synchronisiert, eingelesen und als CSPBindingGroup gespeichert.

Erstellen des Syntaxbaums. Im nächsten Schritt ist es notwendig, aus den eingelesenen Prozessen einen Syntaxbaum zu erstellen. Dieser wird später für den Trace-Generator und den Gleisnetz-Interpreter benötigt. Das Ziel ist es, aus dem Syntaxbaum ein Labelled Transitions System zu generieren. Labelled-Transition-Systems sind für CSP$_M$ nützlich, da CSP-Prozesse ideal mithilfe dieser Darstellung darstellen lassen. [MGR08]
Ein Labelled Transition System ist ein Tupel

$$LTS = (S, S_0, \Sigma, R)$$

, wobei S der Zustandsraum, S_0 die Menge der Startzustände, Σ die Menge der möglichen Events und $R \subseteq S \times \Sigma \times S$ die Menge der Transitionsrelationen ist. [JP19] Ein Zustand S wird hierbei durch ein CSPProcess repräsentiert, sowie Σ durch Werte vom Typ List<CSPEvent>.
Untergliedert ist dieses Vorgehen in mehreren Schritten. Ein Gleisnetz besteht grundsätzlich aus den drei Gleiselementen Point, Crossing und Track-Element. Daher werden zunächst alle eingelesenen CSP$_M$-Prozesse von Point, Crossing und Track-Element gesucht. Diese bildet die Grundlage dafür, dass im nächsten Schritt alle Instanzen dieser Grundelemente erzeugt werden können, die innerhalb einer CSPBindingGroup sind. Die Instanziierung der Elemente in den CSBBindingGroup-Gruppen wird mit der Methode `createProcessInstancesFromGroups()` durchgeführt. Der

genaue Aufrufgraph ist im Anhang in Abbildung 21 zu sehen. Eine CSPBinding-Group enthält Elemente vom Typ CSPBindingElement. Dieses speichert einmal die Synchronisationsbedingung und einmal den CSP_M-Prozess mit samt der Parameter (siehe Listing 51). Nun wird ein neues Objekt vom Typ CSPProcess erzeugt und die in dem `CSPBindingElement` enthaltene Parameter der `paramsMap` (Listing 43) zugeordnet.
Nachdem alle Instanzen erstellt wurden, ist der nächste Schritt, alle Folgeprozesse einer CSP_M-Funktion zu den korrekten CSPProcess-Instanzen zu linken. Wie in Listing 45 zu sehen, hat die Klasse CSPFunction zwei Attribute namens `prev` und `next`. Das Attribut `prev` ist der aktuelle, Ursprungs-CSPProcess. Das Attribut `next` hingegen zeigt auf den nächsten Prozess. Die Methode
`LinkSubProcessFSM(CSPProcess p)` iteriert für einer CSPProcess-Instanz durch alle CSPFunction-Objekte und sucht in allen CSPProcess-Instanzen nach der Instanz des Folgeprozesses und setzt diesen. Dieser Vorgang wird rekursiv ausgeführt, damit alle Folgeprozesse der Sub-Prozesse gesetzt werden. In einem konkreten Fall werden für die Instanz `POINT(1, 2, 3, 4)` für alle Funktionen die Folgeprozesse gesetzt. Da die Liste der Instanzen nur die CSPProcess-Instanzen mit dem Namen POINT, CROSSING und Track-Element enthalten und beispielsweise nicht MARKED oder LOCKED_STR, sowie alle weiteren Sub-Prozesse, iteriert diese Methode rekursiv durch alle Sub-Prozesse und setzt jeweils alle Folgeprozesse.
Aus den nun verlinkten Prozessen soll nun eine Menge von Transitionsrelationen eines LTS entstehen, wie oben definiert. Dazu sind zentral alle CSPFunction-Objekte zu betrachten. Wie in Listing 45 zu sehen, kann aus dem `trace`- sowie `prev`- und `next`-Attribut eine Transition eines LTS entwickelt werden. Die Java-Klasse `Transition` definiert eine Transition eines LTS, wie in Listing 52 zu sehen.
Mithilfe der Java-Methode `getProcTransitions(CSPProcess p)` werden nun für alle CSPProzess-Instanzen die Transitionen erzeugt und in den jeweiligen CSPFunction-Objekten gespeichert.

```
1 public class Transition
2 {
3   public CSPProcess       s0;
4   public List<CSPEvent>   sigma;
5   public CSPProcess       s1;
6 }
```

Listing 52: LTS-Transition Java-Klasse

Da nun jede Funktion einen CSPProcess als Folgeprozess besitzt, können die Transitionen erzeugt werden. Da hierbei nur der Zugtrace auf dem Gleisnetz und die Weichenstellungen nötig sind, werden in `sigma` alle CSPEvents einer CSPFunction gespeichert. Diese Methode ruft sich rekursiv auf, da somit jeder Sub-Prozess der Point-, Crossing- und Track-Element-Instanz verarbeitet wird und keine Transition innerhalb der Modell-Implementierung missachtet wird.
Somit wurden nun alle Transitionen aller CSPProcess-Instanzen erstellt. Dadurch

existieren nun verschiedene Labelled-Transition-Systems für jeden CSPProcess.
Um nun jedoch das konkrete Gleisnetz als LTS zu repräsentieren, müssen die synchronisierten CSP_M-Prozesse als Transitionen dargestellt werden. Die Java-Methode `makeTransitionsRailwayNetwork()` erstellt nun aus allen `CSPBindingGroup` -Instanzen Synchronisationstransitionen.
Diese Java-Methode beachtet dabei drei Fälle, wie CSP_M-Prozesse synchronisiert sein können. Die folgenden drei Fälle beschreiben den Ablauf der Methode. Es ist zudem möglich, dass in jedem Fall die Prozesse über mehrere `rail`-Events synchronisiert. In dem Fall wird jeder Fall für jedes `rail`-Event ausgeführt.

1. Die erste Methode ist direkt zwei CSP_M-Prozessinstanzen zu synchronisieren. Mit Prozessinstanzen ist hierbei gemeint, dass ein Gleiselement-CSP_M-Prozess mit einem CSP_M-Prozess Point/Crossing/Track-Element verbunden ist. Wie in Listing 53 zu sehen, ist `GRP_00` die Synchronisation von `POINT(12, 56, 70, 71)` mit `TRACK_ELEMENT(202, 70, 69)` auf das Event rail.70.

```
1 GRP_00 = (POINT(12, 56, 70, 71) [| { rail.70 } |]
2         TRACK_ELEMENT(202, 70, 69))
```

Listing 53: Zwei CSP_M-Prozesse sind synchronisiert

   Da es sich hierbei um zwei CSP_M-Prozesse vom Typ CSPProcess handelt, wird eine Transition, wie in Listing 52 zu sehen, mit `POINT(12, 56, 70, 71)` als `s0`, `TRACK_ELEMENT(202, 70, 69)` als `s1` und der Wert 70 von `rail.70` zu der `sigma`-Liste hinzugefügt.

2. Als zweiten Fall ist es in CSP_M möglich, dass zu einem bestehenden CSP_M-Prozess, der Prozesse synchronisiert, weitere synchronisiert werden sollen. Soll beispielsweise zum CSP_M-Prozess `GRP_00` aus Listing 53 ein weiteres Gleiselement synchronisiert werden, so ist das auch möglich. Listing 54 zeigt diesen Fall.

```
1 GRP_36 = (GRP_00 [| { rail.56 } |] POINT(11, 56, 53, 52))
```

Listing 54: Synchronisation CSP_M-Prozess zu einer Gruppe

   Dieser Ausdruck in CSP_M-Syntax bedeutet, dass der CSP_M-Prozess `GRP_00` sich über rail.56 mit dem CSP_M-Prozess `POINT(11, 56, 53, 52)` synchronisiert. Da nicht jeder CSP_M-Prozess aus `GRP_00` auf alle erzeugbaren Events definiert ist (siehe Abschnitt 2.4) synchronisiert sich nur ein CSP_M-Prozess der Gruppe `GRP_00` auf dieses Event. Im Folgeschritt muss nun nach genau dem Prozess gesucht werden, der sich auf das Event `rail.56` synchronisiert. Da jede CSP_M-Prozessinstanz die Menge der Events auf die dieser synchronisiert als Parameter besitzt, muss hierbei lediglich durch die Liste der Parameter des CSPProcess iteriert werden. Wurde der passende Prozess gefunden, so kann wie in (1) eine Transition erzeugt werden.

3. Der dritte Fall ist die Synchronisation zwischen Gruppen. Listing 55 zeigt Gruppe `GRP_36` aus (2) sowie eine andere Gruppe `GRP_30`. Synchronisiert werden diese beiden CSP$_M$-Prozesse über `rail.71`.

```
1     GRP_54 = (GRP_30 [| { rail.71 } |] GRP_36)
```

Listing 55: Synchronisation zwischen Gleisnetz-Gruppen

   Ähnlich wie in (2) wird nun nach CSP$_M$-Prozessen gesucht, die sich auf das Event `rail.71` synchronisieren. Jedoch wird hierbei in beiden Gruppen nach diesen Elementen gesucht. `POINT(12, 56, 70, 71)` aus Listing 53 hört auf das Event `rail.71`. Aus der Gruppe `GRP_30` ist das der Prozess `TRACK_ELEMENT(203, 72, 71)`. Diese Prozesse bilden dann eine neue Transition mit dem `sigma`-Event `rail.71`.

Sowohl für den CSP$_M$-Trace-Generator, als auch für den Gleisnetz-Interpreter werden diese Schritte durchgeführt, da diese auf Grundlage von Labelled-Transition-Systems arbeiten. Dabei ist die Java-Klasse `Transition` aus Listing 52 die zentrale Datenstruktur. Durch Traversierung des LTS und, dass das Gleisnetz in beiden Richtungen befahrbar ist, werden danach für jede Transition eine Transition in entgegengesetzter Richtung erzeugt. Dabei werden lediglich `s0` und `s1` der Transition vertauscht.

4.2.4 CSP$_M$-Trace-Generator

Der CSP$_M$-Trace-Generator ist einer der beiden Kernbestandteile dieser Verifikationssoftware. Die Ausgabe dieses Programms ist die Grundlage für den Gleisnetz-Interpreter. Dabei extrahiert der Trace-Generator alle Traces aus einer CSP$_M$-Spezifikation und speichert diese in die Trace-Datei, wie beispielsweise in Listing 37 zu sehen.
Die zugrunde liegende Datenstruktur des CSP$_M$-Trace-Generators ist in Listing 56 zu sehen. Diese bildet den Kontext für den Hauptalgorithmus des Trace-Generators. Dieser traversiert das Labelled-Transition-System und nutzt dabei die Java-Klasse `TreeLayer` als Grundlage.

```
1 public class TreeLayer
2 {
3   public List<CSPEvent>   prefixTrace;
4   public Transition       transition;
5 }
```

Listing 56: TreeLayer Java-Klasse

Das Attribut `transition` der Klasse TreeLayer speichert die nächste zu evaluierende Transition innerhalb des Graphens des LTS. Nachdem diese evaluiert wurde, werden die Events dieser Transition zu der Liste `prefixTrace` hinzugefügt, das alle schon evaluierten Events der zuvor ausgeführten Transitionen enthält.

```
1 private List<List<CSPEvent>> trace(List<TreeLayer> a, List<TreeLayer> b);
```

Listing 57: Prototyp des implementierten Trace-Finding Algorithmus in Java

Der rekursive Algorithmus zur Suche nach den Traces im Gleisnetz implementiert die Breitensuche durch das Labelled Transition System. Zunächst wird ein Startpunkt im Gleisnetz bestimmt in Form einer Weiche, ab der der Trace-Generator alle nachfolgenden Zug-Traces sucht. Danach werden zwei Objekte vom Typ TreeLayer erstellt, die als Parameter für den Trace-Finding-Algorithmus benötigt werden. Diese Java-Methode, der Methode-Prototyp ist in Listing 57 zu sehen, bekommt zwei Parameter als Liste vom Typ TreeLayer `a` und `b`. Die Liste `a` speichert alle TreeLayer-Objekte der aktuellen Schicht im Graphen. Die Liste `b` hingegen speichert alle nachfolgenden TreeLayer-Objekte aus der jeweiligen Schicht nach der aktuellen Schicht. Somit werden immer die Elemente der aktuellen und nächsten Schicht gespeichert. Die Methode gibt dann schließlich eine Liste von Listen von CSPEvent zurück, somit eine Menge von Traces im LTS.
Abbildung 15 zeigt als Beispiel ein Labelled-Transition-System mit

$S = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7\}$
$S_0 = \{s_0\}$
$\Sigma = \{a, b, c, d, e, f, g, h, i\}$
$R = \{(s_0, \{a\}, s_1), (s_0, \{b\}, s_2), (s_1, \{c\}, s_3), (s_2, \{e\}, s_4), (s_2, \{d\}, s_5),$
$(s_3, \{h\}, s_7), (s_4, \{g\}, s_6), (s_5, \{f\}, s_6), (s_6, \{i\}, s_7)\}.$

