KAPITEL 10

OHA-Beispiel — Zugleitbetrieb nach FV-NE

Bernd Sieker

10.1 Einführung

Hier soll die Durchführung einer Ontological Hazard Analysis exemplarisch für den Fall des Zugleitbetriebs nach der Fahrdienstvorschrift für nicht bundeseigene Eisenbahnen [71] gezeigt werden.

Es handelt sich hierbei um ein Bahnbetriebsverfahren für eingleisige Nebenstrecken, das ohne Signalisierung auskommt, und für das in der einfachsten Form mit einem Zugleiter für mehrere Stationen und ohne technische Sicherungssysteme arbeitet.

Für die Durchführung der Ontological Hazard Analysis werden Techniken aus den Bereichen der Informatik und der Logik gebraucht.

10.1.1 Sortenlogik (*Many-Sorted Logic*)

Für die formal-logische Beschreibung der Bahnbetriebsverfahren und ihrer Objekte werden unterschiedliche Arten von Axiomen bzw. Postulaten verwendet:

Systemlogik-Axiome werden nicht explizit notiert, sondern ergeben sich aus der Definition der Sorten (*sorts*) von Objekten, die auftreten.

Bedeutungspostulate (*Meaning Postulates*) [5] beschreiben die Bedeutung von Prädikaten (beispielsweise bei einem Verfeinerungsschritt der OHA) mit bereits

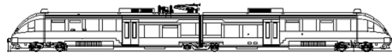
bestehenden Elementen der Systembeschreibung.

Domain-Axiome sind Formeln, die den Betrieb beschreiben, beispielsweise, unter welchen Bedingungen die Fahranfrage im Zugleitbetrieb gestellt wird, und wie darauf reagiert wird.

Sicherheitspostulate (*Safety Postulates*) sind ein Sonderfall der Domain-Axiome, deren Negation einen Hazard darstellt.

Die Ontological Hazard Analysis erzeugt im Laufe ihrer Durchführung auf jeder Ebene eine Systemdefinition. Darin werden alle Sorten und Relationen dieses Systems beschrieben sowie eine vollständige Hazard-Analyse durchgeführt.

10.2 Ebene 0 („UrSpec“)



10.2.1 Sorten (*sorts*)

Auf der ersten Ebene betrachten wir sehr verallgemeinerte Objekte, siehe Tabelle 10.1. Die erste Ebene soll möglichst einfach sein, so dass sich Sicherheitsanforderungen formulieren lassen, die *offensichtlich* richtig sind.

Eisenbahnfahrzeug	Eigenständig fahrende Einheit; Lokomotive, Zug oder Rangiereinheit
Streckenabschnitt	Allgemeine Darstellung eines Streckenabschnitts

Tabelle 10.1: Sorten auf Ebene 0 („Level 0“)

10.2.2 Relationen

Binäre Relationen

Die Tabelle in Tabelle 10.2 zeigt die Relationen der ersten Ebene, die jeweils zwei Objekte verbinden.

Eigenschaft	(informelle) Beschreibung
EISENBAHNFAHRZEUG, STRECKENABSCHNITT	
$\mathrm{inA}(F,S)$	Fahrzeug F befindet sich im Streckenabschnitt S
$\mathrm{ZV}(F,S)$	Fahrzeug F darf den Streckenabschnitt S unter zentraler Verantwortung belegen[a]
$\mathrm{LV}(F,S)$	Fahrzeug F darf den Streckenabschnitt S unter lokaler Verantwortung belegen[b]

a Dies ist der normale, planmäßige Betrieb.
b Dies sind besondere Fahrten, die nicht dem normalen Fahrplanbetrieb entsprechen, z. B. Sperrfahrten.

Tabelle 10.2: Binäre Relationen auf Ebene 0

10.2.3 Sicherheitsaxiome

Zur Erlangung der Axiome für den sicheren Betrieb und Beweis der Vollständigkeit betrachtet man die drei Prädikate $\mathrm{LV}(F1,S) = \mathrm{LV1}$, $\mathrm{inA}(F1,S) = \mathrm{in1}$ und $\mathrm{ZV}(F1, S) = \mathrm{ZV1}$ für einen Zug $F1$, entsprechend LV2, $\mathrm{in2}$ und ZV2 für einen zweiten Zug $F2$.

10.2.4 Ein Zug

Für einen festen Streckenabschnitt S ergeben sich so drei atomare Aussagen über einen bestimmten Zug $F1$, nämlich LV1, $\mathrm{in1}$, ZV1. Es gibt genau $2^3 = 8$ wahrheitsfunktiononale Kombinationen von drei atomaren Aussagen, und damit $2^{2^3} = 2^8 = 256$ verschiedene Wahrheitsfunktionen $\mathrm{TF}_1 \cdots \mathrm{TF}_{256}$. Alle möglichen Axiome haben die Form $\mathrm{TF}_n =$ wahr.

Durch einfache logische Umformungen und Folgerungen ergeben sich für einen Zug die folgenden drei Sicherheits-Axiome:

$$\mathrm{ZV1} \Rightarrow \neg\,\mathrm{LV1} \tag{S0.1}$$
$$\neg\,\mathrm{LV1} \wedge \mathrm{in1} \Rightarrow \mathrm{ZV1} \tag{S0.2}$$
$$\mathrm{in1} \wedge \mathrm{ZV1} \Rightarrow \neg\,\mathrm{LV1} \tag{S0.3}$$

10.2.5 Zwei Züge

Für die Betrachtung von 2 Zügen ergeben sich sechs atomare Aussagen: $\mathrm{LV1}$, $\mathrm{LV2}$, $\mathrm{in1}$, $\mathrm{in2}$, $\mathrm{ZV1}$, $\mathrm{ZV2}$. Die systematische Vorgehensweise ist wie bei einem einzelnen Zug.

Es ist durch vollständige Aufzählung garantiert, dass alle Sicherheits-Axiome für diese Verfeinerungsebene betrachtet wurden. Insgesamt sind diese in Tabelle 10.3 zusammengefasst.

$$\mathrm{ZV1} \Rightarrow \neg\,\mathrm{LV1} \tag{S0.1}$$
$$\neg\,\mathrm{LV1} \wedge \mathrm{in1} \Rightarrow \mathrm{ZV1} \tag{S0.2}$$
$$\mathrm{in1} \wedge \mathrm{ZV1} \Rightarrow \neg\,\mathrm{LV1} \tag{S0.3}$$
$$(F1 \neq F2) \Rightarrow (\mathrm{LV1} \Rightarrow \neg\,\mathrm{ZV2}) \tag{S0.4}$$
$$(F1 \neq F2) \Rightarrow (\mathrm{in1} \Rightarrow \neg\,\mathrm{ZV2}) \tag{S0.5}$$
$$(F1 \neq F2) \Rightarrow (\mathrm{ZV1} \Rightarrow \neg\,\mathrm{ZV2}) \tag{S0.6}$$

Tabelle 10.3: Sicherheitsaxiome der Ebene 0

10.2.6 Verfeinerung (*Refinement*)

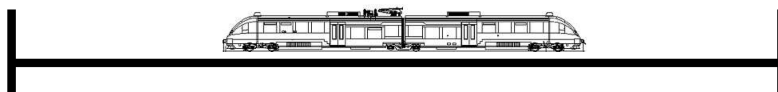
Die Ontological Hazard Analysis schlägt als Methode zur Festlegung, welche Objekte wie verfeinert werden sollen, HAZOP[25] vor. Damit sollen Abweichungen vom geplanten Betrieb festgestellt werden, und bei denjenigen Abweichungen, die sich

nicht in der aktuellen Ontologie ausdrücken lassen, diese um die fehlenden Objekte und Relationen erweitert werden.

HAZOP stellt jedoch nicht das einzige Kriterium dar. Da sich die Spezifikation an der Fahrdienstvorschrift für nicht-bundeseigene Bahnen (*FV-NE*, [71]) orientieren soll, kann es auch nützlich sein, gerade in den ersten, einfachen Stufen, sich gezielt der Darstellung in der Vorschrift zu nähern.

In diesem Fall bietet sich eine Konkretisierung des generischen *Streckenabschnitt*s an, der entweder eine Strecke zwischen zwei *Zuglaufmeldestellen* sein kann, oder ein Gleis im Bahnhof. Die Verfahren im Zugleitbetrieb orientieren sich an Zuglaufmeldestellen, so dass eine Beschreibung des Verfahrens erst erfolgen kann, wenn diese Teil der Sprache sind.

10.3 Ebene 1 — erste Verfeinerung (*Refinement*)



10.3.1 Sorten

Als Verfeinerung von Ebene 0 zu Ebene 1 (*Verfeinerungsschritt 1*) wird der *Streckenabschnitt* näher bestimmt; hierzu ist die Einführung neuer Objekte nötig: *Zuglaufmeldestelle* und *Gleis im Bahnhof*. Tabelle 10.4 zeigt eine erweiterte Liste der Sorten.

Eisenbahnfahrzeug	Eigenständig fahrende Einheit; Lokomotive, Zug oder Rangiereinheit
Streckenabschnitt	Blockabschnitt
Zuglaufmeldestelle	Begrenzung eines Streckenabschnitts
Gleis	Gleis im Bahnhof, also nicht auf der freien Strecke.

Tabelle 10.4: Sorten auf Ebene 1 („Level 1“)

Tabelle 10.5 zeigt die binären, und Tabelle 10.6 die ternären Relationen der Ebene 1, neu sind hier entsprechende Relationen zwischen Fahrzeugen und Zuglaufmeldestellen bzw. Gleisen.

Eigenschaft	(informelle) Beschreibung
EISENBAHNFAHRZEUG, STRECKENABSCHNITT	
inA(F,S)	Fahrzeug F befindet sich im Streckenabschnitt S
ZV(F,S)	Fahrzeug F darf den Streckenabschnitt S unter zentraler Verantwortung belegen
LV(F,S)	Fahrzeug F darf den Streckenabschnitt S unter lokaler Verantwortung belegen
EISENBAHNFAHRZEUG, ZUGLAUFMELDESTELLE	
inZ(F,A)	Fahrzeug F befindet sich in der Zuglaufmeldestelle A
EISENBAHNFAHRZEUG, GLEIS	
inG(F,G)	Fahrzeug F befindet sich auf Gleis G
ZG(F,G)	Fahrzeug F darf Gleis G unter zentraler Verantwortung belegen

Tabelle 10.5: Binäre Relationen auf Ebene 1

Bedeutungspostulate

An dieser Stelle muss nur gezeigt werden, dass sich die Elemente der vorhergehenden Stufe mit einem Bedeutungspostulat (*Meaning Postulate* mit den Elementen dieser Stufe beschreiben lassen.

10.3.2 Sicherheitsaxiome

Durch Anwenden der Bedeutungspostulate auf die Sicherheitsaxiome aus Ebene 0 ergeben sich in Abbildung 10.1 die neuen Axiome.

Eigenschaft	(informelle) Beschreibung
STRECKENABSCHNITT, 2× ZUGLAUFMELDESTELLE	
$\mathrm{begr}(S,A,B)$	Zwei Zuglaufmeldestellen A und B begrenzen den Streckenabschnitt S
EISENBAHNFAHRZEUG, 2× ZUGLAUFMELDESTELLE	
$\mathrm{zw}(F,A,B)$	Eisenbahnfahrzeug befindet sich zwischen den Zuglaufmeldestellen
$\mathrm{ZZ}(F,A,B)$	Fahrzeug F darf den Streckenabschnitt zwischen A und B unter zentraler Verantwortung belegen

Tabelle 10.6: Ternäre Relationen auf Ebene 1

$$\mathrm{ZZ1} \Rightarrow \neg\,\mathrm{LV1} \tag{S1.1}$$

$$\mathrm{ZG1} \Rightarrow \neg\,\mathrm{LV1} \tag{S1.2}$$

$$\neg\,\mathrm{LV1} \wedge \mathrm{ZW1} \Rightarrow \mathrm{ZZ1} \tag{S1.3}$$

$$\neg\,\mathrm{LV1} \wedge \mathrm{G1} \Rightarrow \mathrm{ZG1} \tag{S1.4}$$

$$\mathrm{ZW1} \wedge \mathrm{ZZ1} \Rightarrow \neg\,\mathrm{LV1} \tag{S1.5}$$

$$\mathrm{G1} \wedge \mathrm{ZG1} \Rightarrow \neg\,\mathrm{LV1} \tag{S1.6}$$

$$(F1 \neq F2) \Rightarrow (\mathrm{LV1} \Rightarrow \neg\,\mathrm{ZZ2}) \tag{S1.7}$$

$$(F1 \neq F2) \Rightarrow (\mathrm{LV1} \Rightarrow \neg\,\mathrm{ZG2}) \tag{S1.8}$$

$$(F1 \neq F2) \Rightarrow (\mathrm{ZW1} \Rightarrow \neg\,\mathrm{ZZ2}) \tag{S1.9}$$

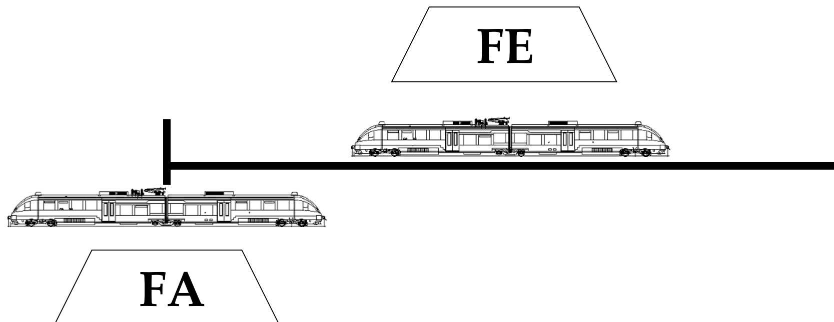
$$(F1 \neq F2) \Rightarrow (\mathrm{G1} \Rightarrow \neg\,\mathrm{ZG2}) \tag{S1.10}$$

$$(F1 \neq F2) \Rightarrow (\mathrm{ZZ1} \Rightarrow \neg\,\mathrm{ZZ2}) \tag{S1.11}$$

$$(F1 \neq F2) \Rightarrow (\mathrm{ZG1} \Rightarrow \neg\,\mathrm{ZG2}) \tag{S1.12}$$

Abbildung 10.1: Zusätzliche Sicherheitsaxiome auf Ebene 1

10.4 Ebene 2



10.4.1 Sorten

Die Verfeinerung von Ebene 1 zu Ebene 2 umfasst keine neuen Sorten, sondern nur neue Relationen. Die Sortentablle ist daher identisch zu der von Ebene 1, siehe Tabelle 10.7

Eisenbahnfahrzeug	Eigenständig fahrende Einheit; Lokomotive, Zug oder Rangiereinheit
Streckenabschnitt	Blockabschnitt
Zuglaufmeldestelle	Begrenzung eines Streckenabschnitts
Gleis	Gleise im Bahnhof, also nicht auf der freien Strecke.

Tabelle 10.7: Sorten auf Ebene 2

10.4.2 Relationen

Alle diese Objekte haben Eigenschaften, die sich in Form von unären Relationen ausdrücken lassen.

Auch in dieser Ebene haben die Objekte außer ihren Sorten-Prädikaten keine unären Relationen.

Hier gibt es die in Tablle 10.8 aufgeführten binäre Relationen.

Eigenschaft	(informelle) Beschreibung
EISENBAHNFAHRZEUG, STRECKENABSCHNITT	
inA(F,S)	Fahrzeug F befindet sich im Streckenabschnitt S
ZV(F,S)	Fahrzeug F darf den Streckenabschnitt S unter zentraler Verantwortung belegen
LV(F,S)	Fahrzeug F darf den Streckenabschnitt S unter lokaler Verantwortung belegen
EISENBAHNFAHRZEUG, ZUGLAUFMELDESTELLE	
inZ(F,A)	Fahrzeug F befindet sich in der Zuglaufmeldestelle A
FAHRZEUG, ZUGLAUFMELDESTELLE	
Next(F,A)	Diese Funktion bezeichnet die nächste fahrplanmäßige Zuglaufmeldestelle für Fahrzeug F nach der Zuglaufmeldestelle A

Tabelle 10.8: Binäre Relationen auf Ebene 2

Es gibt einige Relationen zwischen drei Objekten, siehe Tabelle 10.9.

10.4.3 Beschreibung einer Zugfahrt im Zugleitbetrieb

Diese vereinfachte Beschreibung einer Zugfahrt im Zugleitbetrieb betrachtet eine Fahrt über n Abschnitte, S_1 bis S_n, beginnend auf Zuglaufmeldestelle A_0, und endend auf Zuglaufmeldestelle A_n. Siehe Abbildung 10.2.

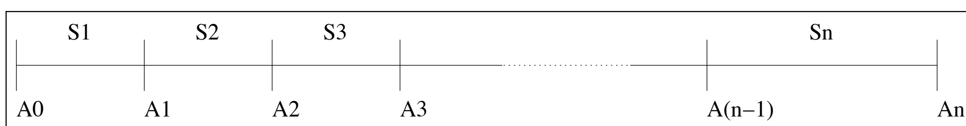


Abbildung 10.2: Schematische Zugleitstrecke

Eigenschaft	(informelle) Beschreibung
STRECKENABSCHNITT, 2× ZUGLAUFMELDESTELLE	
$\mathrm{begr}(S,A,B)$	Zwei Zuglaufmeldestellen A und B begrenzen den Streckenabschnitt S
EISENBAHNFAHRZEUG, 2× ZUGLAUFMELDESTELLE	
$\mathrm{zw}(F,A,B)$	Eisenbahnfahrzeug befindet sich zwischen den Zuglaufmeldestellen
$\mathrm{FA}(F,A,B)$	Fahrzeug F in A hat Fahranfrage bis Zuglaufmeldestelle B gestellt
$\mathrm{FE}(F,A,B)$	Fahrzeug F in A hat Fahrerlaubnis bis Zuglaufmeldestelle B erteilt bekommen
$\mathrm{AFE}(F,A,B)$	Fahrerlaubnis für Fahrzeug F in A bis Zuglaufmeldestelle B abgelehnt [a]
$\mathrm{KH}(F,A,B)$	Es sind keine Hindernisse bekannt, die der Fahrt des Fahrzeuges F von Zuglaufmeldestelle A bis B entgegensprechen.

a Dies ist nicht dasselbe wie $\neg FE(F,B)$, es existiert auch der Fall, dass Fahrerlaubnis weder erteilt noch abgelehnt wurde.

Tabelle 10.9: Ternäre Relationen auf Ebene 2

Als Zustandsautomat (Predicate Action Diagram)

Eine Zugfahrt auf diesem Level kann als Zustandsmaschine dargestellt werden, siehe Abbildung 10.3.