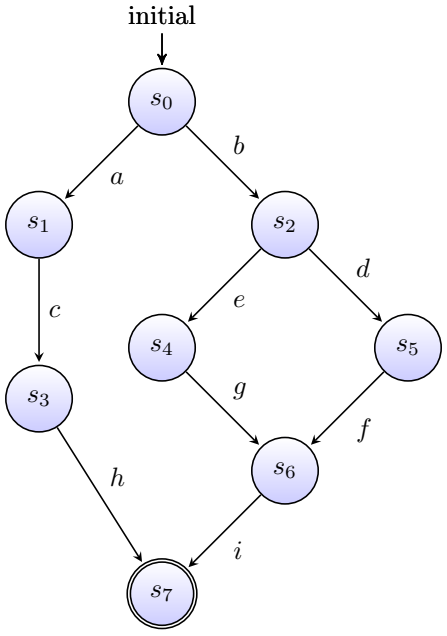

Abbildung 15: Beispiel Labelled-Transition-System

Zunächst beinhaltet die Liste der TreeLayer `a` alle ausgehenden Transitionen von dem angegebenen Startzustand. In diesem Fall ist der Startzustand der Zustand

s_0, folglich werden zwei neue TreeLayer-Objekte erstellt mit jeweils der Transition $(s_0, \{a\}, s_1)$ und einmal mit der Transition $(s_0, \{b\}, s_2)$ als Attribut `transition`. Wenn s_0 der STOP- oder SKIP-Zustand ist, wird direkt terminiert. Der Algorithmus geht dabei nun wie folgt vor. Der Algorithmus iteriert durch alle TreeLayer-Objekte der Liste `a` und evaluiert für jedes `transition`-Attribut des jeweiligen TreeLayers die Eventmenge Sigma. Da nur der Zug-Trace sowie die dafür benötigten Weichenstellungen benötigt werden, sucht der Algorithmus alle `rail`-Events der Sigma-Eventmenge und fügt diese Events der Liste `prefixTrace` hinzu. In einem konkreten Fall besteht der aktuelle `prefixTrace` des TreeLayers mit der Transition $(s_0, \{a\}, s_1)$ genau aus dem Event $\{a\}$. Der Prefix-Trace des aktuellen TreeLayers wird schließlich zu der Liste aller Prefix-Traces hinzugefügt.
Danach sucht der Algorithmus nach Folgetransitionen, indem die Methode `findSuccessor(Transition)` nach Folgetransitionen sucht. Der Algithmus 2 zeigt diese Methode. Dem Algorithmus wird eine Transition übergeben. Danach wird durch alle vorhandenen Transitionen iteriert und geprüft, ob diese ungleich, jedoch einen gemeinsamen Zustand in Form eines CSPProcess (Gleiselement) haben. Dieser gemeinsame Zustand wird beschrieben durch `sState` in Algorithmus 2. Gibt es einen gemeinsamen Zustand, also ein gemeinsames Gleiselement, so kommt die gefundene Transition als Folgetransition infrage. Jedoch muss dazu zuerst evaluiert werden, ob es einen Pfad von der Transition `t` zur Transition `trans` gibt, sodass es einen Übergang von Transition `t` zur Transition `trans` gibt. Wenn es diesen gibt, wird untersucht, ob es eine benötigte Weichenstellung für den Übergang von der Transition `t` zur Transition `trans` gibt. Diese Weichenstellung wird berechnen, indem der Zustandsautomat des gemeinsamen Zustandes (gemeinsamen Gleiselementes) evaluiert wird. Die Variable `pointPosition` speichert diesen Wert in Form eines CSP_M-Events. Dieses repräsentiert das CSP_M-Event, welches die Weiche in die benötigte Stellung versetzt.
Die Folgetransition wird schließlich mit der Weichenstellung zu der Liste `successors` hinzugefügt. Dabei ist der Typ dieser Liste `TransitionSuccessor` wie in Listing 58 zu sehen. Damit wird der Folgetransition eine konkrete Weichenstellung zugeordnet.

```
1 public class TransitionSuccessor
2 {
3   public Transition   transition;
4   public CSPEvent     pointPosition;
5 }
```

Listing 58: TransitionSuccessor- Java-Klasse

Für das Beispiel aus Abbildung 15 sind die Folgetransitionen von der Transition $(s_0, \{b\}, s_2)$ somit $\{(s_2, \{e\}, s_4), (s_2, \{d\}, s_5)\}$. Da genau diese Transitionen infrage kommen, wird für jede Folgetransition ein Objekt vom Typ TreeLayer erzeugt. Das `transition`- Attribut der neu-angelegten TreeLayer-Objekte ist die errechnete Transition des jeweiligen TransitionSuccessor. Damit der errechnete Trace in jeder Schicht erhalten bleibt, wird der Prefix-Trace der vorherigen Transition zu jedem

Algorithm 2: Algorithmus zum Finden der Folgetransitionen

Input : Transition t
Output: Eine Liste von möglichen Folgetransitionen

```
1 findSuccessor (Transition t):
 1: successors;
 2: for trans : allTransitions do
 3:    if trans == t then
 4:       continue;
 5:    end if
 6:    sState = getShared(t, trans);
 7:    if not sState then
 8:       continue;
 9:    end if
10:
11:    t_re = railEvent(t.sigma);
12:    trans_re = railEvent(trans.sigma);
13:
14:    if pathExists(sState, t_re, trans_re) then
15:       pointPos = pointPosition(sState, t_re, trans_re));
16:       successors.add(ts(trans, pointPos));
17:    end if
18: end for
19: return successors;
```

Treelayer-Objekt der Folgetransitionen hinzugefügt. Falls zu der Folgetransition eine benötigte Weichenstellung errechnet wurde, wird dieser Stellbefehl in Form eines CSPEvent-Objektes auch dem jeweiligen Prefix-Trace hinzugefügt. Anschließend werden alle TreeLayer-Objekte der neuen Schicht im Labelled-Transition-System der Liste `b` hinzugefügt. Wurden alle TreeLayer-Objekte der Liste `a` evaluiert, so wird rekursiv diese Methode aufgerufen, indem die Liste `b` nun `a` ist, wodurch nun nach dem Prinzip der Breitensuche die nächste Schicht ausgewertet wird.
Der Algorithmus terminiert genau dann, wenn sowohl Liste `a` und Liste `b` leer sind. Das bedeutet, dass es sowohl keine Elemente der aktuellen Schicht, wie auch aus der nächsten Schicht vorhanden sind. Dann gibt der Algorithmus die Liste aller Prefix-Trace zurück. Der Algorithmus wertet jedoch Folgetransitionen genau dann nicht aus, wenn der Zustand `s0` der Transition STOP, SKIP oder der Startknoten nach Auswertung dessen ist. Ebenso wertet dieser die Folgetransitionen auch dann nicht aus, wenn das `rail`-Event der Transition des aktuellen TreeLayers schon in der Liste der Prefix-Trace enthalten ist, damit Lasso-Traces erkannt und Deadlocks während der Trace-Generierung vermieden werden. [JP19] Ebenso gibt es die Möglichkeit, nach einer bestimmten Suchtiefe im Baum zu terminieren. Der Algorithmus terminiert dann, wenn der Trace eine maximale Länge x hat (Flag `-d` x).
Nach Terminierung des Algorithmus wird eine Trace-Datei ausgegeben, die vom CSP$_M$-Trace-Generator erzeugt wird. Listing 59 zeigt alle Traces, Listing 59 zeigt die Traces nach dem Entfernen aller Traces, die Untermengen von anderen Traces sind.

```
b;
a;
b;d;
b;e;
a;c;
b;d;f;
b;e;g;
a;c;h;
b;d;f;i;
b;e;g;i;
```

Listing 59: Traces aus dem LTS aus Abbildung 15

```
a;c;h;
b;d;f;i;
b;e;g;i;
```

Listing 60: Traces nach Entfernen von Untermengen

Danach werden alle gefundenen Traces nach Entfernen aller Untermengen gemäß der Konvention aus Listing 37 in die `tracefile.txt` geschrieben, dessen Inhalt der aus Listing 60 entspricht.

4.2.5 Gleisnetz-Interpreter

Der Gleisnetz-Interpreter gilt als Gegenstück zu dem Trace-Generator aus Abschnitt 4.2.4. Der Gleisnetz-Interpreter liest die Trace-Datei ein und prüft für jeden Trace, ob dieser in dem vom Gleisnetz-Interpreter eingelesenen Gleisnetz enthalten ist. Dabei gibt dieser entweder ein Passed oder eine Failed als Antwort auf jeden Trace der Trace-Datei.
Grundsätzlich bekommt der Gleisnetz-Interpreter als Eingabe eine zu prüfende Gleisnetz Spezifikation in Form einer CSP-Datei und eine Trace-Datei mit den zu testenden Traces. Die Trace-Datei muss dabei vorliegen, wie in Listing 37 gezeigt. Ebenso gilt für die CSP-Datei die Konvention aus Listing 36. Zunächst wird die CSP-Datei, wie in Abschnitt 4.2.3 erläutert, eingelesen und verarbeitet. Danach wird die Trace-Datei eingelesen. Der Gleisnetz-Interpreter besitzt für jeden Trace einen eigenen `CheckerContext`, der den aktuellen Kontext eines Traces der Trace-Datei speichert, wie in Listing 61 zu sehen. Die LinkedList `railTrace` speichert den Trace nach `rail`-Events. Die HashMap `ppMap` speichert alle Weichenstellungen der Trace-Datei und bildet Weichen IDs auf die benötigte Weichenstellung, `s` für straight und `c` für cross, ab.

```
class CheckerContext
{
    public HashMap<String, String>  ppMap;          /**!< PointPositions */
    public LinkedList<String>       railTrace;      /**!< rail Trace */
    public String                   line = "";
}
```

Listing 61: Checker-Context Java Klasse

Die Trace-Datei wird zeilenweise eingelesen und für jede Zeile wird ein Objekt vom Typ CheckerContext erzeugt. Danach wird für jeden CheckerContext die Methode `checkOneTrace` ausgeführt.
Diese Methode prüft, ob in dem eingelesenen Gleisnetz ein Trace enthalten ist, wie im Aktivitätsdiagram aus Abbildung 16 zu sehen. Dazu liegt das zu prüfende Gleisnetz wieder als LTS vor. Die Methode `checkOneTrace` iteriert durch die Liste aller `rail`-Events des `rail`-Traces aus dem Kontext. Zunächst werden alle Transitionen aus der Transitionsmenge des eingelesenen Gleisnetzes gesucht, die als `rail`-Event das erste `rail`-Event des Traces enthalten. In einem Gleisnetz sollten es immer genau zwei Transitionen sein, die jeweils identisch, jedoch in anderer Fahrtrichtung (s0 und s1 vertauscht) sind. Danach werden die Transitionen gesucht, die als `rail`-Event das nächste `rail`-Event in dem zu prüfenden Trace haben. Nachfolgend wird schließlich geprüft, ob es eine Transition aus der Menge der Transitionen des zweiten `rail`-Events gibt, die als Nachfolger einer Transition des ersten `rail`-Events gibt.
Als Beispiel ist der Trace `{ 4;7;9;3 }` gegeben. Der Algorithmus sucht nun alle Transitionen, die das Event `rail.4` als Event beinhalten. Ebenso werden die Transitionen gesucht, die das Event `rail.7` als Event beinhalten. Danach wird geprüft, ob Transitionen aus der Menge mit dem Event `rail.7` unmittelbare Folgetransitionen aus der Menge der Transitionen mit dem Event `rail.4` sind. Falls keine Transitionen mit `rail.7` als Nachfolger von Transitionen mit `rail.4` infrage kommen, so existiert der Trace in dem zu prüfenden Gleisnetz nicht und es wird `false` zurückgegeben. Andernfalls wird genau eine Transition gefunden, die als Folgetransition infrage kommt.

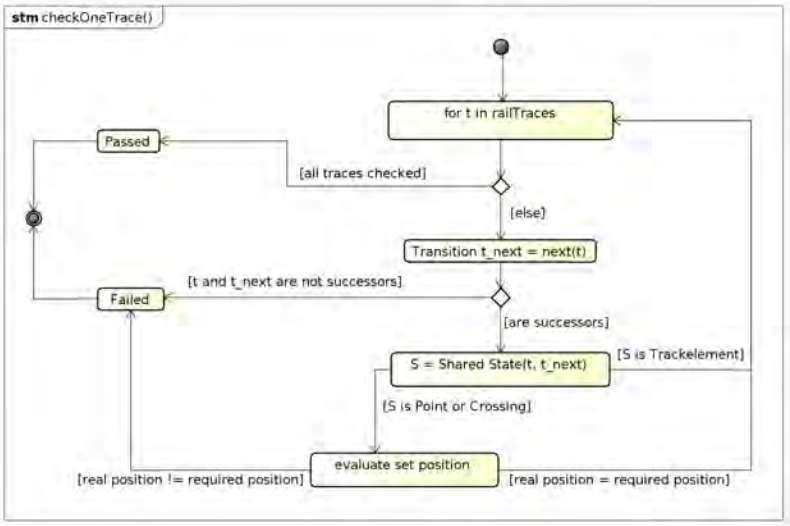

Abbildung 16: Zustandsautomat der Methode `checkOneTrace()`

Wurde eine möglich Folgetransition gefunden, so muss geprüft werden, ob der gemeinsame Zustand vom Typ CSPProcess vom Typ Point oder Crossing ist. In diesem Fall muss die benötigte Weichenstellung ermittelt werden, damit die Folgetransition als valide Folgetransition für die Starttransition gilt. Wurde die benötigte Weichen-

stellung ermittelt, so wird in der HashMap `ppMap` geprüft, ob für diesen Trace für diese Weiche diese ermittelte Weichenstellung benötigt wird. Falls nicht, dann wird `false` zurückgegeben, da der Trace so nicht in dem zu testenden Gleisnetz enthalten ist. Wenn die Weichenstellung jedoch übereinstimmt, so wird mit der nun validen Folgetransition als Starttransition und mit dem nächsten `rail`-Event des Traces aus dem aktuelle CheckerContext fortgefahren.
Der Algorithmus terminiert dann, wenn jedes Event des zu prüfenden Traces überprüft wurde oder ein bestimmtes Event oder der gesamte Trace nicht in dem zu testenden Gleisnetz enthalten ist. Ist der Trace in dem zu testenden Gleisnetz enthalten, so wird `Passed` ausgegeben, sonst `Failed`.

4.2.6 Verifikation mit dem Trace-Generator und dem Gleisnetz-Interpreter

In diesem Abschnitt wird nun mit Hilfe des Gleisnetz-Interpreters und des Gleisnetz-Trace-Generators das vom CSP_M-Code-Generator generierte Gleisnetz auf Äquivalenz mit der Referenzimplementierung geprüft. Dabei soll jeweils gezeigt werden, dass alle in der Referenzimplementierung befindlichen Traces in dem generierten enthalten sind sowie, dass alle Traces aus dem generierten Gleisnetz in der Referenzimplementierung enthalten sind. Dazu wird zunächst geprüft, ob der Gleisnetz-Trace-Generator und der Gleisnetz-Interpreter korrekt auf Trace-Äquivalenz prüfen. Hinterher wird aus dem TEAMOD-Gleisnetz der Referenzimplementierung Traces generiert und überprüft, ob diese in dem generierten Gleisnetz enthalten sind.

Test Korrektheit des Trace-Generators und Interpreters Zunächst wird geprüft, ob der Gleisnetz-Trace-Generator und der Gleisnetz-Interpreter korrekt Traces generieren und evaluieren. Dazu wird das kleine Gleisnetz, welches ein Teil des TEAMOD-Gleisnetzes ist, herangezogen.