Die Zustände sind in Tabelle 10.10 exakt beschrieben, Tabelle 10.11 zeigt eine informelle Beschreibung in natürlicher Sprache:

Diese Beschreibungen (bzw. deren formale Entsprechung in Abbildung 10.10) sind die Axiome zur Durchführung des normalen Betriebes bei einer Fahrt von einer Zuglaufmeldestelle zur nächsten.

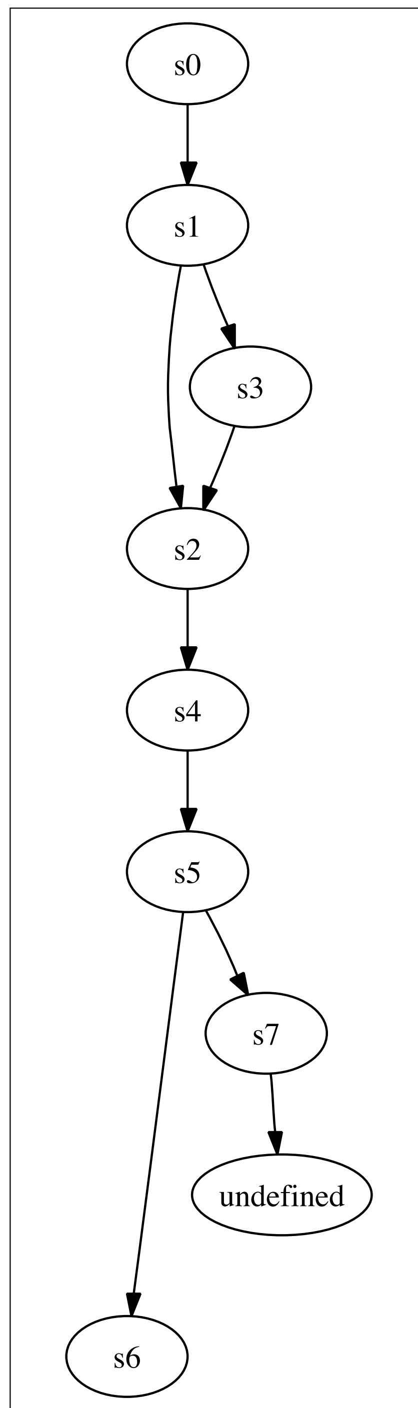


Abbildung 10.3: Zustandsmaschine

$$
\begin{aligned}
s0 &= \mathrm{inZ}(F,A) \\[1em]
s1 &= \wedge\, \mathrm{inZ}(F,A) \\
&\quad \wedge\, \mathrm{FA}(F,A,Next(F,A)) \\
&\quad \wedge \neg\, \mathrm{FE}(F,A,Next(F,A)) \\[1em]
s2 &= \wedge\, \mathrm{inZ}(F,A) \\
&\quad \wedge\, \mathrm{FA}(F,A,Next(F,A)) \\
&\quad \wedge\, \mathrm{KH}(F,A,Next(F,A)) \\[1em]
s3 &= \wedge\, \mathrm{inZ}(F,A) \\
&\quad \wedge\, \mathrm{FA}(F,A,Next(F,A)) \\
&\quad \wedge \neg\, \mathrm{FE}(F,A,Next(F,A)) \\
&\quad \wedge \neg\, \mathrm{KH}(F,A,Next(F,A)) \\
&\quad \wedge\, \mathrm{AFE}(F,A,Next(F,A)) \\[1em]
s4 &= \wedge\, \mathrm{inZ}(F,A) \\
&\quad \wedge\, \mathrm{FA}(F,A,Next(F,A)) \\
&\quad \wedge\, \mathrm{FE}(F,A,Next(F,A)) \\
&\quad \wedge\, \mathrm{KH}(F,A,Next(F,A) \\[1em]
s5 &= \wedge\, \mathrm{zw}(F,A,Next(F,A)) \\
&\quad \wedge\, \mathrm{FE}(F,A,Next(F,A)) \\
&\quad \wedge\, \mathrm{KH}(F,A,Next(F,A)) \\
&\quad \wedge \neg\, \mathrm{LV}(F) \\[1em]
s6 &= \mathrm{inZ}(F,A) \\
&= s0 \\[1em]
s7 &= \wedge\, \mathrm{zw}(F,A,Next(F,A)) \\
&\quad \wedge\, \mathrm{FE}(F,A,Next(F,A)) \\
&\quad \wedge \neg\, \mathrm{KH}(F,A,Next(F,A)) \\
&\quad \wedge \neg\, \mathrm{LV}(F)
\end{aligned}
$$

Tabelle 10.10: Zustände des Automaten in Ebene 2

s0	Fahrzeug ist in Zuglaufmeldestelle A
s1	wie s0, zusätzlich wurde die Fahranfrage bis zur nächsten Zuglaufmeldestelle gestellt, aber noch keine Fahrerlaubnis erteilt.
s2	Fahrzeug ist in Zuglaufmeldestelle A, die Fahranfrage wurde gestellt, und der Fahrt bis zur nächsten Zuglaufmeldestelle stehen keine Hindernisse entgegen.
s3	Fahrzeug ist in Zuglaufmeldestelle A, die Fahranfrage wurde gestellt, es wurde keine Fahrerlaubnis erteilt, und der Fahrt stehen Hindernisse entgegen. Die Fahrerlaubnis wurde abgelehnt.
s4	Fahrzeug ist in Zuglaufmeldestelle A, Es wurde Fahrerlaubnis erteilt und der Fahrt steht kein Hindernis entgegen.
s5	Fahrzeug ist im Streckenabschnitt zwischen den Zuglaufmeldestellen A und der nächsten. Die Fahrerlaubnis wurde erteilt, es stehen der Fahrt keine Hindernisse entgegen. Das Fahrzeug fährt nicht nach Sichtfahrtregeln.
s6	wie s0
s7	Das Fahrzeug ist mir Fahrerlaubnis in den Streckenabschnitt eingefahren, und unterwegs wurde ein Hindernis festgestellt. Dieser Zustand ist in der FV-NE so nicht vorgesehen.

Tabelle 10.11: Beschreibung der Zustände in natürlicher Sprache

Bedeutungspostulate

An dieser Stelle muss nur gezeigt werden, dass sich die Elemente einer vorhergehenden Stufe mit einem Bedeutungspostulat (*Meaning Postulate* mit den Elementen dieser Stufe beschreiben lassen.

Bedeutungspostulate

Die Bedeutungspostulate dieser Ebene sind sehr wenige und einfach:

$$\mathrm{ZV}(F,A,B) \Rightarrow \mathrm{KH}(F,A,B) \tag{M2.1}$$

$$\mathrm{ZV}(F,A,B) \Rightarrow \mathrm{FE}(F,A,B) \tag{M2.2}$$

$$\mathrm{inZ}(F,A) \Rightarrow \neg\, \mathrm{LV}\,1 \tag{M2.3}$$

Durch das Postulat (M2.3) ist gewährleistet, dass $\neg LV1$ eine Invariante des gesamten Zustandsautomaten ist.

10.4.4 Sicherheitsaxiome

Da die Sicherheitsaxiome auf Ebene 0 durch vollständige Aufzählung als ausreichend festgestellt wurden, müssen hier nur Sicherheitsaxiome definiert werden, die es erlauben, zu beweisen, dass Ebene 2 eine Verfeinerung der Ebene 0 darstellt. Der Punkt der möglichst einfachen Beschreibung auf Ebene 0 ist ja gerade, dass diese Art der Vollständigkeitsanalyse möglich ist, dadurch dass diese Ebene so einfach und klein ist.

Folgende Abkürzungen dienen der besseren Übersichtlichkeit in den folgenden Formeln:

$$\begin{aligned}
\mathrm{KH1} &= \mathrm{KH}(F1,A,Next(F1,A)) \\
\mathrm{FE1} &= \mathrm{FE}(F1,A,Next(F1,A)) \\
\mathrm{FE2} &= \mathrm{FE}(F2,A,Next(F1,A)) \\
\mathrm{FE2Rev} &= \mathrm{FE}(F2,Next(F1,A),A) \\
\mathrm{ZV1} &= \mathrm{ZV}(F1,A,Next(F1,A))
\end{aligned}$$

Die Sicherheitsaxiome auf Ebene 2 sind in Tabelle 10.4 dargestellt.

10.4.5 Beweis der Verfeinerung

Mit Hilfe der Bedeutungs-Postulate (M2.1), (M2.2), (M2.3), der Sicherheitsaxiomen (S2.1), (S2.2) und der Definition des Zustandsautomaten lassen sich die Sicherheitsaxiome aus Ebene 0 beweisen, wie folgt:

$$\text{KH1} \wedge \text{FE1} \Rightarrow \text{ZV1} \tag{S2.1}$$

D. h. wenn keine Hindernisse festgestellt wurden, und Fahrerlaubnis erteilt wurde, dann ist der Fahrweg tatsächlich frei.

$$(F1 \neq F2) \Rightarrow \neg(\text{FE1} \wedge (\text{FE2} \vee \text{FE2Rev})) \tag{S2.2}$$

D. h. wenn für einen von zwei verschiedenen Zügen die Fahrerlaubnis erteilt wurde, dann wurde für den zweiten Zug weder die Fahrerlaubnis in dieselbe Richtung, noch in die Gegenrichtung erteilt.

$$\begin{aligned} \Box \vee & (s0 \wedge s1') \\ \vee & (s1 \wedge s2') \\ \vee & (s1 \wedge s3') \\ \vee & (s3 \wedge s2') \\ \vee & (s2 \wedge s4') \\ \vee & (s4 \wedge s5') \\ \vee & (s5 \wedge s6') \\ \vee & (s5 \wedge s7') \\ \vee & (s7 \wedge undefined') \end{aligned} \tag{S2.3}$$

D. h. alles Aktionen, die stattfinden (□: immer (*always*)), sind Transitionen im Zustandsautomaten. Anders gesagt, das Predicate-Action-Diagramm beschreibt das Verfahren *vollständig*), es geschieht nichts, was nicht als Transition im Diagramm vorkommt.
Dies sieht aus wie eine Zuverlässigkeitsanforderung, ist jedoch auch ein Sicherheitsaxiom, da die Analyse des Verfahrens darauf beruht, dass nur genau diese Zustände und Transitionen auftreten.

Abbildung 10.4: Sicherheitsaxiome auf Ebene 2

[Beweis von (S0.1)] ZV1 impliziert KH1 (nach (M2.1)), KH1 impliziert, dass sich F1 im Zustand s2, s4 oder s5 befindet (nach Definition des Zustandsautomaten). In

diesen Zuständen gilt $\neg\,\mathrm{LV1}$ (nach Definition des Zustandsautomaten).

[Beweis von (S0.2)] $(\neg\,\mathrm{LV1} \wedge \mathrm{in1})$ impliziert, dass sich F1 im Zustand s5 befindet (nach Definition des Zustandsautomaten). Zustand s5 impliziert $(\mathrm{KH1} \wedge \mathrm{FE1})$ (nach Definition des Zustandsautomaten). $(\mathrm{KH1} \wedge \mathrm{FE1})$ impliziert $\mathrm{ZV1}$ (nach (S2.1)).

Die weitere Beweise sind ähnlich einfach.

10.4.6 Hazards

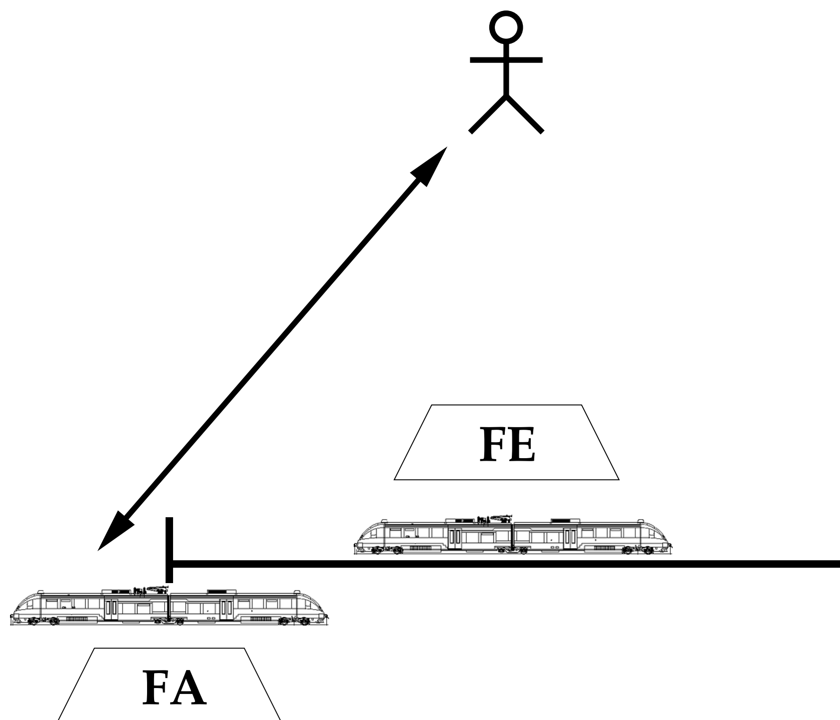
Die Hazards auf dieser Ebene sind die Möglichkeiten, dass eines der Sicherheitsaxiome (S2.1) oder (S2.2) verletzt wird. Das bedeutet Hazard (H2.1) besteht in dem Fall, dass kein Hindernis für die Fahrt erkannt wurde, und Fahrerlaubnis erteilt wurde, obwohl ein Hindernis besteht.

$$\mathrm{FE1} \wedge \mathrm{KH1} \wedge \neg\,\mathrm{ZV1} \tag{H2.1}$$

Hazard (H2.2) besteht, wenn für zwei verschiedene Züge Fahrerlaubnis für denselben Streckenabschnitt gegeben wurde:

$$(F1 \neq F2) \wedge \mathrm{FE1} \wedge (\mathrm{FE2} \vee \mathrm{FE2Rev}) \tag{H2.2}$$

10.5 Ebene 3



10.5.1 Unvollständigkeit der FV-NE

Es zeigt sich, dass die in der FV-NE beschriebenen Verfahren eine vollständige Durchführung des Zugleitbetriebsverfahren nicht garantieren können. Die hier verwendeten Zustandsmaschinen sind die von Lamport in [42] vorgestellten Predicate-Action-Diagrammen[42], die sich vollständig auf eine Spezifikation in TLA abbilden lassen.

Im Folgenden soll eine Verfeinerung stattfinden, die Kommunikation einschliesst. Zuverlässige Protokolle hierfür sind lange bekannt und beschrieben.

Erweiterung des Vokabulars

Es gibt als neues Objekt eine Nachricht, die gesendet und empfangen werden kann. Diese Nachricht kann eine der vier Zuglaufmeldungen *Fahranfrage* (FA). *Fahrerlaubnis* (FE), Ablehnung der Fahrerlaubnis (AFE) und *Ankunftmeldung* (AM) sein.

$\mathrm{Sent}(N, \text{Parameter})$	Nachricht N wurde gesendet
$\mathrm{Recd}(N, \text{Parameter})$	Nachricht N wurde korrekt empfangen

Diese definieren jeweils folgendermaßen die entsprechenden Prädikate aus Ebene 2:

10.5.2 Zustandsautomaten mit Kommunikation

Unter Einbeziehung der Kommunikation zwischen Zugleiter und Zugführer kann das Predicate-Action-Diagramm erweitert werden, und stellt dann eine vollständige Beschreibung einer Fahrt von einer Zuglaufmeldestelle zur folgenden im Zugleitbetrieb dar.

Dieser nun komplexere Zustandsautomat kann in einen Message-Flow-Graph (MFG) überführt werden, der aus individuellen Zustandsmaschinen besteht, die Nachrichten austauschen. Die Zustandsänderungen einzelner Automaten sind darin senkrechte Linien, die Nachrichtenübermittlungen schräge Linien.

10.5.3 Message-Flow-Graph

Die mit $\widetilde{s_j}$ bezeichneten Zustände bedeuten, dass das *Rational Cognitive Model* des Beteiligten jeweils so aussieht, als wäre das Gesamtsystem in Zustand s_j, obwohl der tatsächliche globale Zustand ein anderer sein kann.

Wenn tatsächlich nur ein Zug auf der Strecke vorhanden ist, können gegenüber der Variante mit vertrauenswürdiger keine weiteren Hazards entstehen, da immer klar ist, für welchen Zug oder von welchem Zug die Meldungen sind. Auch ist immer die nächste Zuglaufmeldestelle eindeutig.

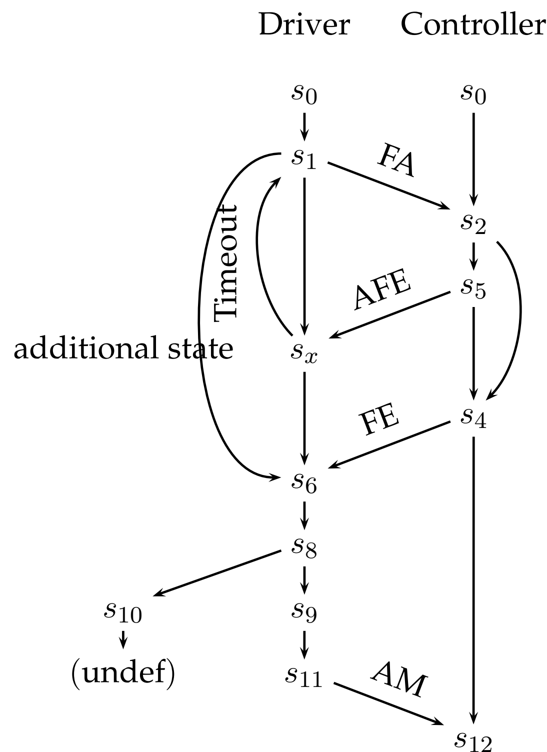


Abbildung 10.5: Message Flow Graph

10.6 Implementation in SPARK

10.6.1 SPARK

SPARK ist eine Programmiersprache und ein Spezifikationssystem [1] der britischen Firma Praxis High-Integrity Systems. Es basiert auf einer Untermenge der Programmiersprache *Ada* und benutzt *Annotations*, um Daten- und Informationsfluss zu beschreiben, sowie Pre- und Post-Conditions für Funktionen und Prozeduren zu definieren.

Zum SPARK-Toolset gehört unter anderem ein statischer Code-Analyser, der die Annotationen benutzt, um die Abwesenheit bestimmter Klassen von Laufzeitfehlern zu beweisen, beispielsweise Division durch Null, oder Buffer-Overflows oder andere Bereichsüberschreitungen, noch bevor der Code übersetzt wird.