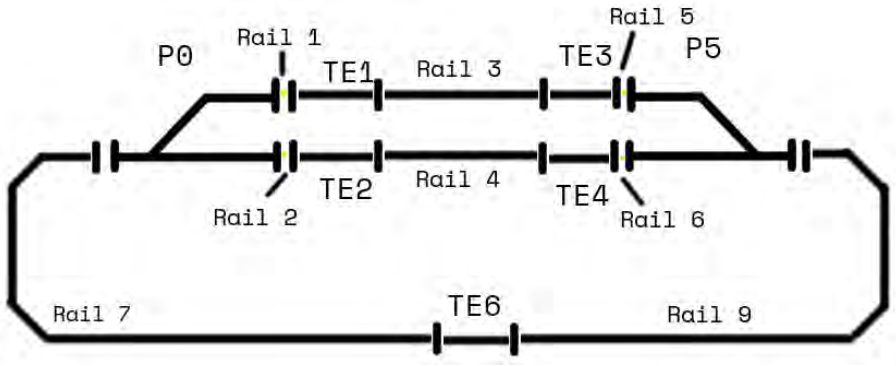

Abbildung 17: Kleines Gleisnetz zum Test des Trace-Generators und des Interpreters

Nach dem Ausführen des Trace-Generators mit der Weiche mit der ID 0 als Startpunkt für das Trace-Generieren wird dabei eine `tracefile.txt` erzeugt, deren Inhalt in Listing 62 gezeigt ist.

```
5:c;0:c;1;3;5;9;7;
5:s;0:s;2;4;6;9;7;
5:c;0:c;7;9;5;3;1;
5:s;0:c;7;9;6;4;2;
```

Listing 62: Korrekte Trace-Datei des kleinen Gleisnetzes

Hierbei ist klar zu sehen, dass die Weichen für die jeweiligen Traces korrekt gestellt sind, sowie der Trace dem `rail`-Trace aus dem Gleisnetz aus Abbildung 17 entspricht. Wird nun der Gleisnetz-Interpreter auf dasselbe Gleisnetz-Modell mit der `tracefile.txt` ausgeführt, so wird für jeden Trace ein Passed-Ergebnis erwartet. Das Ergebnis ist in Listing 63 zu sehen und erfüllt die Erwartungen.

```
5:c;0:c;1;3;5;9;7; -> Passed.
5:s;0:s;2;4;6;9;7; -> Passed.
5:c;0:c;7;9;5;3;1; -> Passed.
5:s;0:c;7;9;6;4;2; -> Passed.
============================================
Gleisnetz-Interpreter Result:
0/4 Failed.
============================================
```

Listing 63: Ergebnis positiv-Test

Nun wird die `tracefile` geändert, indem die Weichenstellungen falsch gestellt werden. Damit wird nun getestet, ob der Gleisnetz-Interpreter auch falsche Weichenstellungen erkennt. Dazu wird die Weiche mit der ID 5 aus der ersten Zeile auf straight gesetzt und Weiche mit der ID 0 aus der vierten Zeile auf cross gesetzt, sodass in beiden Fällen ein Fehler entsteht. Die Trace-Datei, den Inhalt entsprechend Listing 64 hat.

```
5:s;0:c;1;3;5;9;7;
5:s;0:s;2;4;6;9;7;
5:c;0:c;7;9;5;3;1;
5:s;0:c;7;9;6;4;2;
```

Listing 64: Manipulierte Trace-Datei des kleinen Gleisnetzes

Das Ergebnis ist wie erwartet. Beide Fehler sind erkannt worden, indem für die jeweiligen Traces ein Fehler erkannt wurde. Das ist an der Ausgabe des Gleisnetz-Interpreters in Listing 65 zu sehen.

```
5:s;0:c;1;3;5;9;7; -> Failed.
5:s;0:s;2;4;6;9;7; -> Passed.
5:c;0:c;7;9;5;3;1; -> Passed.
5:s;0:c;7;9;6;4;2; -> Failed.
============================================
Gleisnetz-Interpreter Result:
2/4 Failed.
============================================
```

Listing 65: Ergebnis negativ-Test

Somit ist zunächst gezeigt, dass der Gleisnetz-Interpreter wie auch der Trace-Generator richtige Traces generieren und, dass der Gleisnetz-Interpreter das Gleisnetz richtig interpretiert und prüft.

Verfikation des TEAMOD Gleisnetzes Nun soll das TEAMOD-Gleisnetz verifiziert werden, indem geprüft wird, ob die Referenzimplementierung aus [Brü20] äquivalent zu dem generierten Gleisnetz der Ausgabe des CSP$_M$-Code-Generators ist. Dazu werden zunächst aus der Referenzimplementierung des Gleisnetzes in CSP$_M$ Traces mithilfe des CSP$_M$-Trace-Generators generiert. Danach wird mithilfe des Gleisnetz-Interpreters geprüft, ob jeder Trace der Trace-Datei auch von dem generierten Gleisnetz implementiert wird. Danach werden Traces aus dem generierten Gleisnetz generiert und geprüft, ob diese von der Referenzimplementierung implementiert werden. Jedoch muss zunächst die Ausgabe des Code-Generators angepasst werden, damit die CSP$_M$-Kanäle des generierten Gleisnetzes in demselben Wertebereich sind wie die von der Referenzimplementierung. Das Manipulieren der Ausgabe ist im Abschnitt 4.1.1 genauer beschrieben. Danach muss die Konfiguration zum Generieren der Traces erstellt werden. Wie in 4.2.4 beschrieben, sucht der CSP$_M$-Trace-Generator bis zu einer bestimmten Länge der Traces in dem Labelled-Transition-Systems des Gleisnetzes. Es wird dann abgebrochen, wenn der Trace eine bestimmte Länge überschreitet. Somit ist es möglich, die Länge aller Traces generell zu beeinflussen und sorgt dafür, dass der Algorithmus nicht weiter nach Events sucht, die gegebenenfalls schon evaluiert wurden. Die Länge eines Traces setzt sich aus der Anzahl der für den Trace nötigen Weichenstellungen und den `rail`-Trace zusammen. Das TEAMOD-Gleisnetz besteht aus 77 möglichen `rail`-Events und insgesamt 28 Weichen und Kreuzweichen, somit kann der längste Trace höchstens 28+77 = 105 Elemente lang sein. Aus dem Grund, dass Weichen nicht mehrmals in einem Trace befahren werden dürfen sowie `rail`-Events ebenfalls nur einmal in einem Trace vorkommen dürfen, hat der längste mögliche Trace 105 Stellen. Somit kann der längste Trace in dem TEAMOD-Gleisetz höchstens 105 Elemente haben.

```
-> Generating traces ...
-> Generate traces done, preparing Tracefile.
-> Remove duplicates ...
############################
CSP2Trace Generator.

Total Traces found: 1145
############################
```

Listing 66: Ergebnis Trace-Generieren aus der Referenzimplementierung und dem Startpunkt Weiche 13

Somit werden Traces mit dem CSP$_M$-Trace-Generator mit der Länge 105 berechnet. Zuerst werden nun die Traces aus der Referenzimplementierung generiert. Das Ergebnis des CSP$_M$-Trace-Generators ist in Listing 66 und 67 zu sehen. Als Startpunkt wird eine beliebige Weiche ausgewählt, ab der Traces generiert werden. Begonnen

wird mit der Weiche mit der ID 13. Danach wird mit derselben Konfiguration Traces generiert, jedoch ist nun der Startpunkt die Weiche mit der ID 26. Damit wird gezeigt, dass der Trace-Generierungsprozess unabhängig von der jeweiligen Startweiche ist. Das Ergebnis des Trace-Generierungs-Prozesses mit dem Startpunkt als Weiche 13 ist in Listing 66 und mit Weiche 26 als Startpunkt in Listing 67 zu sehen.

```
-> Generating traces ...
-> Generate traces done, preparing Tracefile.
-> Remove duplicates ...
############################
CSP2Trace Generator.

Total Traces found: 1316
############################
```

Listing 67: Ergebnis Trace-Generieren aus der Referenzimplementierung und dem Startpunkt Weiche 26

Die Trace-Datei enthält somit insgesamt 1145 und einmal 1316 Traces, die aus der Referenzgleisnetz-Implementierung generiert wurden. Nun wird der Gleisnetz-Interpreter genutzt, um das generierte Gleisnetz zu verifizieren. Dieser erhält jeweils als zu testendes Gleisnetz das generierte Gleisnetz und als Tracefile die zuvor aus der Referenzimplementierung generierte Trace-Datei. Das Testergebnis mit dem Startpunkt Weiche mit der ID 13 ist in Listing 68 und mit dem Startpunkt Weiche mit der ID 26 ist in Listing 69 zu sehen.

```
============================================
Gleisnetz-Interpreter Result:
0/1145 Failed.
============================================
```

Listing 68: Ergebnis des Gleisnetz-Interpreters mit dem Startpunkt Weiche 13

```
============================================
Gleisnetz-Interpreter Result:
0/1316 Failed.
============================================
```

Listing 69: Ergebnis des Gleisnetz-Interpreters mit dem Startpunkt Weiche 26

Durch das Resultat ist zu sehen, dass alle Traces aus der Referenzimplementierung des TEAMOD-Gleisnetzes in dem generierten Gleisnetz des CSP$_M$-Code-Generators enthalten sind. In dieser Hinsicht ist der Äquivalenztest in dieser Richtung schon erfolgreich abgeschlossen. Zur Vollständigkeit muss nun verifiziert werden, ob dieses Ergebnis auch in anderer Richtung erzielt wird, sodass alle Traces aus dem generierten Gleisnetz auch in der Referenzimplementierung enthalten ist, damit ein vollständiger Äquivalenztest entsteht.
Somit werden nun Traces aus dem generierten Gleisnetz-CSP$_M$-Code generiert. Das Ergebnis des Trace-Generierungs-Prozesses mit der Weiche mit der ID 13 als Start-

punkt in Listing 70 zu sehen. Das Ergebnis mit der Weiche mit der ID 26 als Startpunkt ist in Listing 71 zu sehen.

```
-> Generating traces ...
-> Generate traces done, preparing Tracefile.
-> Remove duplicates ...
############################
CSP2Trace Generator.

Total Traces found: 1145
############################
```

Listing 70: Ergebnis Trace-Generieren aus dem generierten Gleisnetz und dem Startpunkt Weiche 13

```
-> Generating traces ...
-> Generate traces done, preparing Tracefile.
-> Remove duplicates ...
############################
CSP2Trace Generator.

Total Traces found: 1316
############################
```

Listing 71: Ergebnis Trace-Generieren aus dem generierten Gleisnetz und dem Startpunkt Weiche 26

Nun wird geprüft, ob die generierten Traces in der Referenzimplementierung mithilfe des Gleisnetz-Interpreters enthalten sind. Als zu prüfendes Gleisnetz wird folglich die Referenzimplementierung ausgewählt, als Trace-Datei die zuvor generierte Trace-Dateien. Das Ergebnis mit Weiche mit der 13 als Startpunkt ist in Listing 72 zu sehen. Das Ergebnis mit Weiche mit der ID 26 ist in Listing 73 zu sehen.

```
============================================
Gleisnetz-Interpreter Result:
0/1145 Failed.
============================================
```

Listing 72: Ergebnis des Gleisnetz-Interpreters mit dem Startpunkt Weiche 13

```
============================================
Gleisnetz-Interpreter Result:
0/1316 Failed.
============================================
```

Listing 73: Ergebnis des Gleisnetz-Interpreters mit dem Startpunkt Weiche 26

Hierbei ist zu sehen, dass auch bei diesem Äquivalenztest alle Traces des generierten Gleisnetzes in der Referenzimplementierung enthalten sind und kein Test mit Failed endete. Somit ist auch in diesem Fall gezeigt, dass die Referenzimplementierung des TEAMOD-Gleisnetzes und das generierte Gleisnetz äquivalent sind.

4.3 Erneute Testausführung der bestehenden Testfälle

In diesem Kapitel sollen nun Tests der Stellwerkstabelle aus [Brü20] erneut auf das generierte TEAMOD-Gleisnetz ausgeführt werden. Das erneute Ausführen dieser Tests stärkt das Vertrauen in die Äquivalenz des generierten Gleisnetzes zum Referenzgleisnetz, garantieren diese jedoch nicht, da diese Tests nicht erschöpfend sind. In diesem Abschnitt wird nur die Stellwerkstabelle mit dem generierten Gleisnetz verifiziert. Der zentrale Sicherheitsaspekt ist dabei, dass ein Zug, der eine valide, vom Stellwerk freigegeben Route befährt, nicht mit einem Zug kollidiert, der darauf wartet, eine Konfliktroute zu befahren. Simultanes Befahren einer Korrekter-, sowie Konfliktroute ist in keinem Fall zulässig. Dafür trifft das Stellwerk, auf Grundlage der Stellwerkstabelle, die Sicherheitsentscheidungen, welcher Zug fahren darf.
Die Stellwerkstabelle beschreibt dabei die möglichen Routen, die in dem Gleisnetz von Zügen befahren werden dürfen. Diese wird dabei vom zentralen autonomen Stellwerk genutzt, um den Zug sicher über das Schienennetz zu leiten. Ein Ausschnitt aus der Stellwerkstabelle ist in Abbildung 18 gezeigt.

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

Abbildung 18: Ausschnitt aus der Stellwerkstabelle

Die Stellwerkstabelle besteht aus Routen, wobei jede Zeile in dieser Tabelle eine Route repräsentiert. In der ersten Spalte ist die eindeutige Identifikationsnummer der Route gegeben. Danach folgt in Spalte `SRC` die das Start-, sowie Spalte `DST` das Ziel-Zugdetektionsgleis beschreibt. In der Spalte `PATH` sind die Nummern von Zugdetektionsgleisen gegeben, über die diese Route führt. In der Spalte `POINTS` sind die für die Route notwendigen Weichenstellungen beschrieben. Die jeweiligen Informationen sind per Komma getrennt. Eine Weichenstellung wird beschrieben durch die Weichen ID sowie die Stellung (Plus/Minus) nach dem Doppelpunkt. Die relative Länge der Route ist in Spalte `LENGTH`, sowie die erlaubte relative Höchstgeschwindigkeit ist in Spalte `MAX SPEED` gezeigt. Die nächste Spalte `CONFLICTS` zeigt die Routen, mit denen diese in Konflikt steht. Zwei Routen stehen genau dann in Konflikt, wenn diese sich über Weichen oder gemeinsam genutzte Zugdetektionsgleise kreuzen oder überschneiden. Die letzte Spalte `DIRECTION` beschreibt dabei die Fahrtrichtung der Route. Die Zahl null zeigt hierbei, dass die Route im Uhrzeigersinn verläuft, beziehungsweise eins gegen den Uhrzeigersinn.
Insgesamt besteht die Stellwerkstabelle aus 96 Routen, die dem Stellwerk zur Verfügung stehen. Jede Route wird im Folgenden zweimal getestet. Zunächst im Abschnitt 4.3.1 jede Route auf dem Gleisnetz verifiziert, ohne dass Konflikte auftreten. Dabei befindet sich nur ein Zug pro Test auf dem Gleisnetz. Weiter im Abschnitt 4.3.2 werden die Konfliktrouten im Bezug zur validen Route getestet. Ein Zug befährt dabei wieder die valide Route sowie jeweils drei Züge Konfliktrouten dieser Route. Das

wird schließlich für alle Konfliktrouten ausgeführt. Im Abschnitt 4.3.3 wird wieder jede valide Route befahren, jedoch simultan mit drei weiteren Zügen, die jeweils eine nicht in Konflikt stehende Route befahren. Das soll garantieren, dass es möglich ist, dass mehrere, nicht in Konflikt stehende Routen zur selben Zeit befahren werden können.

Der Testaufbau jedes Tests ist dabei in Abbildung 19 gezeigt.