Der Proof-Checker verwendet die Pre- und Postconditions, um zu beweisen, dass die Implementation des Spezifikationen genügt.

Von Altran Praxis entwickelte Software-Systeme haben die geringsten gemessenen Fehlerraten von im Einsatz befindlichen Software-Systemen.

10.6.2 Spezifikation der Funktionen

Ada-Code enthält in sogenannten Annotationen (aus Ada-Sicht syntaktisch Kommentare) Beschreibungen des Daten- und Informationsflusses durch jede Prozedur Funktion, und Pre- und Postconditions für jede Prozedur und Funktion.

Diese Spezifikationen befinden sich üblicherweise in getrennten Dateien, so dass eine klare Trennung zwischen Spezifikation und Implementation gegeben ist.

Beispiel-Spezifikation in SPARK

```
procedure Prepare_Next_Move (For_Train : in Train.ID_T);
--# global in out C_State;
--# derives C_State from
--#            *,
--#            For_Train;
--# pre State_Of(For_Train, C_State) = C_S12;
--# post State_Of(For_Train, C_State) = C_S0;
```

Abbildung 10.6: Beispiel einer Prozedur-Spezifikation in SPARK

Abbildung 10.6 zeigt eine solche Spezifikation, in diesem Falle der Prozedur `Prepare_Next_Move` (©2006 Phil Thornley).

Die „`derives`"-Annotation beschreibt, welche Variablen aus den Ursprungswerten welcher anderen Variablen abgeleitet werden in dieser Prozedur. In diesem Falle wird der Zustand des Zugleiters (`C_State`, *Controller State*), aus dem ursprünglichen Wert von sich selbst ($\star$) und dem Parameter `For_Train` abgeleitet.

Die Precondition dieser Prozedur ist, dass der Zustand des Zugleiter in Bezug auf Zug `For_Train` dem Zustand s_{12} des Message Flow-Graphs entspricht. Postcondition ist, dass nach Ende der Prozedur dieser Zustand s_0 entspricht, das heißt, es beginnt

nun ein neuer Abschnitt.

10.6.3 Zugführer

Die Prozeduren und Funktionen in `Driver/drivers.ads` entsprechen jeweils einem Übergang eines Teilautomaten des MFG.

Übergang ZF	SPARK-Prozedur	Precond.	Postcond.
$s_0 \rightarrow s_1$	Send_FA	D_State(DS) = D_S0	To_S1(DS , DS)
$s_1 \rightarrow s_x$	Process_AFE	—	(D_State(DS)=D_S1->To_Sx(DS , DS)) and (D_State(DS)/=D_S1->DS =DS)
$s_1 \rightarrow s_6$ $s_x \rightarrow s_6$	Process_FE	D_State(DS)=D_S1 or D_State(DS)=D_Sx	To_S6(DS , DS)
$s_6 \rightarrow s_8$	Start_Move	D_State(DS)=D_S6	To_S8(DS , DS)
$s_8 \rightarrow s_9$	Check_End_Move	D_State(DS)=D_S8	(Compl_Mv(DS,Tr.Pos.Sensor)->To_S9(DS ,DS)) and (not Compl_Mv(DS, Tr.Pos.Sensor)->DS =DS)
$s_9 \rightarrow s_{11}$	Send_AM	D_State(DS)=D_S9	To_S11(DS , DS)

Abbildung 10.7: Prozeduren für Zugführer

Abbildung 10.7 zeigt den Zusammenhang zwischen den Prozeduren des SPARK-Codes und den Übergängen im Zustandsautomaten des Fahrers.

Dabei stellt in den Postconditions ein mit einer Tilde ( ) versehener Bezeichner den Zustand der Variable vor Beginn der Prozedur dar, den sogenannten „importierten“ Wert. `D_State(DS)` liefert den aktuellen Zustand der abstrakten State Machine, die den Zustand des jeweiligen Zugführers enthält.

Die Precondition garantiert jeweils, dass die Prozedur nur ausgeführt wird, wenn sich der Zustandsautomat in einem der Zustände befindet, aus dem der jeweilige Übergang erlaubt ist. In der Regel ist dies nur ein einzelner Zustand. Hiervon gibt es zwei Ausnahmen: Zum einen Prozedur `Process_AFE`. Da Die Ablehnung der Fahrerlaubnis als Broadcast-Nachricht behandelt wird, muss diese in jedem Zustand behandelt werden. Dies spiegelt sich auch in der Postcondition wieder, die den Übergang in den nächsten Zustand nur vorsieht, wenn der vorherige Zustand s_1 war. Die andere Ausnahme ist `Process_FE`: die Fahrerlaubnis kann sowohl im Zustand s_1 als auch im Zustand s_x empfangen werden.

Die automatischen und halbautomatischen Beweise mit Hilfe des Proof-Checker der SPARK-Tools gewährleisten, dass die Implementation der SPARK-Funktionen und -Prozeduren die Nachbedingungen (postconditions) erfüllt.

Diese detaillierten Beweise, dass der Code alle Bedingungen in den Annotations erfüllt, sind von Thornley abgeschlossen worden.

10.7 Model-Checking mit SPIN

Mit Hilfe der bisher gezeigten Methode des Final Refinement mit Bedeutungspostulaten und Sicherheitsaxiomen kann gezeigt werden, dass die Sicherheitsaxiome der ersten Ebene, die als vollständig erkannt wurden, von allen folgenden Ebenen erfüllt werden.

Dies gewährleistet sicheren Betrieb. Darüberhinaus muss aber auch gezeigt werden, dass das beschriebene Verfahren überhaupt funktioniert.

Ein Kriterium für ein funktionierendes Bahnbetriebsverfahren ist, dass jeder Zug an seinem Ziel ankommt.

Um dies zu zeigen, kann ein Model-Checker verwendet werden. Der Model-Checker SPIN[21] bietet sich an, da in ihm direkt die von mir verwendeten kommunizierenden Zustandsautomaten abgebildet werden können.

SPIN überprüft Modelle, die in der Beschreibungssprache PROMELA (PROcess MEta LAnguage) geschrieben sind.

Zu bemerken ist, dass PROMELA sehr gut geeignet ist, Kontrollstrukturen in Software nachzubilden, oft ist zum Beispiel eine direkte Konvertierung von C-Programmen in PROMELA-Modelle möglich. Insbesondere ist es geeignet, nebenläufige, kommunizierende Programme zu modellieren und zu überprüfen.

10.7.1 Der SPIN-Model-Checker

Design

SPIN ist für die Verifikation von Software ausgelegt, und nicht für Hardware. Das System beziehungsweise die Algorithmen werden in einer high-level-Sprache (PROMELA) beschrieben. SPIN wurde verwendet für das logische Design von verteilten Systemen, zum Beispiel für Telefon-Switches [23], Betriebssysteme [9], und auch bereits für Bahnsysteme [14]. Das Softwaretool prüft logische Konsistenz und meldet Deadlocks, Race Conditions und andere Probleme.

Für diese Arbeit ist vor allem die Möglichkeit des vollständigen Beweises interessant. Das System ist beschränkt genug, dass dies Erfolg verspricht.

Die Modellierungssprache PROMELA

PROMELA (PROcess MEta Language) ist eine nicht-deterministische Sprache, ähnlich Dijkstras *guarded command language* [6], erweitert um die Ein-/Ausgabefunktionen von Hoares CSP-Sprache [19].

10.7.2 Implementation der kommunizierenden Zustandsautomaten in PROMELA

Durch Model-Checking soll für eine konkrete Bahnanlage bewiesen werden, dass der vorgeschlagene Algorithmus des Zugleitbetriebs mit digitaler Kommunikation seine Aufgabe erfüllt, nämlich erstens, dass keine Kollision stattfindet, und zweitens, dass alle Züge an ihrem Ziel ankommen. Der erste Punkt wird durch das Formal Refinement der Ontological Hazard Analysis garantiert, der zweite Punkt soll wird durch das Model-Checking gewährleistet.

Um ein handhabbares Szenario zu schaffen, gelten folgende Annahmen:

- Die gesamte Strecke ist ein geschlossener Kreis
- es gibt n Bahnhöfe
- es gibt m Züge
- Die Zahl der Bahnhöfe ist größer als die Zahl der Züge
- Alle Bahnhöfe sind zweigleisig mit Rückfallweichen
- Die Strecken zwischen den Bahnhöfen sind eingleisig

Für jeden Zug und für den Zugleiter wird jeweils ein Prozess gestartet. Diese Prozesse kommunizieren durch synchrone Kommunikation miteinander. Diese Kommunikation wird die Prozesse untereinander synchronisieren.

Für jeden der Züge gibt es jeweils die relevanten Prädikate und Funktionen, die exakt die Prädikate der Predicate-Action-Diagramme und damit auch des Message-Flow-Graphs abbilden.

Im normalen Zugleitbetrieb gibt es nur einen Zugleiter, der mehrere Züge in einem größeren Bereich kontrolliert. Eine Modellierung mit jeweils einem Zugleiter-Prozess

pro Zug ist jedoch möglich, da alle Zugleiter-Prozesse auf die Zustände aller Züge Zugriff haben. Die verschiedenen Zugleiter-Prozesse sind ein einfacher Weg, die unterschiedlichen Kommunikationskanäle des in der Realität einzigen Zugleiters unkompliziert in PROMELA zu beschreiben.

Die Strecke, das heißt die Folge der Stationen, die jeder Zug zu fahren hat, wird zu Anfang des Model-Checking festgelegt. Eine explizite Darstellung von Echtzeit ist nicht vorgesehen.

Entsprechend der Übergänge im Message Flow Graph ändern sich in jedem einzelnen Zustandsautomaten die Werte der Prädikate.

Zustände und Globale Variablen

Für dieses Model wird nur synchrone Kommunikation verwendet, da auch im Message Flow Graph synchrone, zuverlässige Kommunikation betrachtet wird.

Es werden pro Zug zwei Prozesse erzeugt, je einer für den Zugführer des Zuges und einer für den Zugleiter. Zwar gibt es nur einen Zugleiter, jedoch muss für jeden Zug getrennt ein Zugleiter-Zustandsautomat betrachtet werden, mit unabhängigen Zuständen.

Der Übergang in den jeweiligen Zustand findet statt, in PROMELA wiedergegeben durch einen Sprung zum Label, das dem jeweiligen Zustand entspricht, wenn die Bedingungen für den Übergang in den jeweiligen Zustand gegeben sind, das heißt, wenn die Prädikate alle den entsprechenden Wert haben.

Die symbolischen Labels, die die Namen der Zustände aus den kommunizierenden Zustandsmaschinen des MFG haben, dienen vor allem der Strukturierung des Codes während seiner Erstellung. Der tatsächliche Nachweis der Entsprechung von PROMELA-Maschinen und denen des MFG findet in Abschnitt 10.7.8 anhand der von SPIN erzeugten und ausgegebenen Zustandsmaschinen statt.

Synchronisation

SPIN bietet zwei Möglichkeiten, Prozesse zu synchronisieren:

Globale Variablen Hierdurch können bestimmte Prozesse blockiert werden, oder die Ausführung in diesen Prozessen gesteuert werden.

Kommunikation SPIN bietet sowohl synchrone als auch asynchrone Kommunikationskanäle. Bei synchroner Kommunikation, die hier verwendet wird, wird ein Prozess, der auf den Empfang einer Nachricht wartet, solange blockiert, bis der Sender die Nachricht übermittelt. Ebenso wird der Prozess, der die Nachricht sendet, blockiert, bis ein Prozess, der geschedult werden kann, die Nachricht annehmen kann.

Besonderheiten

- Richtung

  Jeder Zug startet an einer bestimmten Station und fährt zu einer bestimmten anderen Station, und erreicht diese Station in einer bestimmten vorgegebenen Richtung.
- Ende der Fahrt

  Am Ende der Fahrt, das heißt, wenn der vorgegebene Zielbahnhof erreicht ist, wird der Zug auf einem Nebengleis abgestellt. Dies wird im Modell dadurch dargestellt, dass für keine Zuglaufmeldestelle A_k das Prädikat $\mathrm{inZ}(F, A_k)$ wahr ist.
- $\mathrm{KH}(F_i, A, \mathrm{Next}(F_i, A))$

  Dies kann für den Zug F_i in A nur dann wahr werden, wenn für alle anderen Züge $F_k, i \neq k$ gilt:

$$
\begin{aligned}
&\wedge \neg\,\mathrm{ZV}(F_k, A, \mathrm{Next}(F_i, A)) \\
&\wedge \neg\,\mathrm{ZV}(F_k, \mathrm{Next}(F_i, A), A) \\
&\wedge \neg\,\mathrm{LV}(F_k, A, \mathrm{Next}(F_i, A)) \\
&\wedge \neg\,\mathrm{LV}(F_k, \mathrm{Next}(F_i, A), A) \\
&\wedge \vee \neg\,\mathrm{inZ}(F_k, \mathrm{Next}(F_i, A)) \\
&\quad \vee \mathrm{Next}(F_k, \mathrm{Next}(F_i.A)) = A
\end{aligned}
$$

  Die letzte Zeile stellt sicher, dass ein anderer Zug F_k, der in der Station steht, in die der betrachtete Zug F_i fahren soll, dem Zug entgegenkommt. Durch die Rückfallweichen ist gewährleistet, dass dann der andere Zug auf dem anderen Gleis steht, und der betrachtete Zug einfahren kann.

Ablauf der Zugfahrten im Modell

Im Initialisierungsprozess `init` von SPIN werden nicht-deterministisch Fahrstrecken für alle Züge festgelegt. Die aufwendig scheinende Initialisierungsroutine dafür garantiert, dass für eine festgelegte Anzahl Stationen und Züge bei einem Verifikationslauf von SPIN (Model-Checking) *alle* möglichen Kombinationen überprüft werden. Im Simulationsmodus wird nicht-deterministisch („zufällig“) eine Möglichkeit festgelegt, und diese simuliert.

Der Ablauf der einzelnen Prozesse wird sich strikt an die kommunizierenden State Machines der Message Flow Graphs halten. Da diese die Fahrt von einer Zuglaufmeldestelle zur folgenden beschreiben, sind folgenden Fälle darüber hinaus zu behandeln:

- Nach Ankunft an einer Zuglaufmeldestelle und Abgeben der Ankunftmeldung wird geprüft, ob sich der Zug an seinem endgültigen Ziel befindet. Ist dies der Fall, wird das Prädikat $\mathrm{inZ}(F, A)$ auf falsch gesetzt; der Zug ist damit nicht mehr auf dem normalen Gleis, und behindert keine anderen Züge.
- Ist der Zug nach Ankunft an einer ZMS und Abgabe der Ankunftmeldung noch nicht an seinem endgültigen Ziel, wird der Wert der Variablen, die die nächstfolgende ZMS anzeigt angepasst, und der zugehörige Zustandsautomat springt wieder in den Zustand (zum label) s_0.

10.7.3 Assertions

Die Einhaltung der Sicherheitsaxiome wird bei der Ontological Analysis gewährleistet durch das Formal Refinement und die Meaning-Postulates.

Dies stellt nur sicher, dass keine der Sicherheitsaxiome verletzt werden, erlaubt jedoch keine Aussage darüber, ob tatsächlich ein Betrieb stattfinden kann. Wenn alle Züge auf ihrer Ausgangsstation stehen blieben, könnte dies ein sicherer Betrieb sein, wäre aber nutzlos.

Um einen Betrieb zu ermöglichen, müssen jedoch auch eine Anzahl Liveness-Properties bewiesen werden. Dies sind:

- Jeder Zug kommt irgendwann (*eventually*) an seiner Zielstation an.
- Jeder Zug kommt in der richtigen Reihenfolge an allen Stationen, die auf seinem Weg liegen, vorbei.

10.7.4 Lineare Strecke

Alle möglichen Ausgangssituationen mit 4 Zügen sollen im Model-Checker getestet werden. Die Bedingungen sind dabei so anzunehmen: Alle Zuglaufmeldestellen sind zweigleisig mit Rückfallweichen, so dass in einer Station zwei Züge in entgegengesetzter Fahrtrichtung gleichzeitig stehen können, aber nicht in gleicher Fahrtrichtung. Ob dabei die Züge jeweils rechts oder links einfahren ist irrelevant, entscheidend ist nur, dass beide Einfahrtweichen gleich funktionieren, so dass entgegenkommende Züge stets auf den unterschiedlichen Gleisen stehen.

Zur Vereinfachung des Ablaufs der Prozesse fahren alle Züge stets bis zum Ende der Strecke. Züge, die am Beginn der Strecke starten, fahren nicht aus der Strecke heraus. Dies schränkt die Anzahl der möglichen Situationen weiter ein.

Situationen

Tabelle 10.12 zeigt die Auflistung aller möglichen Anfangs-Plazierungen. In den Spalten ist jeweils aufgeführt, welche Züge (0 bis 3) sich in welchen Zuglaufmeldestellen („Stationen“, 0 bis 4) befinden. Es wird jeweils die Umgebung eines Zuges 0 betrachtet, der sich in der Mitte einer Strecke mit 4 Abschnitten befindet. In dieser Umgebung befinden sich 3 weitere Züge, 1 bis 3. Die möglichen Verteilungen dieser weiteren Züge werden systematisch aufgelistet, unter der Rahmenbedingung, dass in jeder Station gleichzeitig 0, 1 oder 2 Züge sein können.

10.7.5 Zustandsmaschinen

Der Model-Checker SPIN erzeugt aus dem PROMELA-Code kommunizierende Zustandsmaschinen. Der Verifier, der aus dem Code erzeugt wird, kann diese State-Machine nach den ersten Optimierungsschritten ausgeben. Der wichtigste dieser Optimierungsschritte ist Partial Order Reduction [22] mit *statement merging* [20]. Dabei werden wo immer möglich statements zu einem Zustand des Automaten zusammengefasst (vergleichbar mit dem PROMELA-Konstrukt `D_STEP`) so dass das Ergebnis der Verifikation nicht beeinflusst wird.