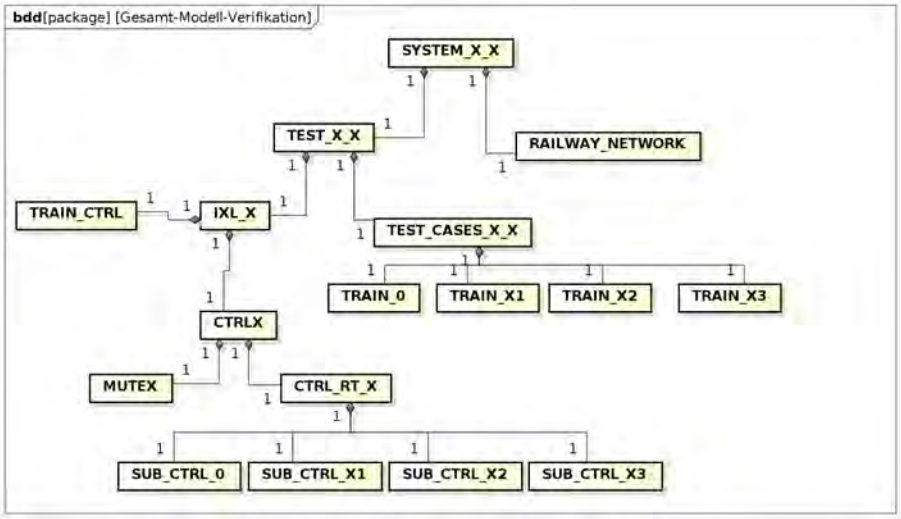

Abbildung 19: Testaufbau des Systemtests

Das Testsystem, hier `SYSTEM_X_X`, besteht aus den beiden CSP$_M$-Prozessen `RAILWAY_NETWORK`, welches das Gleisnetz beschreibt, und dem eigentlichen Testfall `TEST_X_X`. Dieser wiederum besteht aus den einzelnen Testfällen (`TEST_CASES_X_X`), den Zügen, die das Gleisnetz befahren, und aus dem Stellwerk (`IXL_X`). Die einzelnen Testfälle sind hierbei die CSP$_M$-Prozesse `TRAIN_0`, `TRAIN_X1` bis `TRAIN_X3`. Sobald das Stellwerk den Zügen eine movement-authority erteilt, befahren diese das Gleisnetz und erzeugen jeweils den Zug-Trace der jeweiligen Route, die diese befahren. Das Stellwerk besteht zum Einen aus dem `TRAIN_CTRL`, dem Train-Controller, der die Routenanfragen von den Zügen verarbeitet und daraufhin einen Sub-Controller startet, der die angefragte Route verwaltet. Der CSP$_M$-Prozess `CTRLX` bildet die Menge für diesen Tests benötigten Sub-Controller für die jeweiligen Routen, die befahren werden sollen. Diese Sub-Controller sind dargestellt durch `SUB_CTRL_0`, `SUB_CTRL_X1` bis `SUB_CTRL_X3`. Der CSP$_M$-Prozess `MUTEX` modelliert einen Mutex, sodass das Stellen von Weichen nur einem Sub-Controller zur Zeit gestattet ist.

`TRAIN_0` und `SUB_CTRL_0` sind hierbei jeweils für die valide Route zuständig. `TRAIN_0` befährt diese sowie `SUB_CTRL_0` stellt die Weichen in korrekter Position. Die anderen Zug-CSP$_M$-Prozesse `TRAIN_X1` bis `TRAIN_X3` befahren jeweils eine Konfliktroute oder eine nicht-Konfliktroute, sowie `SUB_CTRL_X1` bis `SUB_CTRL_X3` stellen dabei die Weichen und verwalten diese.

4.3.1 Route-Tests

Im ersten Schritt werden nun alle Routen der Stellwerkstabelle ohne Befahren von Konfliktrouten oder Nicht-Konfliktrouten befahren. Dabei fährt ein Zug pro Test auf dem Gleisnetz. Der Testaufbau ist wie in Abbildung 19 zu sehen, jedoch ohne die Züge `TRAIN_X1` bis `TRAIN_X3` und folglich ohne die Sub-Controller `SUB_CTRL_X1` bis `SUB_CTRL_X3`. Die Ausführung dieser Test hat den Zweck, dass hiermit geprüft wird, ob jede Route korrekt befahren werden kann. Dabei wird ein großer Fokus darauf gelegt, ob der CSP_M-Code-Generator die einzelnen Gleiselemente richtig, gemäß der Spezifikation miteinander synchronisiert hat, sodass die Routen auf dem Gleisnetz befahren werden können. Ein weiterer wichtiger Aspekt ist, dass nicht nur die richtigen Gleiselemente miteinander verbunden wurden, sondern auch an den richtigen Punkten verbunden wurden. Ist eine Weiche beispielsweise mit Plus anstatt mit Minus mit einer weiteren Weiche verbunden, so hat das enorme Folgen auf spätere System. Dadurch drohen große Unfälle und es kommt somit zu einer großen Katastrophe. Dadurch ist gerade dieser Test zu Beginn so wichtig.
Die Tests werden dabei lediglich mit dem generierten Gleisnetz ausgeführt. Jedoch wird das generierte Gleisnetz, wie in Abschnitt 4.1.1 beschrieben, umgeformt, sodass das generierte Gleisnetz dieselben Channel IDs besitzt wie das Referenzmodell. Insgesamt gibt es 96 Routen, folglich 96 Tests.
Nach erfolgreichen Ausführen der Tests gab es folgendes Ergebnis, wie in Listing 74 zu sehen.

```
1 $ cat Testergebnis.txt | grep "Passed" | wc -l
2 96
3
4 $ cat Testergebnis.txt | grep "Failed" | wc -l
5 0
```

Listing 74: Testergebnis der Route-Tests

Durch das Testergebnis aus Listing 74 ist zu sehen, dass alle Testergebnisse erfolgreich waren und kein Test fehlschlug. Dadurch ist gezeigt, dass jede Route aus der Stellwerkstabelle ohne Probleme auf das generierte Gleisnetz befahren werden kann. Danach ist jedes Gleiselement in dem generierten Gleisnetz richtig verbunden und zudem an jedem Punkt korrekt verbunden hinsichtlich der in der Stellwerkstabelle befindlichen Routen.

4.3.2 Konfliktrouten-Tests

Konfliktrouten sind in dem autonomen Stellwerk eine zentrale Komponente, die das sichere Befahren des Gleisnetzes mit mehreren Zügen erst ermöglichen. Wenn eine Weiche Bestandteil mehrerer Routen ist, dann dürfen diese Routen nicht simultan befahren werden. Andernfalls kommt es zu Zugkollisionen, da Weichen mehrfach gestellt werden. Möchte ein Zug eine bestimmte Route befahren, werden zunächst alle Weichen in die korrekte Position versetzt. Die Züge auf den Konfliktrouten

dürfen erst dann die Weichen für die eigene Route stellen, wenn der andere Zug seine Route verlässt. Sonst kann es sein, dass der wartende Zug die Weichen stellt und somit den aktuell fahrenden Zug entgleisen lässt. Damit das verhindert wird, existieren Konfliktrouten, die genau dann nicht gestellt und befahren werden dürfen. Um dieses Verhalten zu verifizieren, befährt ein Zug eine Route auf dem Gleisnetz. Jeweils drei andere Züge befahren eine Konfliktroute. Gemäß den Anforderungen, dass vier Züge das Gleisnetz befahren können, wird der Test für jede Route jeweils mit vier Zügen auf dem Gleisnetz ausgeführt.
Der Testaufbau ist in Abbildung 19 bezeigt. Die Züge `TRAIN_X1` bis `TRAIN_X3` befahren jeweils eine Konfliktroute sowie die Sub-Controller `SUB_CTRL_X1` bis `SUB_CTRL_X3` verwalten diese.
Das Ergebnis ist in Listing 75 zu sehen. Dabei ist klar zu erkennen, dass kein Test fehlschlug, alle endeten erfolgreich.

```
$ cat positive00.txt | grep "Passed" | wc -l
237
$ cat positive01.txt | grep "Passed" | wc -l
218
$ cat positive10.txt | grep "Passed" | wc -l
256
$ cat positive11.txt | grep "Passed" | wc -l
155

$ cat positive00.txt | grep "Failed" | wc -l
0
$ cat positive01.txt | grep "Failed" | wc -l
0
$ cat positive10.txt | grep "Failed" | wc -l
0
$ cat positive11.txt | grep "Failed" | wc -l
0
```

Listing 75: Ergebnis des Nicht-Konfliktrouten Tests

Daraus ergibt sich, dass Konflikrouten nie parallel zu den Routen der Stellwerkstabelle befahren werden. Somit ist das sichere Befahren der Routen auf dem Gleisnetz möglich.

4.3.3 Nicht-Konfliktrouten-Tests

Im finalen Schritt soll nun getestet werden, wie sich das Gleisnetz verhält, wenn vier Züge zur selben Zeit das Gleisnetz befahren. Dabei stehen die von den Zügen befahrenen Routen nicht in Konflikt, sodass diese Züge sich simultan über das Gleisnetz bewegen können. Wichtig ist dabei, dass auch mehrere Züge das Gleisnetz befahren können.
Der Testaufbau ist derselbe, wie in Abschnitt 4.3.2, gezeigt durch Abbildung 19. Wieder wird dabei jede Route der Stellwerkstabelle von einem Zug befahren sowie drei Züge befahren parallel eine Route, die nicht Bestandteil der Konfliktrouten ist. Die Züge `TRAIN_X1` bis `TRAIN_X3` befahren jeweils eine nicht in Konflikt stehende

Route sowie die Sub-Controller `SUB_CTRL_X1` bis `SUB_CTRL_X3` verwalten diese. Aufgeteilt wird der Test in vier CSP$_M$-Dateien zur Performanzoptimierung. Das Ergebnis ist in Listing 76 gezeigt.

```
$ cat negative00.txt | grep "Passed" | wc -l
562
$ cat negative01.txt | grep "Passed" | wc -l
714
$ cat negative10.txt | grep "Passed" | wc -l
421
$ cat negative11.txt | grep "Passed" | wc -l
485

$ cat negative00.txt | grep "Failed" | wc -l
0
$ cat negative01.txt | grep "Failed" | wc -l
0
$ cat negative10.txt | grep "Failed" | wc -l
0
$ cat negative11.txt | grep "Failed" | wc -l
0
```

Listing 76: Ergebnis des Nicht-Konfliktrouten Tests

Hierbei ist zu sehen, dass kein Test fehlschlägt. Somit können auf dem generierten Gleisnetz vier Züge, deren Routen nicht in Konflikt stehen, simultan befahren werden. Wie zu erwarten ist das Ergebnis des Test auch hier erfolgreich, wie auf dem Referenzgleisnetz. Dieses Ergebnis zeigt wiederum, dass das Referenzgleisnetz äquivalent zu dem generierten Gleisnetz ist.

4.4 Auswertung der Tests

In diesem Kapitel wurde der zuvor implementierte CSP$_M$-Code-Generator auf Korrektheit verifiziert. Dieser hat die Aufgabe aus einer Gleisnetz-Spezifikation in Form einer XML-Datei äquivalenten CSP$_M$-Code zu generieren. Die Aufgabe in diesem Kapitel bestand darin, das TEAMOD-Gleisnetz und den inneren Kreis des TEAMOD-Gleisentzes aus [Brü20] in XML zu modellieren und den durch den CSP$_M$-Code-Generator generierten korrespondierenden CSP$_M$-Code zu verifizieren. In dem Kontext sollte geprüft werden, ob das generierte Gleisnetz äquivalent zu der händischen, schon verifizierten Modellierung der Gleisnetze aus [Brü20] ist. Dazu wurden verschiedene Techniken herangezogen, die schlussendlich die Äquivalenz der generierten Gleisnetze mit der händischen Referenzversion und somit die Korrekt des CSP$_M$-Code-Generators zeigen.
Zunächst wurde mithilfe von FDR4 geprüft, ob die generierten Gleisnetze äquivalent zu den generierten Gleisnetzen sind durch Test auf Failures Refinement. Diese Methode liefert einen Beweis darüber, ob beide Modelle äquivalent sind. Der große Nachteil ist der große Zustandsraum einiger CSP$_M$-Modelle. Das kleine Gleisnetz, welches den inneren Kreis des TEAMOD-Gleisnetzes darstellt, konnte mit vertretbaren Aufwand mithilfe der Failures Refinement erfolgreich verifiziert werden. Die

Äquivalenz zum Referenzmodell wurde nachgewiesen (siehe Abschnitt 4.1.2). Bei dem TEAMOD-Gleisnetz hingegen gab es einen zu großen Zustandsraum während des Prüfens auf Failures Refinement. Die Folge war, dass kein akzeptables Ergebnis innerhalb vertretbaren Aufwandes erzielt werden konnte. Dach wurde das Kompositionelle Model-Checking angewendet. Auch mit dem Verfahren konnte nicht das gesamte Gleisnetz verifiziert werden. Daher wurde die Strategie des Property-Kataloges herangezogen. Dieser Test beruht auf das Aufstellen möglicher Zug-Traces im Gleisnetz, ersetzt jedoch keinen vollständigen Äquivalenzvergleich. Wenn jedoch beide Gleisnetze auf allen möglichen Zugrouten äquivalent reagieren, kann daraus abgeleitet werden, dass beide im Bereich Sicherheit äquivalent sind.
Der CSP$_M$-Trace-Generator bietet eine zusätzliche Verifikationsmethode und setzt dort an, indem dieser implementiert wurde, um alle möglichen Zug-Pfade (Traces) innerhalb des Gleisnetzes zu ermitteln und diese in eine Trace-Datei auszugeben (Abschnitt 4.2). Als Gegenstück dazu dient der Gleisnetz-Interpreter, der die Trace-Datei und ein zu testendes Gleisnetz einliest und prüft, ob jeder einzelne Trace in dem Gleisnetz enthalten ist. Diese Methode beschränkt sich ebenfalls nur auf Weichenstellungen und Zug-Pfade im Gleisnetz, ist für den Äquivalenztest für Gleisnetze völlig akzeptabel. Denn genau diese Pfade zeigen, dass der Zug dieselben Routen sowohl auf der Referenzimplementierung des Gleisnetzes wie auch auf der Implementierung des generierten Gleisnetzes befahren kann.
Ebenfalls wurden alle bestehenden Testfälle aus [Brü20] erneut auf das generierte Gleisnetz ausgeführt. Diese Testfälle repräsentieren den Realbetrieb des Gleisnetzes, indem alle Routen aus der Interlocking-Table auf das Gleisnetz ausgeführt wurden. Diese sind jedoch nicht erschöpfend und steigern daher nur die Glaubwürdigkeit der zuvor ausgeführten Tests. Die Tests endeten wie erwartet erfolgreich und wurde vom generierten Gleisnetz bestanden (siehe Abschnitt 4.3).
Es kann somit eine direkt Aussage darüber getroffen werden, ob der CSP$_M$-Code-Generator korrekten CSP$_M$-Code erzeugt. Aufgrund der vielen Tests und der hohen Testestärke kombinierter Prüfverfahren kann klar gesagt werden, dass der CSP$_M$-Code-Generator korrekten Code erzeugt. Das wird untermauert durch die Verifikation mit den genannten Verifikationsmethoden. Dadurch kann mithilfe des CSP$_M$-Code-Generators aus einer XML-Spezifikation schnell CSP$_M$-Code generiert werden, der zur Verifikation von Eisenbahnsystemen genutzt werden kann.