Der Verifier, den SPIN aus der Beschreibung in PROMELA erzeugt, ist ein C-Programm, dass mit einem normalen C-Compiler übersetzt werden kann, und dann die Zustandsraumsuche sehr effizient ausführt. Ein unkomprimiertes Abspeichern

Stationen / Situation	0	1	2	3	4	Bemerkungen
A	1 2	3	0	–	–	1)
B	1 2	–	3 0	–	–	1)
C	1 2	–	0	3	–	1)
D	1 2	–	0	–	3	1)
E	1	2 3	0	–	–	
F	1	2	3 0	–	–	
G	1	2	0	3	–	
H	1	2	0	–	3	
I	1	–	2 0	3	–	
J	1	–	2 0	–	3	
K	1	–	0	2 3	–	
L	1	–	0	2	3	2)
M	1	–	0	–	2 3	1),3)
N	–	1 2	3 0	–	–	
O	–	1 2	0	3	–	
P	–	1 2	0	–	3	4)
Q	–	1	2 0	3	–	
R	–	1	2 0	–	3	5)
S	–	1	0	2 3	–	6)
T	–	1	0	2	3	7)
U	–	1	0	–	2 3	1),8)
V	–	–	0 1	2 3	–	9)
W	–	–	0 1	2	3	10)
X	–	–	0 1	–	2 3	1),11)
Y	–	–	0	1	2 3	1),12)

Tabelle 10.12: Liste aller möglichen Ausgangssituationen für 4 Züge und 5 Stationen

```
/* Situation E: 0--1--2--3--4
                -  3  -  -  -
                1  2  0  -  -
 */
start_station[1] = 0;
start_station[2] = 1;
start_station[3] = 1;

direction[1] = forward;

direction[2] = forward;
direction[3] = backward;

if
::direction[0] = forward;
::direction[0] = backward;
fi;
```

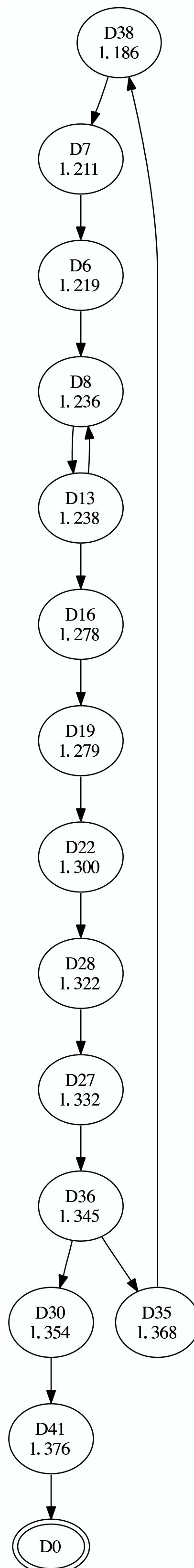
Abbildung 10.8: Beispiel eines Include-Files für den Model-Checkers

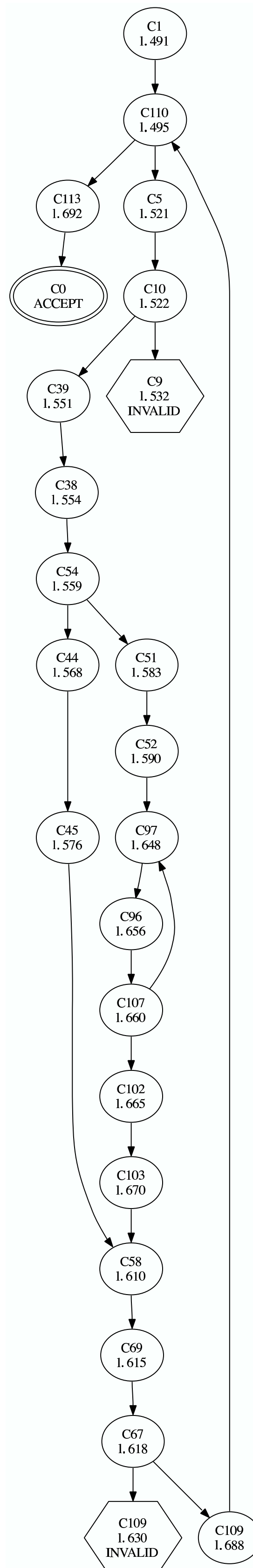
aller möglichen globalen Zustände würde den Rahmen heute für private Anwendung üblicher Computer sprengen. Daher verwende ich in diesem Fall Graph-Encoding [15], mit dem bei geringerer Zustandssuchrate eine Kompression des benötigten Datenvolumens auf ca. 1/500 des sonst nötigen Speichers möglich ist. Trotz der erheblichen Reduktion der Suchrate um ein bis zwei Größenordnungen, ist diese Methode bei Speicherplatzknappheit erheblich schneller als die Nutzung von durch das Betriebssystem verwaltetem virtuellen, auf externe Massenspeicher (Festplatte) augelagerten Speicher.

Abbildungen 10.9 und 10.10 zeigen die Zustandsautomaten, die SPIN aus den jeweiligen Prozessbeschreibungen erzeugt hat, und die für die Verifikation verwendet werden. Die Nummern in den Zustandsknoten sind vom Model-Checker vergeben, darunter steht jeweils die Nummer der Code-Zeile im PROMELA-File des Zustands.

10.7.6 Verifikation

Alle 10 Durchläufe des Verifiers konnten abgeschlossen werden. Dabei wurden keine ungültigen Endzustände gefunden. Dies bedeutet, dass alle Prozesse in ihrem vorgesehenen Endzustand ankommen, der nur dann erreicht werden kann, wenn der betreffende Zug an seiner Zielstation angelangt ist.


D38
l. 186
D7
l. 211
D6
l. 219
D8
l. 236
D13
l. 238
D16
l. 278
D19
l. 279
D22
l. 300
D28
l. 322
D27
l. 332
D36
l. 345
D30
l. 354
D35
l. 368
D41
l. 376
D0


C1
l. 491
C110
l. 495
C113
l. 692
C5
l. 521
C0
ACCEPT
C10
l. 522
C39
l. 551
C9
l. 532
INVALID
C38
l. 554
C54
l. 559
C44
l. 568
C51
l. 583
C52
l. 590
C45
l. 576
C97
l. 648
C96
l. 656
C107
l. 660
C102
l. 665
C103
l. 670
C58
l. 610
C69
l. 615
C67
l. 618
C109
l. 630
INVALID
C109
l. 688

Stationen	0	1	2	3	4
Situation					
E	1	2 3	0	–	–
F	1	2	3 0	–	–
G	1	2	0	3	–
H	1	2	0	–	3
I	1	–	2 0	3	–
J	1	–	2 0	–	3
K	1	–	0	2 3	–
N	–	1 2	3 0	–	–
O	–	1 2	0	3	–
Q	–	1	2 0	3	–

Tabelle 10.13: Liste der zu überprüfenden Ausgangssituationen für 4 Züge und 5 Stationen

Abbildung 10.11 zeigt die Ausgabe des Verifiers an einem Beispiel. Die Ausgabe bedeutet im Einzelnen:

1 SPIN-Version

2 Partial Order-Reduction wurde verwendet (+)

3 State-Compression wurde verwendet

4 Graph-Encoding für die States wurde verwendet. Langsam, aber sehr speichereffizient

```
 1: (Spin Version 5.1.6 -- 9 May 2008)
 2:         + Partial Order Reduction
 3:         + Compression
 4:         + Graph Encoding (-DMA=25)
 5:
 6: Full statespace search for:
 7:         never claim             - (none specified)
 8:         assertion violations    +
 9:         cycle checks            - (disabled by -DSAFETY)
10:         invalid end states      +
11:
12: State-vector 524 byte, depth reached 550, errors: 0
13: MA stats: -DMA=23 is sufficient
14: Minimized Automaton:    243123 nodes and 1.17092e+06 edges
15:  33521384 states, stored
16:  68469880 states, matched
17: 1.0199126e+08 transitions (= stored+matched)
18:  22183639 atomic steps
19: hash conflicts:         0 (resolved)
20:
21: Stats on memory usage (in Megabytes):
22: 17262.981       equivalent memory usage for states (stored*(State-vector + overhead))
23:    32.677       actual memory usage for states (compression: 0.19%)
24:     2.000       memory used for hash table (-w19)
25:     0.267       memory used for DFS stack (-m10000)
26:    35.080       total actual memory usage
27:
28: nr of templates: [ globals chans procs ]
29: collapse counts: [ 42118 11 174 210 ]
30: unreached in proctype :init:
31:         (0 of 36 states)
32: unreached in proctype driver
33:         (0 of 41 states)
34: unreached in proctype controller
35:         line 544, state 9, "assert(0)"
36:         line 642, state 66, "assert(0)"
37:         (2 of 113 states)
38:
39: pan: elapsed time 1.87e+03 seconds
40: pan: rate 17922.232 states/second
41:
```

Abbildung 10.11: Ausgabe eines SPIN-Verifier-Durchlaufs

7 Es wurde keine Überprüfung von Never-Claims durchgeführt, da keine solchen angegeben wurden.

8 Der Code wurde auf Verletzung von Assertions überprüft.

9 Es wurde keine Überprüfung auf acceptance-Cycles durchgeführt.

10 Eine Suche nach invalid end states (timeouts/deadlocks) wurde durchgeführt.

12 Zur Speicherung des Zustandsvektors wurden jeweils 524 Bytes benötigt, maximale Suchtiefe war 550, es sind keine Fehler aufgetreten.

13 Eine Suche mit geringerer Zustandskompression wäre bei gegebenen Speicherbedarf-Constraints möglich gewesen. Da die Suche aber trotzdem in akzeptabler Zeit beendet werden konnte, ist dies unerheblich.

14 Der minimierte globale Zustandsautomat hatte 243123 Knoten und 1,17 Millionen Kanten

15–18 Insgesamt wurden ca. 108 Millionen Übergänge geprüft, dabei sind 36243553 eindeutige Zustände gespeichert worden, 72101724 mal wurden Zustände mehrmals besucht. 23072395 Übergänge waren Teil einer `atomic`-Sequenz.