KAPITEL 5

Evaluation

Das Ziel, einen CSP_M-Code-Generator zu entwickeln, wurde erreicht. Dazu wurde anhand aussagekräftiger Verfahren bewiesen, dass der produzierte CSP_M-Code äquivalent zu den Referenzmodellen ist. In [Brü20] wurden die hier verwendeten Gleisnetze händisch in CSP_M implementiert und durch geeignete Methoden erfolgreich verifiziert. Diese dienen für diese Arbeit als Referenzmodelle zum Prüfen, ob der CSP_M-Code-Generator äquivalenten Code produziert. Ist genau das der Fall, so ist gezeigt, dass dieser validen Code erzeugen kann.

Die einfachste Möglichkeit ist, dass Tests auf Failures-Refinement durch FDR4 ausgeführt werden. Diese prüfen, ob ein CSP_M-Prozess dieselbe Fehlermenge besitzt wie der zu testende. Dadurch konnte die Failures-Refinement zwischen der Referenzimplementierung des kleinen Gleisnetzes und des generierten Gleisnetzes bewiesen werden. Zudem wurde ein Robustheitstest ausgeführt, indem auf Failures Refinement in entgegengesetzter Richtung geprüft wurde. Die Äquivalenz zur Referenzimplementierung wurde erfolgreich nachgewiesen. Das TEAMOD-Gleisnetz konnte aufgrund von einem zu großen Zustandsraum nicht mithilfe Failures Refinement verifiziert werden. Dazu wurde versucht, mithilfe der kompositionellen Verifikation die Äquivalenz nachzuweisen. Das konnte jedoch nur für einige Teile des Gleisnetzes beweisen werden. Um das gesamte Gleisnetz zu verifizieren, wurde nun ein Property-Katalog aufgestellt, der alle Eigenschaften enthält, die von beiden Gleisnetzen erfüllt werden müssen. Dabei wurde erfolgreich nachgewiesen, dass beide Gleisnetze dieselbe Anzahl Zug-Pfade im Gleisnetz zulassen. Dabei wurde gezeigt, dass beide Gleisnetze in jeder kritischen Sektion äquivalent reagieren. Um die Verifikationsstärke zu erhöhen, wurde schließlich der Gleisnetz-Interpreter und Gleisnetz-Trace-Generator entwickelt, der alle Traces aus einer Gleisnetz-Spezifikation extrahiert. Der Gleisnetz-Interpreter prüft schließlich alle extrahierten Traces gegen eine Spezifikation. Dadurch wurde gezeigt, dass alle möglichen Zug-Traces mit dafür nötiger Weichenstellungen ein beiden Gleisnetzes gleich sind. Schließlich wurden alle Tests aus [Brü20] erneut auf das nun generierte Gleisnetz ausgeführt, sodass zudem gezeigt werden konnte, dass die TEAMOD-Stellwerkstabelle ohne Probleme mit der

generierten Version des Gleisnetzes verifiziert werden kann. Die Ergebnisse aus der Bachelor-Thesis waren identisch mit den Ergebnissen der Tests mit dem generierten Gleisnetz.
Alles in allem wurde der CSP_M-Code-Generator für den Fall ausreichend getestet, dass der generierte Code dem Referenzmodell voll und ganz äquivalent ist. Es konnte durch eine hohe Verifikationsstärke gezeigt werden, dass generierte Gleisnetze aus der XML-Spezifikation dem Referenzmodell der händischen Implementierung entsprechen. Zum Ausführen der Verifikation mithilfe von FDR4 wurde der Service der Google Cloud Plattform verwendet, der das stündliche Mieten großer Server ermöglicht. So wurden viele Test auf einem Intel(R) Xeon(R) Gold 6242 CPU @ 2.8 GHz mit 16 Kernen und 128GB DDR4 Arbeitsspeicher ausgeführt, einige auf einem Notebook mit einem 64-Bit Linux Debian 10 System mit einem Intel Core i5-8250U mit vier physischen und acht logischen Prozessor-Kernen sowie 16GB RAM. [Goo21] Die Tabelle 5 zeigt einige Statistiken der Testausführung. Die Angabe *Sek* steht dabei für Sekunden, *Min* für Minuten.

Test	System	Dauer
Test auf Failures Refinement kleines Gleisnetz mit FDR4	Notebook	ca. 10 Min.
Test auf Failures Refinement TEAMOD-Gleisnetz mit FDR4	Server	Error
Property-Katalog	Notebook	ca. 30 Sek.
Kompositionelles Model-Checking	Notebook	ca. $<$ 10 Sek.
Trace-Generator TEAMOD-Gleisnetz mit 1145 Traces (Länge 105, Start 13)	Notebook	ca. 7,5 Min.
Gleisnetz-Interpreter TEAMOD-Gleisnetz mit 1145 Traces (Länge 105, Start 13)	Notebook	ca. 30 Sek.
Trace-Generator TEAMOD-Gleisnetz mit 1316 Traces (Länge 105, Start 26)	Notebook	ca. 8 Min.
Gleisnetz-Interpreter TEAMOD-Gleisnetz mit 1316 Traces (Länge 105, Start 26)	Notebook	ca. 30 Sek.
Gesamtsystem: Routen	Server	ca. 26 Sek.
Gesamtsystem: Konflikte 1	Server	ca. 17 Min.
Gesamtsystem: Konflikte 2	Server	ca. 10 Min.
Gesamtsystem: Konflikte 3	Server	ca. 16 Min.
Gesamtsystem: Konflikte 4	Server	ca. 9 Min.
Gesamtsystem: Nicht-Konflikte 1	Server	ca. 4 Min.
Gesamtsystem: Nicht-Konflikte 2	Server	ca. 5 Min.
Gesamtsystem: Nicht-Konflikte 3	Server	ca. 3 Min.
Gesamtsystem: Nicht-Konflikte 4	Server	ca. 3,5 Min.

Tabelle 1: Benchmarkergebnisse

KAPITEL 6

Fazit und Ausblick

In dieser Arbeit wurde der CSP_M-Code-Generator zum automatischen Generieren von CSP_M-Code aus einer XML-Spezifikation vorgestellt. Dieser ermöglicht es, dass Gleisnetze nicht mehr direkt in CSP_M modelliert werden müssen, was eine erhebliche Zeitersparnis im Bereich der Modellierung zur Folge hat. Stattdessen ist es nur noch nötig, Gleisnetze in XML zu modellieren. Der Editor oXygen XML-Editor unterstützt diesen Prozess dabei stark. [SRL21] Die Darstellung in XML ist dabei übersichtlicher und verhindert von dem Menschen gemachte Fehler, sodass das Prüfen auf korrekter Modellierung zuvor einfach gestaltet wird. Eine Modellierung in XML ist beipielsweise in Listing 84 bis Listing 89 zu sehen. Ebenso wurde die Korrektheit des CSP_M-Code-Generators durch geeignete und aussagekräftige Methoden auf Korrektheit geprüft, sodass stets gewährleistet ist, dass dieser zu der Spezifikation äquivalenten Code generiert.

Dieses Programm leistet bei der Verifikation einen erheblichen Beitrag, da das aufwendige Modellieren einer Gleisnetzspezifikation in CSP_M sehr mühselig sein kann und durch den Einsatz des Code-Generators nicht mehr nötig ist. Menschliche Fehler, die ein falsches Verifikationsergebnis zur Folge hätten, können somit größtenteils vermieden werden. Wie im Abschnitt 3.1 zu sehen, wird schließlich ein Gleisnetz durch die viel einfachere Darstellung modelliert und dann CSP_M-Code generiert. Der Code-Generator hat zudem den Vorteil, dass Änderungen schnell verarbeitet und dann ein neues Gleisnetz in CSP_M generiert werden kann. Langwieriges und fehleranfälliges Modellieren in CSP_M ist somit nicht mehr möglich.

Eine weitere Arbeit in diesem Kontext wurde von Matthias Lange verfasst, dessen Code-Generator dieselbe XML-Konvention aus Abschnitt 3.1 nutzt und daraus äquivalenten nuXmv-Code generiert. [Lan21] Der Model-Checker nuXmv arbeitet nach dem klassischen Model-Checking und nutzt einen anderen Formalismus Modelle zu modellieren. Üblicherweise ist der Code von nuXmv um Weiten länger als der einer CSP_M-Spezifikation. Der von ihm implementierte Code-Generator benötigt zu der Gleisnetz-Spezifikation in Form einer XML-Datei zudem eine Stellwerkstabelle in Form einer CSV-Datei, um die für nuXmv nötige SMV-Datei zu erstellen. Dabei ist

jedes Gleisnetz als Finite State Machine beschrieben. Matthias verifiziert den Code-Generator ähnlich wie in dieser Arbeit durch Prüfen auf Trace-Äquivalenz, indem Transitionen aus der Referenz-FSM extrahiert werden und mit denen des generierten Gleisnetzes verglichen werden. Zur Robustheit wird das auch in anderer Richtung ausgeführt. Abschließend wurden mit beiden Arbeiten zwei unterschiedliche Lösungen entwickelt, wie Gleisnetze einfacher beschrieben werden können, um hinterher äquivalenten CSP_M-Code, wie in dieser Arbeit oder SMV-Code zu generieren, wie in der Arbeit von Matthias Lange.
Der Code-Generator dieser Arbeit sowie der von Matthias Lange bilden die Grundlage für weitere Automatisierungen. Weiter könnten beispielsweise Modelle einzelner Gleiselemente durch SysML-Modellierungen beschrieben werden, um daraus CSP_M-, oder SMV-Code zu generieren. [Gro17] Ebenso ist es auch möglich, die Gleisnetzspezifikation in Form einer XML-Datei durch eine GUI zu erstellen, indem der Nutzer Gleiselemente visuell verbindet und somit das zu verifizierende Gleisnetz modelliert. In einem Schritt kann dann das Modell entweder in CSP_M oder SMV modelliert werden. Ebenso ist es in Zukunft hilfreich, Requirements direkt modellieren und zu generieren, sodass das Verifizieren in einem Schritt möglich ist. Folglich gibt es noch viele weitere Ansätze, in denen ein Code-Generator die Verifikationsarbeit erleichtern kann. Ferner bildet der Code-Generator dieser Arbeit den Grundstein für eine neue Art Gleisnetze mithilfe von CSP_M zu verifizieren. Das Paper [PHC19] liefert hierbei den Ansatz, dass der von dem Code-Generator generierten CSP_M-Code zur Generierung von Testmengen für das Model-Basierte Testen genutzt werden kann. Dieses Paper liefert somit den theoretischen Ansatz für weitere automatisierte Verifikationslösungen der Zukunft.

Literaturverzeichnis

[Bas20] Jacques Basaldúa. Formal fields: A framework to automate code generation across domains. *CoRR*, abs/2007.14075, 2020.

[BDF+17] Michael J. Butler, Dana Dghaym, Tomas Fischer, Thai Son Hoang, Klaus Reichl, Colin F. Snook, and Peter Tummeltshammer. Formal modelling techniques for efficient development of railway control products. In Alessandro Fantechi, Thierry Lecomte, and Alexander B. Romanovsky, editors, *Reliability, Safety, and Security of Railway Systems. Modelling, Analysis, Verification, and Certification - Second International Conference, RSSRail 2017, Pistoia, Italy, November 14-16, 2017, Proceedings*, volume 10598 of *Lecture Notes in Computer Science*, pages 71–86. Springer, 2017.

[BHL+19] Felix Brüning, Raven Hölting, Matthias Lange, Thomas Lipps, Tom Niewöhner, Jan R. Kropp, Lars Forquignon, Jan Leuschner, Felix Kohlhase, and Nikas Kandsorra. Projektbericht Bachelorprojekt TEAMOD. techreport, Universität Bremen, AG Betriebssysteme-Verteilte Systeme, 2019.

[Brü20] Felix Brüning. Model Checking eines Stellwerksalgorithmus mit FDR4. https://www.szi.uni-bremen.de/wp-content/uploads/2020/03/thesis1_compressed.pdf, 2020.

[Bur] Jonathan Burton. Compositional verification of a network of csp processes: using fdr2 to verify refinement in the event of interface difference.

[GABR16] Thomas Gibson-Robinson, Philip J. Armstrong, Alexandre Boulgakov, and A. W. Roscoe. FDR3: a parallel refinement checker for CSP. *Int. J. Softw. Tools Technol. Transf.*, 18(2):149–167, 2016.

[Goo21] Google. Google Cloud Platform. website, 2021. https://cloud.google.com/, Mai 2021.

[GP21] GNOME-Project. The XML C parser and toolkit of Gnome, 2021.

[Gro15] Object Management Group. OMG Unified Modeling Language 2.5, 2015. https://www.omg.org/spec/UML/2.5/PDF.

[Gro17] Object Management Group. OMG System Modeling Language 1.5, Mai 2017. https://www.omg.org/spec/SysML/1.5/PDF.

[Hax12] Anne E. Haxthausen. Automated generation of safety requirements from railway interlocking tables. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Applications and Case Studies - 5th International Symposium, ISoLA 2012, Heraklion, Crete, Greece, October 15-18, 2012, Proceedings, Part II*, volume 7610 of *Lecture Notes in Computer Science*, pages 261–275. Springer, 2012.

[Hax14] Anne E. Haxthausen. An institution for imperative RSL specifications. In Shusaku Iida, José Meseguer, and Kazuhiro Ogata, editors, *Specification, Algebra, and Software - Essays Dedicated to Kokichi Futatsugi*, volume 8373 of *Lecture Notes in Computer Science*, pages 441–464. Springer, 2014.

[Hax21] Anne E. Haxthausen. Robustrails wp 4.1 formal development and verification of railway control systems, 2021.

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

[IBM21] IBM. IBM Rational Rhapsody, 2021.

[JP19] Elena Gorbachuk Jan Peleska, Wen-ling Huang. Theory of reactive systems - lecture skript, 2012 - 2019.

[Lan19] Matthias Lange. IXL-Modelchecking mit nuXmv. https://www.szi.uni-bremen.de/wp-content/uploads/2020/01/bachelorarbeit_matthias_lange_compressed.pdf, 2019.

[Lan21] Matthias Lange. Automatische Generierung von Gleisnetzmodellen in nuXmv aus XML-Spezifikationen mit libxml2. Master thesis, Universität Bremen, 2021.

[LBK+20] Matthias Lange, Felix Brüning, Jan R. Kropp, Jan Leuschner, Patrick Wilde, and Tobias Wegner. Projektbericht Masterprojekt TEAMOD. techreport, Universität Bremen, AG Betriebssysteme-Verteilte Systeme, 2020.

[LF08] Michael Leuschel and Marc Fontaine. Probing the depths of CSP-M: A new fdr-compliant validation tool. In Shaoying Liu, T. S. E. Maibaum, and Keijiro Araki, editors, *Formal Methods and Software Engineering, 10th International Conference on Formal Engineering Methods, ICFEM 2008, Kitakyushu-City, Japan, October 27-31, 2008. Proceedings*, volume 5256 of *Lecture Notes in Computer Science*, pages 278–297. Springer, 2008.

[MFH17] Hugo Daniel Macedo, Alessandro Fantechi, and Anne E. Haxthausen. Compositional model checking of interlocking systems for lines with multiple stations. In Clark W. Barrett, Misty Davies, and Temesghen Kahsai, editors, *NASA Formal Methods - 9th International Symposium, NFM 2017, Moffett Field, CA, USA, May 16-18, 2017, Proceedings*, volume 10227 of *Lecture Notes in Computer Science*, pages 146–162, 2017.

[MGR08] Nick Moffat, Michael Goldsmith, and Bill Roscoe. A representative function approach to symmetry exploitation for CSP refinement checking. In Shaoying Liu, T. S. E. Maibaum, and Keijiro Araki, editors, *Formal Methods and Software Engineering, 10th International Conference on Formal Engineering Methods, ICFEM 2008, Kitakyushu-City, Japan, October 27-31, 2008. Proceedings*, volume 5256 of *Lecture Notes in Computer Science*, pages 258–277. Springer, 2008.

[Pel20] Jan Peleska. Specification of embedded systems, 2020.

[PHC19] Jan Peleska, Wen-ling Huang, and Ana Cavalcanti. Finite complete suites for CSP refinement testing. *Sci. Comput. Program.*, 179:1–23, 2019.

[Pro21] Eclipse Project. Eclipse papyrus, 2021.

[RBP19] Elie Richa, Etienne Borde, and Laurent Pautet. Translation of ATL to AGT and application to a code generator for simulink. *Softw. Syst. Model.*, 18(1):321–344, 2019.

[SRL21] Syncro Soft SRL. oXygen XML Editor 23.1, 2021. https://www.oxygenxml.com/xml_editor.html.

[TB08] C. M. Sperberg-McQueen Eve Maler François Yergeau Tim Bray, Jean Paoli. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. 2008.

[VHP17] Linh H. Vu, Anne E. Haxthausen, and Jan Peleska. A domain-specific language for generic interlocking models and their properties. In Alessandro Fantechi, Thierry Lecomte, and Alexander B. Romanovsky, editors, *Reliability, Safety, and Security of Railway Systems. Modelling, Analysis, Verification, and Certification - Second International Conference, RSSRail 2017, Pistoia, Italy, November 14-16, 2017, Proceedings*, volume 10598 of *Lecture Notes in Computer Science*, pages 99–115. Springer, 2017.

[WW09] Heike Wehrheim and Daniel Wonisch. Compositional csp traces refinement checking. *Electronic Notes in Theoretical Computer Science*, 250:135–151, 09 2009.

Anhang

```
1  ---------------------------------
2  -- Definition of cross-section
3  ---------------------------------
4
5  -- straight-left     := s0
6  -- cross-left        := s1
7  -- straight-right    := s2
8  -- cross-right       := s3
9
10 CROSSING(id, s0, s1, s2, s3) =
11   reqsec.id -> MARKED(id, s0, s1, s2, s3)
12   []
13     ([] x : {s0, s1, s2 ,s3} @ rail.x -> crash.id -> STOP)
14
15 MARKED(id, s0, s1, s2, s3) =
16   set.id.str -> LOCKED_STR(id, s0, s1, s2, s3)
17   []
18   set.id.crss -> LOCKED_CRSS(id, s0, s1, s2, s3)
19   []
20     ([] x : {s0, s1, s2 ,s3} @ rail.x -> crash.id -> STOP)
21   []
22   reqsec.id -> derailing.id -> STOP
23
24 LOCKED_STR(id, s0, s1, s2, s3) =
25   rail.s0 -> LOCKED_STR_OCC_L2R_UP(id, s0, s1, s2, s3)
26   []
27   rail.s1 -> LOCKED_STR_OCC_L2R_DWN(id, s0, s1, s2, s3)
28   []
29   rail.s2 -> LOCKED_STR_OCC_R2L_DWN(id, s0, s1, s2, s3)
30   []
31   rail.s3 -> LOCKED_STR_OCC_R2L_UP(id, s0, s1, s2, s3)
32   []
33   release.id -> CROSSING(id, s0, s1, s2, s3)
34   []
35   reqsec.id -> derailing.id -> STOP
36   []
37   set.id.str -> derailing.id -> STOP
38   []
39   set.id.crss -> derailing.id -> STOP
```

Listing 77: CSP-Code der Kreuzweiche Teil 1

```
LOCKED_CRSS(id, s0, s1, s2, s3) =
  rail.s0 -> LOCKED_CRSS_OCC_L2R_DWN(id, s0, s1, s2, s3)
  []
  rail.s1 -> LOCKED_CRSS_OCC_L2R_UP(id, s0, s1, s2, s3)
  []
  rail.s2 -> LOCKED_CRSS_OCC_R2L_UP(id, s0, s1, s2, s3)
  []
  rail.s3 -> LOCKED_CRSS_OCC_R2L_DWN(id, s0, s1, s2, s3)
  []
  release.id -> CROSSING(id, s0, s1, s2, s3)
  []
  reqsec.id -> derailing.id -> STOP
  []
  set.id.str -> derailing.id -> STOP
  []
  set.id.crss -> derailing.id -> STOP

-------------------------------------------------------------------------------

-- straight
-------------------------------------------------------------------------------

-- from s2
LOCKED_STR_OCC_R2L_DWN(id, s0, s1, s2, s3) =
  rail.s0 -> LOCKED_STR(id, s0, s1, s2, s3)
  []
    ([] x : {s1, s2, s3} @ rail.x -> derailing.id -> crash.id -> STOP)
  []
  reqsec.id -> derailing.id -> STOP
  []
  set.id.str -> derailing.id -> STOP
  []
  set.id.crss -> derailing.id -> STOP

--from s0
LOCKED_STR_OCC_L2R_UP(id, s0, s1, s2, s3) =
    ([] x : {s0, s1, s3} @ rail.x -> derailing.id -> crash.id -> STOP)
  []
  rail.s2 -> LOCKED_STR(id, s0, s1, s2, s3)
  []
  reqsec.id -> derailing.id -> STOP
  []
  set.id.str -> derailing.id -> STOP
  []
  set.id.crss -> derailing.id -> STOP

-- from s1
LOCKED_STR_OCC_L2R_DWN(id, s0, s1, s2, s3) =
    ([] x : {s0,s1,s2} @ rail.x -> derailing.id -> crash.id -> STOP)
  []
  rail.s3 -> LOCKED_STR(id, s0, s1, s2, s3)
  []
  reqsec.id -> derailing.id -> STOP
  []
  set.id.str -> derailing.id -> STOP
  []
  set.id.crss -> derailing.id -> STOP
```

Listing 78: CSP-Code der Kreuzweiche Teil 2

```
-- from s3
LOCKED_STR_OCC_R2L_UP(id, s0, s1, s2, s3) =
    ([] x : {s0,s3,s2} @ rail.x -> derailing.id -> crash.id -> STOP)
  []
  rail.s1 -> LOCKED_STR(id, s0, s1, s2, s3)
  []
  reqsec.id -> derailing.id -> STOP
  []
  set.id.str -> derailing.id -> STOP
  []
  set.id.crss -> derailing.id -> STOP

-------------------------------------------------------------------------------

-- Cross
-------------------------------------------------------------------------------

-- from s1
LOCKED_CRSS_OCC_L2R_UP(id, s0, s1, s2, s3) =
    ([] x : {s0, s1, s3} @ rail.x -> derailing.id -> crash.id -> STOP)
  []
  rail.s2 -> LOCKED_CRSS(id, s0, s1, s2, s3)
  []
  reqsec.id -> derailing.id -> STOP
  []
  set.id.str -> derailing.id -> STOP
  []
  set.id.crss -> derailing.id -> STOP

-- from s2
LOCKED_CRSS_OCC_R2L_UP(id, s0, s1, s2, s3) =
  rail.s1 -> LOCKED_CRSS(id, s0, s1, s2, s3)
  []
    ([] x : {s0, s2, s3} @ rail.x -> derailing.id -> crash.id -> STOP)
  []
  reqsec.id -> derailing.id -> STOP
  []
  set.id.str -> derailing.id -> STOP
  []
  set.id.crss -> derailing.id -> STOP

-- from s0
LOCKED_CRSS_OCC_L2R_DWN(id, s0, s1, s2, s3) =
    ([] x : {s0, s1, s2} @ rail.x -> derailing.id -> crash.id -> STOP)
  []
  rail.s3 -> LOCKED_CRSS(id, s0, s1, s2, s3)
  []
  reqsec.id -> derailing.id -> STOP
  []
  set.id.str -> derailing.id -> STOP
  []
  set.id.crss -> derailing.id -> STOP
```

Listing 79: CSP-Code der Kreuzweiche Teil 3

```
-- from s3
LOCKED_CRSS_OCC_R2L_DWN(id, s0, s1, s2, s3) =
    ([] x : {s3, s1, s2} @ rail.x -> derailing.id -> crash.id -> STOP)
  []
  rail.s0 -> LOCKED_CRSS(id, s0, s1, s2, s3)
  []
  reqsec.id -> derailing.id -> STOP
  []
  set.id.str -> derailing.id -> STOP
  []
  set.id.crss -> derailing.id -> STOP

--------------------------------------------------------------------------------
```

Listing 80: CSP-Code der Kreuzweiche Teil 4

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

Listing 81: CSP-Code der Weiche Teil 1

```
P_LOCKED_CRSS(id, s0, s1, s2) =
  rail.s0 -> P_LOCKED_CRSS_OCC_FROM_LEFT(id, s0, s1, s2)
  []
  rail.s1 -> P_LOCKED_CRSS_OCC_FROM_LEFT2(id, s0, s1, s2)
  []
  rail.s2 -> derailing.id -> STOP
  []
  release.id -> POINT(id, s0, s1, s2)
  []
  reqsec.id -> derailing.id -> STOP
  []
  set.id.str -> derailing.id -> STOP
  []
  set.id.crss -> derailing.id -> STOP


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

P_LOCKED_CRSS_OCC_FROM_LEFT2(id, s0, s1, s2) =
  rail.s0 -> P_LOCKED_CRSS(id, s0, s1, s2)
  []
    ([] x :{s1, s2} @ rail.x -> derailing.id -> crash.id -> STOP)
  []
  reqsec.id -> derailing.id -> STOP
  []
  set.id.str -> derailing.id -> STOP
  []
  set.id.crss -> derailing.id -> STOP
```

Listing 82: CSP-Code der Weiche Teil 2

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

Listing 83: CSP-Code eines Track-Elementes

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!DOCTYPE rail_decl [
  <!ELEMENT network (#PCDATA)>
  <!ELEMENT section (#PCDATA)>
  <!ELEMENT point (#PCDATA)>
  <!ELEMENT crossover (#PCDATA)>
  <!ELEMENT connection (#PCDATA)>
]>
<network id="teamod_gleisnetz">
  <section id="t201">
    <connection ref="t219" side="up" />
    <connection ref="p13" side="down" />
  </section>
  <section id="t200">
    <connection ref="t218" side="up" />
    <connection ref="p14" side="down" />
  </section>
  <section id="t202">
    <connection ref="t220" side="up" />
    <connection ref="p12" side="down" />
  </section>
  <section id="t203">
    <connection ref="p13" side="up" />
    <connection ref="p12" side="down" />
  </section>
  <section id="t204">
    <connection ref="p13" side="up" />
    <connection ref="c10" side="down" />
  </section>
  <section id="t205">
    <connection ref="p14" side="up" />
    <connection ref="c9" side="down" />
  </section>
  <section id="t206">
    <connection ref="p14" side="up" />
    <connection ref="p8" side="down" />
  </section>
  <section id="t207">
    <connection ref="p7" side="up" />
    <connection ref="p28" side="down" />
  </section>
  <section id="t208">
    <connection ref="p7" side="up" />
    <connection ref="p5" side="down" />
  </section>
  <section id="t209">
    <connection ref="c10" side="up" />
    <connection ref="p5" side="down" />
  </section>
  <section id="t210">
    <connection ref="c9" side="up" />
    <connection ref="p6" side="down" />
  </section>
```

Listing 84: XML-Repräsentation des TEAMOD-Gleisnetzes Teil 1

```
<section id="t211">
  <connection ref="p8" side="up" />
  <connection ref="p6" side="down" />
</section>
<section id="t212">
  <connection ref="p5" side="up" />
  <connection ref="p3" side="down" />
</section>
<section id="t213">
  <connection ref="p24" side="up" />
  <connection ref="p1" side="down" />
</section>
<section id="t214">
  <connection ref="c23" side="up" />
  <connection ref="p2" side="down" />
</section>
<section id="t215">
  <connection ref="p21" side="up" />
  <connection ref="p24" side="down" />
</section>
<section id="t216">
  <connection ref="p22" side="up" />
  <connection ref="c23" side="down" />
</section>
<section id="t217">
  <connection ref="p26" side="up" />
  <connection ref="c23" side="down" />
</section>
<section id="t218">
  <connection ref="p16" side="up" />
  <connection ref="t200" side="down" />
</section>
<section id="t219">
  <connection ref="p15" side="up" />
  <connection ref="t201" side="down" />
</section>
<section id="t220">
  <connection ref="p25" side="up" />
  <connection ref="t202" side="down" />
</section>
<section id="t221">
  <connection ref="p6" side="up" />
  <connection ref="p4" side="down" />
</section>
<section id="t100">
  <connection ref="p27" side="up" />
</section>
<section id="t101">
  <connection ref="p27" side="up" />
</section>
<section id="t102">
  <connection ref="p28" side="up" />
</section>
<section id="t103">
  <connection ref="t107" side="up" />
  <connection ref="p21" side="down" />
</section>
```

Listing 85: XML-Repräsentation des TEAMOD-Gleisnetzes Teil 2

```
<section id="t104">
  <connection ref="t108" side="up" />
  <connection ref="p21" side="down" />
</section>
<section id="t105">
  <connection ref="t109" side="up" />
  <connection ref="p22" side="down" />
</section>
<section id="t106">
  <connection ref="t110" side="up" />
  <connection ref="p22" side="down" />
</section>
<section id="t107">
  <connection ref="p20" side="up" />
  <connection ref="t103" side="down" />
</section>
<section id="t108">
  <connection ref="p20" side="up" />
  <connection ref="t104" side="down" />
</section>
<section id="t109">
  <connection ref="p19" side="up" />
  <connection ref="t105" side="down" />
</section>
<section id="t110">
  <connection ref="p19" side="up" />
  <connection ref="t106" side="down" />
</section>
<section id="t111">
  <connection ref="p25" side="up" />
  <connection ref="t113" side="down" />
</section>
<section id="t112">
  <connection ref="p25" side="up" />
  <connection ref="t114" side="down" />
</section>
<section id="t113">
  <connection ref="t111" side="up" />
  <connection ref="p26" side="down" />
</section>
<section id="t114">
  <connection ref="t112" side="up" />
  <connection ref="p26" side="down" />
</section>
<point id="p1">
  <connection ref="p4" side="plus"/>
  <connection ref="p2" side="minus"/>
  <connection ref="t213" side="stem"/>
</point>
<point id="p2">
  <connection ref="t214" side="plus"/>
  <connection ref="p1" side="minus"/>
  <connection ref="p3" side="stem"/>
</point>
```

Listing 86: XML-Repräsentation des TEAMOD-Gleisnetzes Teil 3

```
<point id="p3">
  <connection ref="p4" side="minus"/>
  <connection ref="t212" side="plus"/>
  <connection ref="p2" side="stem"/>
</point>
<point id="p4">
  <connection ref="p3" side="minus"/>
  <connection ref="p1" side="plus"/>
  <connection ref="t221" side="stem"/>
</point>
<point id="p5">
  <connection ref="t208" side="minus"/>
  <connection ref="t209" side="plus"/>
  <connection ref="t212" side="stem"/>
</point>
<point id="p6">
  <connection ref="t211" side="minus"/>
  <connection ref="t210" side="plus"/>
  <connection ref="t221" side="stem"/>
</point>
<point id="p7">
  <connection ref="t207" side="minus"/>
  <connection ref="t208" side="plus"/>
  <connection ref="p11" side="stem"/>
</point>
<point id="p8">
  <connection ref="c9" side="minus"/>
  <connection ref="t206" side="plus"/>
  <connection ref="t211" side="stem"/>
</point>
<point id="p11">
  <connection ref="c10" side="minus"/>
  <connection ref="p7" side="plus"/>
  <connection ref="p12" side="stem"/>
</point>
<point id="p12">
  <connection ref="t202" side="minus"/>
  <connection ref="t203" side="plus"/>
  <connection ref="p11" side="stem"/>
</point>
<point id="p13">
  <connection ref="t203" side="minus"/>
  <connection ref="t204" side="plus"/>
  <connection ref="t201" side="stem"/>
</point>
<point id="p14">
  <connection ref="t206" side="minus"/>
  <connection ref="t205" side="plus"/>
  <connection ref="t200" side="stem"/>
</point>
<point id="p15">
  <connection ref="p16" side="minus"/>
  <connection ref="p18" side="plus"/>
  <connection ref="t219" side="stem"/>
</point>
```

Listing 87: XML-Repräsentation des TEAMOD-Gleisnetzes Teil 4

```
<point id="p16">
  <connection ref="p15" side="minus"/>
  <connection ref="t218" side="plus"/>
  <connection ref="p17" side="stem"/>
</point>
<point id="p17">
  <connection ref="p18" side="minus"/>
  <connection ref="p20" side="plus"/>
  <connection ref="p16" side="stem"/>
</point>
<point id="p18">
  <connection ref="p17" side="minus"/>
  <connection ref="p15" side="plus"/>
  <connection ref="p19" side="stem"/>
</point>
<point id="p19">
  <connection ref="t110" side="minus"/>
  <connection ref="t109" side="plus"/>
  <connection ref="p18" side="stem"/>
</point>
<point id="p20">
  <connection ref="t107" side="minus"/>
  <connection ref="t108" side="plus"/>
  <connection ref="p17" side="stem"/>
</point>
<point id="p21">
  <connection ref="t103" side="minus"/>
  <connection ref="t104" side="plus"/>
  <connection ref="t215" side="stem"/>
</point>
<point id="p22">
  <connection ref="t106" side="minus"/>
  <connection ref="t105" side="plus"/>
  <connection ref="t216" side="stem"/>
</point>
<point id="p24">
  <connection ref="c23" side="minus"/>
  <connection ref="t215" side="plus"/>
  <connection ref="t213" side="stem"/>
</point>
<point id="p25">
  <connection ref="t112" side="minus"/>
  <connection ref="t111" side="plus"/>
  <connection ref="t220" side="stem"/>
</point>
<point id="p26">
  <connection ref="t114" side="minus"/>
  <connection ref="t113" side="plus"/>
  <connection ref="t217" side="stem"/>
</point>
<point id="p27">
  <connection ref="t101" side="minus"/>
  <connection ref="t100" side="plus"/>
  <connection ref="p28" side="stem"/>
</point>
```

Listing 88: XML-Repräsentation des TEAMOD-Gleisnetzes Teil 5

```
  <point id="p28">
    <connection ref="p27" side="minus"/>
    <connection ref="t102" side="plus"/>
    <connection ref="t207" side="stem"/>
  </point>
  <crossover id="c23">
    <connection ref="t216" side="up right"/>
    <connection ref="t217" side="up left"/>
    <connection ref="p24" side="down right"/>
    <connection ref="t214" side="down left"/>
  </crossover>
  <crossover id="c10">
    <connection ref="p11" side="up right"/>
    <connection ref="t204" side="up left"/>
    <connection ref="t209" side="down right"/>
    <connection ref="c9" side="down left"/>
  </crossover>
  <crossover id="c9">
    <connection ref="c10" side="up right"/>
    <connection ref="t205" side="up left"/>
    <connection ref="t210" side="down right"/>
    <connection ref="p8" side="down left"/>
  </crossover>
</network>
```

Listing 89: XML-Repräsentation des TEAMOD-Gleisnetzes Teil 6

```
1  ------------------------------------------------------------
2  -- This file was generated by the 'XML to CSP' Code Generator
3  -- Version 0.5.4
4  --
5  -- by Felix Brüning, University of Bremen 2021
6  ------------------------------------------------------------
7  include "channel.csp"
8  include "track_elem.csp"
9  include "crossing.csp"
10 include "point.csp"
11
12 -- Group elements:
13 -- CSP-ID 17 -> 't216'
14 -- CSP-ID 18 -> 't217'
15 -- CSP-ID 15 -> 't214'
16 -- CSP-ID 63 -> 'c23'
17 CROSSING_c23 = ( TRACK_ELEMENT(216, 24, 22) [| { rail.24 } |]
18       ( TRACK_ELEMENT(217, 61, 60) [| { rail.60 } |]
19       ( TRACK_ELEMENT(214, 28, 25) [| { rail.25 } |]
20       CROSSING(23, 25, 26, 24, 60) )))
21
22 -- Group elements:
23 -- CSP-ID 5 -> 't204'
24 -- CSP-ID 10 -> 't209'
25 -- CSP-ID 64 -> 'c10'
26 CROSSING_c10 = ( TRACK_ELEMENT(204, 73, 57) [| { rail.57 } |]
27       ( TRACK_ELEMENT(209, 49, 39) [| { rail.49 } |]
28       CROSSING(10, 57, 53, 49, 54) ))
29
30 -- Group elements:
31 -- CSP-ID 6 -> 't205'
32 -- CSP-ID 11 -> 't210'
33 -- CSP-ID 65 -> 'c9'
34 CROSSING_c9 = ( TRACK_ELEMENT(205, 74, 58) [| { rail.58 } |]
35       ( TRACK_ELEMENT(210, 50, 40) [| { rail.50 } |]
36       CROSSING(9, 58, 54, 50, 55) ))
37
38 -- Group elements:
39 -- CSP-ID 14 -> 't213'
40 -- CSP-ID 38 -> 'p1'
41 POINT_p1 = ( TRACK_ELEMENT(213, 29, 27) [| { rail.29 } |]
42       POINT(1, 29, 30, 32) )
43
44 -- Group elements:
45 -- CSP-ID 39 -> 'p2'
46 POINT_p2 = POINT(2, 31, 30, 28)
47
48 -- Group elements:
49 -- CSP-ID 13 -> 't212'
50 -- CSP-ID 40 -> 'p3'
51 POINT_p3 = ( TRACK_ELEMENT(212, 36, 34) [| { rail.34 } |]
52       POINT(3, 31, 33, 34) )
```

Listing 90: Generierter CSP$_M$-Code des TEAMOD-Gleisnetzes Teil 1

```
-- Group elements:
-- CSP-ID 22 -> 't221'
-- CSP-ID 41 -> 'p4'
POINT_p4 = ( TRACK_ELEMENT(221, 37, 35) [| { rail.35 } |]
      POINT(4, 35, 33, 32) )

-- Group elements:
-- CSP-ID 9 -> 't208'
-- CSP-ID 42 -> 'p5'
POINT_p5 = ( TRACK_ELEMENT(208, 48, 38) [| { rail.38 } |]
      POINT(5, 36, 38, 39) )

-- Group elements:
-- CSP-ID 12 -> 't211'
-- CSP-ID 43 -> 'p6'
POINT_p6 = ( TRACK_ELEMENT(211, 51, 41) [| { rail.41 } |]
      POINT(6, 37, 41, 40) )

-- Group elements:
-- CSP-ID 8 -> 't207'
-- CSP-ID 44 -> 'p7'
POINT_p7 = ( TRACK_ELEMENT(207, 47, 46) [| { rail.47 } |]
      POINT(7, 52, 47, 48) )

-- Group elements:
-- CSP-ID 7 -> 't206'
-- CSP-ID 45 -> 'p8'
POINT_p8 = ( TRACK_ELEMENT(206, 75, 59) [| { rail.59 } |]
      POINT(8, 51, 55, 59) )

-- Group elements:
-- CSP-ID 46 -> 'p11'
POINT_p11 = POINT(11, 56, 53, 52)

-- Group elements:
-- CSP-ID 3 -> 't202'
-- CSP-ID 4 -> 't203'
-- CSP-ID 47 -> 'p12'
POINT_p12 = ( TRACK_ELEMENT(202, 70, 69) [| { rail.70 } |]
      ( TRACK_ELEMENT(203, 72, 71) [| { rail.71 } |]
      POINT(12, 56, 70, 71) ))

-- Group elements:
-- CSP-ID 1 -> 't201'
-- CSP-ID 48 -> 'p13'
POINT_p13 = ( TRACK_ELEMENT(201, 76, 2) [| { rail.76 } |]
      POINT(13, 76, 72, 73) )

-- Group elements:
-- CSP-ID 2 -> 't200'
-- CSP-ID 49 -> 'p14'
POINT_p14 = ( TRACK_ELEMENT(200, 77, 1) [| { rail.77 } |]
      POINT(14, 77, 75, 74) )
```

Listing 91: Generierter CSP$_M$-Code des TEAMOD-Gleisnetzes Teil 2

```
-- Group elements:
-- CSP-ID 20 -> 't219'
-- CSP-ID 50 -> 'p15'
POINT_p15 = ( TRACK_ELEMENT(219, 2, 79) [| { rail.79 } |]
      POINT(15, 79, 5, 4) )

-- Group elements:
-- CSP-ID 19 -> 't218'
-- CSP-ID 51 -> 'p16'
POINT_p16 = ( TRACK_ELEMENT(218, 1, 78) [| { rail.78 } |]
      POINT(16, 3, 5, 78) )

-- Group elements:
-- CSP-ID 52 -> 'p17'
      POINT_p17 = POINT(17, 3, 6, 7)

-- Group elements:
-- CSP-ID 53 -> 'p18'
      POINT_p18 = POINT(18, 8, 6, 4)

-- Group elements:
-- CSP-ID 33 -> 't110'
-- CSP-ID 32 -> 't109'
-- CSP-ID 54 -> 'p19'
POINT_p19 = ( TRACK_ELEMENT(110, 12, 16) [| { rail.12 } |]
      ( TRACK_ELEMENT(109, 11, 15) [| { rail.11 } |]
      POINT(19, 8, 12, 11) ))

-- Group elements:
-- CSP-ID 30 -> 't107'
-- CSP-ID 31 -> 't108'
-- CSP-ID 55 -> 'p20'
POINT_p20 = ( TRACK_ELEMENT(107, 9, 13) [| { rail.9 } |]
      ( TRACK_ELEMENT(108, 10, 14) [| { rail.10 } |]
      POINT(20, 7, 9, 10) ))

-- Group elements:
-- CSP-ID 26 -> 't103'
-- CSP-ID 27 -> 't104'
-- CSP-ID 16 -> 't215'
-- CSP-ID 56 -> 'p21'
POINT_p21 = ( TRACK_ELEMENT(103, 13, 17) [| { rail.17 } |]
      ( TRACK_ELEMENT(104, 14, 18) [| { rail.18 } |]
      ( TRACK_ELEMENT(215, 23, 21) [| { rail.21 } |]
      POINT(21, 21, 17, 18) )))

-- Group elements:
-- CSP-ID 29 -> 't106'
-- CSP-ID 28 -> 't105'
-- CSP-ID 57 -> 'p22'
POINT_p22 = ( TRACK_ELEMENT(106, 16, 20) [| { rail.20 } |]
      ( TRACK_ELEMENT(105, 15, 19) [| { rail.19 } |]
      POINT(22, 22, 20, 19) ))
```

Listing 92: Generierter CSP_M-Code des TEAMOD-Gleisnetzes Teil 3

```
-- Group elements:
-- CSP-ID 58 -> 'p24'
POINT_p24 = POINT(24, 27, 26, 23)

-- Group elements:
-- CSP-ID 35 -> 't112'
-- CSP-ID 34 -> 't111'
-- CSP-ID 21 -> 't220'
-- CSP-ID 59 -> 'p25'
POINT_p25 = ( TRACK_ELEMENT(112, 67, 65) [| { rail.67 } |]
      ( TRACK_ELEMENT(111, 66, 64) [| { rail.66 } |]
      ( TRACK_ELEMENT(220, 69, 68) [| { rail.68 } |]
      POINT(25, 68, 67, 66) )))

-- Group elements:
-- CSP-ID 37 -> 't114'
-- CSP-ID 36 -> 't113'
-- CSP-ID 60 -> 'p26'
POINT_p26 = ( TRACK_ELEMENT(114, 65, 63) [| { rail.63 } |]
      ( TRACK_ELEMENT(113, 64, 62) [| { rail.62 } |]
      POINT(26, 61, 63, 62) ))

-- Group elements:
-- CSP-ID 24 -> 't101'
-- CSP-ID 23 -> 't100'
-- CSP-ID 61 -> 'p27'
POINT_p27 = ( TRACK_ELEMENT(101, 43, 81) [| { rail.43 } |]
      ( TRACK_ELEMENT(100, 42, 82) [| { rail.42 } |]
      POINT(27, 45, 43, 42) ))

-- Group elements:
-- CSP-ID 25 -> 't102'
-- CSP-ID 62 -> 'p28'
POINT_p28 = ( TRACK_ELEMENT(102, 44, 80) [| { rail.44 } |]
      POINT(28, 46, 45, 44) )

SYSTEM_27 = CROSSING_c23 [| { rail.28 } |] POINT_p2
SYSTEM_26 = POINT_p1 [| { rail.30 } |] SYSTEM_27
SYSTEM_25 = POINT_p3 [| { rail.31 } |] SYSTEM_26
SYSTEM_24 = POINT_p4 [| { rail.33, rail.32 } |] SYSTEM_25
SYSTEM_23 = POINT_p5 [| { rail.36 } |] SYSTEM_24
SYSTEM_22 = CROSSING_c10 [| { rail.39 } |] SYSTEM_23
SYSTEM_21 = CROSSING_c9 [| { rail.54 } |] SYSTEM_22
SYSTEM_20 = POINT_p6 [| { rail.40, rail.37 } |] SYSTEM_21
SYSTEM_19 = POINT_p7 [| { rail.48 } |] SYSTEM_20
SYSTEM_18 = POINT_p8 [| { rail.55, rail.51 } |] SYSTEM_19
SYSTEM_17 = POINT_p11 [| { rail.53, rail.52 } |] SYSTEM_18
SYSTEM_16 = POINT_p12 [| { rail.56 } |] SYSTEM_17
SYSTEM_15 = POINT_p13 [| { rail.72, rail.73 } |] SYSTEM_16
SYSTEM_14 = POINT_p14 [| { rail.75, rail.74 } |] SYSTEM_15
SYSTEM_13 = POINT_p15 [| { rail.2 } |] SYSTEM_14
SYSTEM_12 = POINT_p16 [| { rail.1, rail.5 } |] SYSTEM_13
SYSTEM_11 = POINT_p17 [| { rail.3 } |] SYSTEM_12
SYSTEM_10 = POINT_p18 [| { rail.6, rail.4 } |] SYSTEM_11
```

Listing 93: Generierter CSP_M-Code des TEAMOD-Gleisnetzes Teil 4

```
SYSTEM_9 = POINT_p19 [| { rail.8 } |] SYSTEM_10
SYSTEM_8 = POINT_p20 [| { rail.7 } |] SYSTEM_9
SYSTEM_7 = POINT_p21 [| { rail.13, rail.14 } |] SYSTEM_8
SYSTEM_6 = POINT_p22 [| { rail.16, rail.15, rail.22 } |] SYSTEM_7
SYSTEM_5 = POINT_p24 [| { rail.26, rail.23, rail.27 } |] SYSTEM_6
SYSTEM_4 = POINT_p25 [| { rail.69 } |] SYSTEM_5
SYSTEM_3 = POINT_p26 [| { rail.65, rail.64, rail.61 } |] SYSTEM_4
SYSTEM_2 = POINT_p28 [| { rail.46 } |] SYSTEM_3
SYSTEM_1 = POINT_p27 [| { rail.45 } |] SYSTEM_2

RAILWAY_NETWORK = SYSTEM_1
```

Listing 94: Generierter CSP_M-Code des TEAMOD-Gleisnetzes Teil 5

```
assert GRP_11 [F= POINT_p13
assert GRP_15 [F= POINT_p14
assert GRP_08 [F= POINT_p15
assert GRP_05 [F= POINT_p16
assert GRP_20 [F= POINT_p25
assert GRP_23 [F= POINT_p20
assert GRP_25 [F= POINT_p19
assert GRP_26 [F= POINT_p22
assert GRP_13 [F= POINT_p7

assert POINT_p13 [F= GRP_11
assert POINT_p14 [F= GRP_15
assert POINT_p15 [F= GRP_08
assert POINT_p16 [F= GRP_05
assert POINT_p25 [F= GRP_20
assert POINT_p20 [F= GRP_23
assert POINT_p19 [F= GRP_25
assert POINT_p22 [F= GRP_26
assert POINT_p7  [F= GRP_13
```

Listing 95: Vollständiger Test der Sub-Prozesse von dem Referenzmodell und dem generierten Modell in eins-zu-eins Korrespondenz

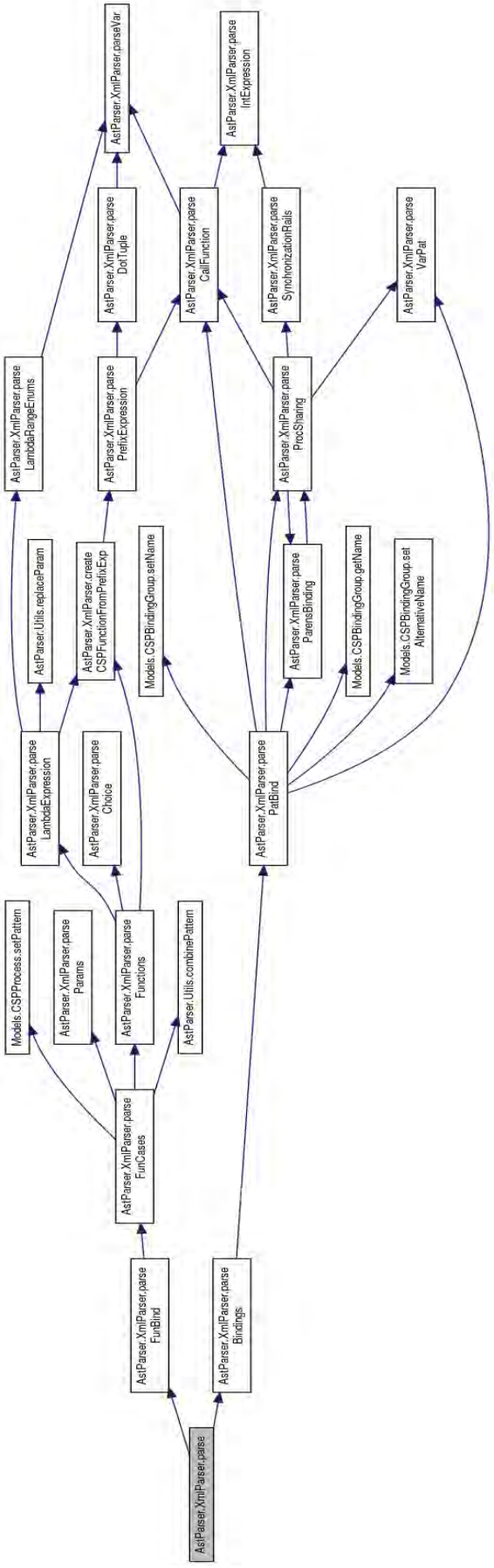

Abbildung 20: Parsieren des Syntaxbaums der XML-Datei

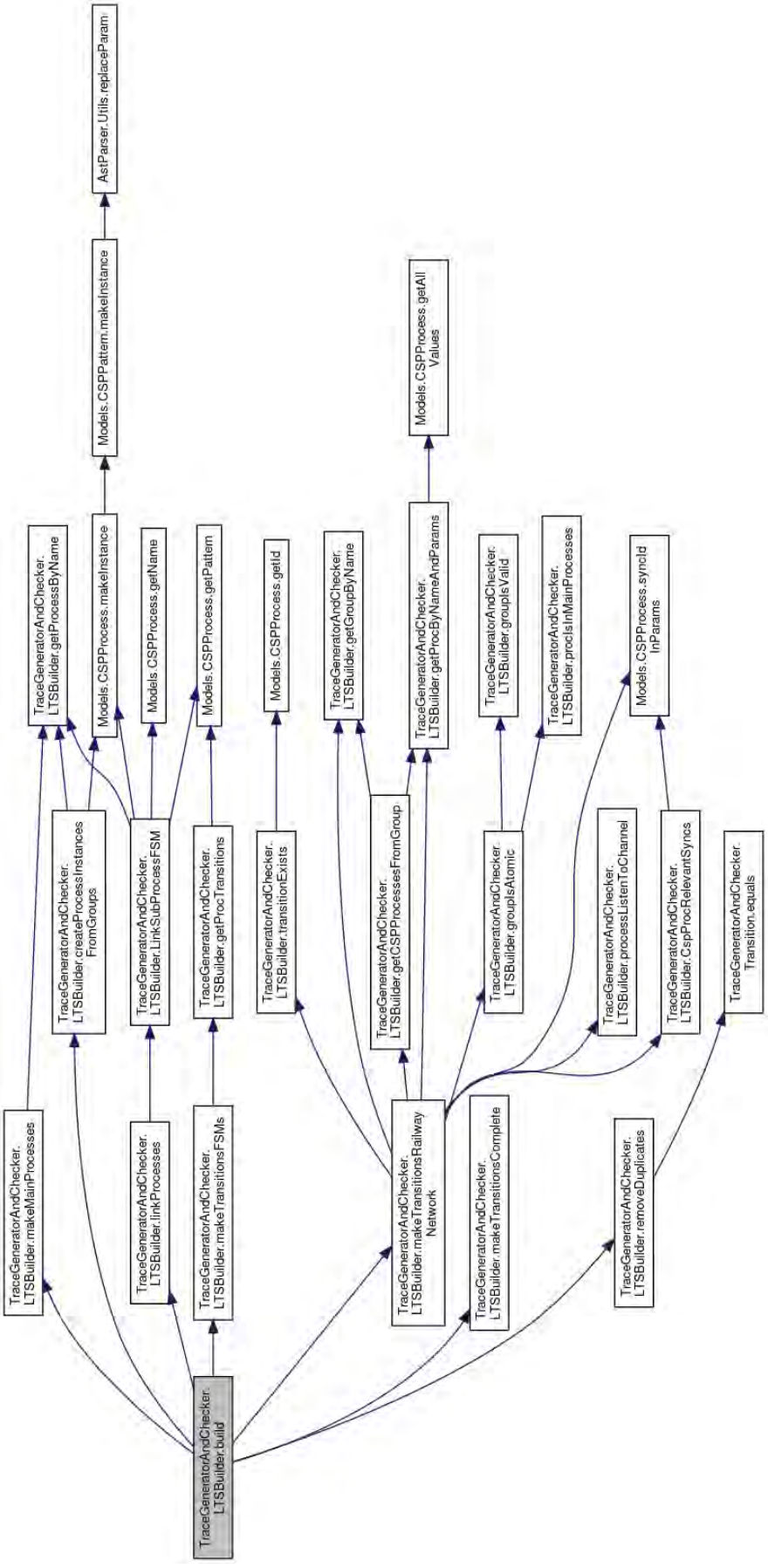

Abbildung 21: Aufrufgraph zum Erstellen des LTS

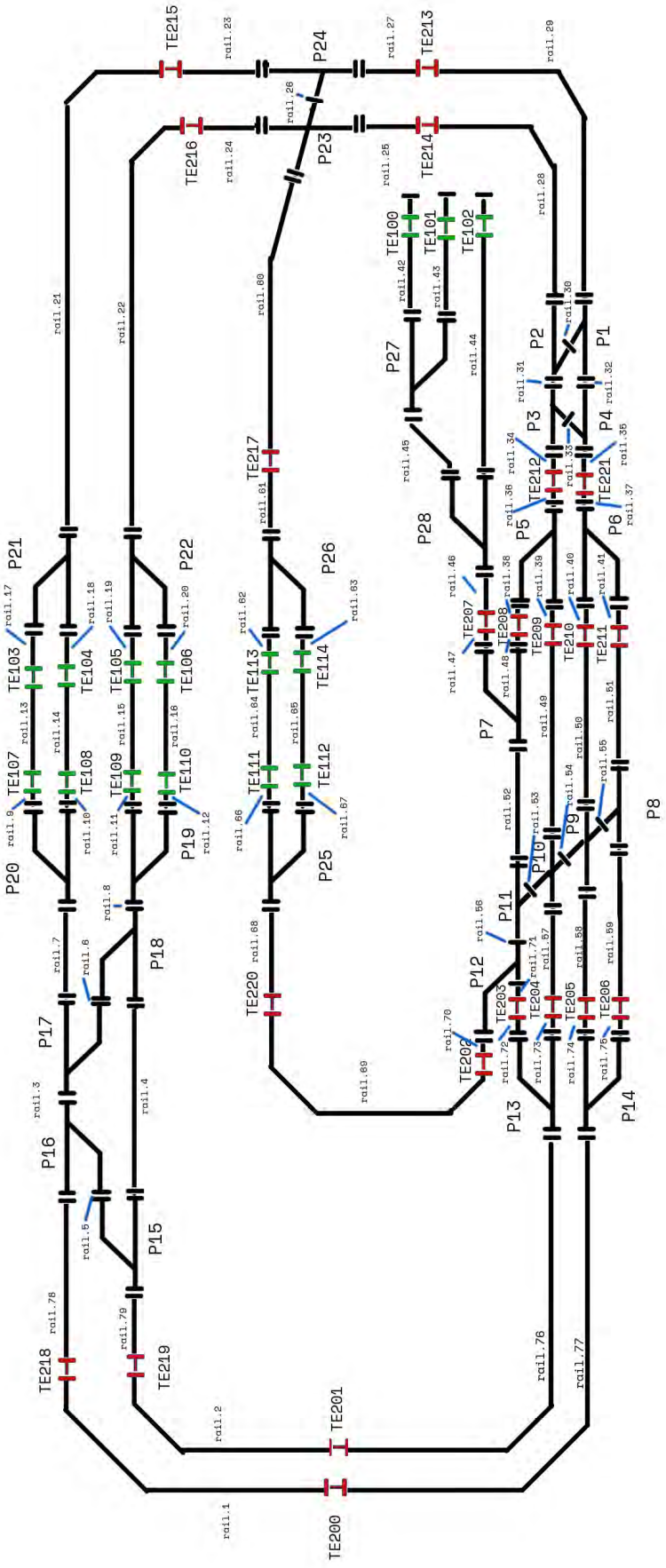

Abbildung 22: TEAMOD-Gleisnetz

Inhalt des Speichermediums

```
.
|-- Software
|   |-- 3d_party
|   |   `-- cspmf
|   |       |-- CSPM-Frontend
|   |       |-- CSPM-ToProlog
|   |       `-- CSPM-cspm-frontend
|   |-- CSP_Code_Generator
|   |   `-- c
|   |       |-- bin
|   |       |-- codegen
|   |       |-- doc
|   |       |-- models
|   |       `-- xmlImport
|   `-- gleisnetz_trace_gen_interpreter
|       |-- doc
|       |   |-- html
|       |   `-- man
|       |-- examples
|       |   |-- teamod_gleisnetz
|       |   `-- kleines_gleisnetz
|       |-- gradle
|       `-- src
|           |-- main
|           |   |-- java
|           |   |   |-- AstParser
|           |   |   |-- Models
|           |   |   `-- TraceGeneratorAndChecker
|           |   `-- resources
|           |       `-- csp
|           `-- test
|               |-- java
|               `-- resources
|-- Thesis
`-- Verifikation
  |-- AequivalenzTests_Beweise
  |   |-- kleines_gleisnetz
  |   `-- teamod_gleisnetz
  |-- PropertyCatalog
  |-- Systemtests_bachelorthesis
  |   |-- KonfliktTest
  |   |-- NichtKonfliktTest
  |   `-- Routen
  `-- Tests_TraceGenerator_Interpreter
    |-- teamod_gleisnetz
    `-- kleines_gleisnetz

70 directories
```

Listing 96: Inhalt des Speichermediums

Eidesstattliche Erklärung

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

Ort, Datum *Unterschrift*