19 Es sind keine Hash-Konflikte aufgetreten.

21–26 Ohne Kompression und Graph-Encoding wären ca. 17 GB Speicher erforderlich gewesen. Heute nicht mehr absurd, aber noch nicht alltäglich. Durch Kompression und Graph-Encoding waren nur knapp 33 MB erforderlich, das entspricht einer Kompression auf 0,19%. Außerdem waren 2 MB für die Hash-Table und 0,267 MB für den DFS- (depth-first-search-) Stack erforderlich.

30–37 Im init-Prozess und im Driver-Prozess waren keine unerreichbaren Zustände. Im Controller-Prozess waren nur die beiden „assert(0)"-Statements unerreichbar. Diese wären nur im Falle des Empfangs eines falschen Nachrichtentyps erreicht worden. Die Nichterreichbarkeit ist damit ein Indiz für das Funktionieren des Protokolls.

39–40 Für diesen Verifikationslauf wurden 1870 Sekunden benötigt, bei einer Rate von 17922 Übergängen pro Sekunde.

Der Zeit- und Speicheraufwand war auch auf einem heute vergleichsweise kleinen und langsamen Computer (AthlonXP 2400+, 512MB Hauptspeicher) unter NetBSD-3[54] vertretbar.

Tabelle 10.14 zeigt die jeweilige Größe des überprüften globalen Zustandsautomaten, sowie die für die Verifikation benötigten Ressourcen und die erzielte Testrate in Übergängen pro Sekunde.

10.7.7 Meaning Postulates

Die Implementation des Verfahrens in PROMELA lehnt sich eng an die kommunizierenden Zustandsautomaten im Message Flow Graph an. Es muss hier definiert werden, welche Zustände der Automaten im Modell des Model-Checkers welchen Zuständen der Automaten im MFG entsprechen sollen.

Links stehen die PROMELA-Statements und Variablen, rechts die entsprechende globalen Zustände des Message Flow Graphs.

Situation	Übergänge	Speicher (MB)	Zeit (s)	Rate (s^{-1})
E	1.0834×10^8	37	2490	14569
F	1.3066×10^8	40	3220	13382
G	5.5399×10^8	159	16300	11312
H	6.1950×10^8	150	17300	11840
I	1.4895×10^8	44	3860	12826
J	1.7152×10^8	46	4330	13131
K	1.1421×10^8	35	2510	14980
N	2.4471×10^7	11	444	18017
O	1.0199×10^8	35	2080	16103
Q	1.2837×10^8	41	2960	14335

Tabelle 10.14: Benötigte Ressourcen für die Verifikation

Ein Statement wie `t2c!ZLM` bedeutet dabei, dass das Statement beendet ist. Das `!` in PROMELA steht für das Senden der Nachricht ZLM über den Kanal `t2c`. `t2c` ist in diesem Fall eine Kurzschrift für *train to controller*. Zu beachten ist, dass im MFG und in den Zustandsautomaten der vorhergehenden Ebenen, Sender und Empfänger implizit waren, da bestimmte Nachrichtentypen immer nur von einem bestimmten Sender gesendet werden können: Fahranfragen (FA) und Ankunftmeldungen (AM) immer nur vom Zugführer, Fahrerlaubnis (FE) und Ablehnung der Fahrerlaubnis (AFE) immer nur vom Zugleiter (*controller*).

Wenn nichts anderes explizit angegeben ist, sei immer F ein beliebiger, aber fester Zug, A und B Zuglaufmeldestellen.

Eine weitere Besonderheit ist auch noch die Verwendung synchroner Kommunikation, das bedeutet, dass die Aufrufe von sendendem und empfangendem Statement immer gleichzeitig beendet werden. Wenn eine Nachricht in einem Prozess gesendet wurde, steht sie sofort dem empfangenden Prozess zur Verfügung, sobald dieser wieder geschedult wird.

$$\left.\begin{aligned}&\wedge\ \texttt{t2c[}F\texttt{]!ZLM}\\&\wedge\ \texttt{ZLM.type == FA}\\&\wedge\ \texttt{ZLM.train == }F\\&\wedge\ \texttt{ZLM.destination == }A\end{aligned}\right\}\Rightarrow \mathrm{Sent}(\mathrm{FA}\,,F,A) \tag{10.1}$$

$$\left.\begin{aligned}&\wedge\ \texttt{c2t[}F\texttt{]!ZLM}\\&\wedge\ \texttt{ZLM.type == FE}\\&\wedge\ \texttt{ZLM.train == }F\\&\wedge\ \texttt{ZLM.destination == }A\end{aligned}\right\}\Rightarrow \mathrm{Sent}(\mathrm{FE}\,,F,A) \tag{10.2}$$

$$\left.\begin{aligned}&\wedge\ \texttt{c2t[}F\texttt{]!ZLM}\\&\wedge\ \texttt{ZLM.type == AFE}\\&\wedge\ \texttt{ZLM.train == }F\\&\wedge\ \texttt{ZLM.destination == }A\end{aligned}\right\}\Rightarrow \mathrm{Sent}(\mathrm{AFE}\,,F,A) \tag{10.3}$$

$$\left.\begin{aligned}&\wedge\ \texttt{t2c[}F\texttt{]!ZLM}\\&\wedge\ \texttt{ZLM.type == AM}\\&\wedge\ \texttt{ZLM.train == }F\\&\wedge\ \texttt{ZLM.destination == }A\end{aligned}\right\}\Rightarrow \mathrm{Sent}(\mathrm{AM}\,,F,A) \tag{10.4}$$

$$\left.\begin{aligned}&\wedge\ \texttt{t2c[}F\texttt{]?ZLM}\\&\wedge\ \texttt{ZLM.type == FA}\\&\wedge\ \texttt{ZLM.train == }F\\&\wedge\ \texttt{ZLM.destination == }A\end{aligned}\right\}\Rightarrow \mathrm{Recd}(\mathrm{FA}\,,F,A) \tag{10.5}$$

$$\left.\begin{aligned}&\wedge\ \texttt{c2t[}F\texttt{]?ZLM}\\&\wedge\ \texttt{ZLM.type == FE}\\&\wedge\ \texttt{ZLM.train == }F\\&\wedge\ \texttt{ZLM.destination == }A\end{aligned}\right\}\Rightarrow \mathrm{Recd}(\mathrm{FE}\,,F,A) \tag{10.6}$$

$$\left.\begin{array}{l}\wedge\ \texttt{c2t[}F\texttt{]?ZLM}\\ \wedge\ \texttt{ZLM.type == AFE}\\ \wedge\ \texttt{ZLM.train == }F\\ \wedge\ \texttt{ZLM.destination == }A\end{array}\right\}\Rightarrow \mathrm{Recd}(\mathrm{AFE},F,A) \tag{10.7}$$

$$\left.\begin{array}{l}\wedge\ \texttt{t2c[}F\texttt{]?ZLM}\\ \wedge\ \texttt{ZLM.type == AM}\\ \wedge\ \texttt{ZLM.train == }F\\ \wedge\ \texttt{ZLM.destination == }A\end{array}\right\}\Rightarrow \mathrm{Recd}(\mathrm{AM},F,A) \tag{10.8}$$

$$\texttt{inZ[}F\texttt{].station[}A\texttt{] == true} \Rightarrow \mathrm{inZ}(F,A) \tag{10.9}$$

$$\left.\begin{array}{l}\text{Sei } n \text{ Anzahl der Züge:}\\ \text{Sei } F_i, i \in \{0,\cdots,n-1\}\text{, ein Zug:}\\ \forall F_k, k \in \{0,\ldots,n-1\}, k \neq i:\\ \wedge\texttt{inZ[}F_i\texttt{].station[}A\texttt{] == true}\\ \wedge\texttt{ZV[}F_k\texttt{].previous[}A\texttt{].next[Next[}F_i\texttt{]] == false}\\ \wedge\texttt{ZV[}F_k\texttt{].previous[Next[}F_i\texttt{]].next[}A\texttt{] == false}\\ \wedge \vee\ \texttt{inZ[}F_k\texttt{].station[Next[}F_i\texttt{]] == false}\\ \quad \vee\ \texttt{Next[}F_k\texttt{] == }A\end{array}\right\}\Rightarrow \mathrm{KH}(F,A) \tag{10.10}$$

$$\texttt{zw[}F\texttt{].previous[}A\texttt{].next[}b\texttt{]} \Rightarrow \mathrm{zw}(F,A,B) \tag{10.11}$$

$$\texttt{ZV[}F\texttt{].previous[}A\texttt{].next[}b\texttt{]} \Rightarrow \mathrm{ZV}(F,A,B) \tag{10.12}$$

$$\left.\begin{array}{l}\wedge\ \texttt{inZ[}F\texttt{].previous[}A\texttt{].next[}B\texttt{] == true}\\ \wedge\ \texttt{ZV[}F\texttt{].previous[}A\texttt{].next[}B\texttt{] == false}\end{array}\right\}\Rightarrow \mathrm{LV}(F,A,B) \tag{10.13}$$

10.7.8 Implementation der MFG durch die SPIN-Zustandsmaschinen

Um zu zeigen, dass der Model-Checker verwertbare Aussagen über die Zustandsmaschinen in den Message-Flow-Graphs macht, und damit zeigt, dass das hier entwickelte Protokoll einen Betrieb ermöglicht, muss gezeigt werden, dass die kommunizierenden Prozesse des PROMELA-Modells die kommunizierenden Zustandsautomaten des MFG implementieren.

Zu beachten ist dabei, dass die PROMELA-Prozesse zusätzliche Funktionen und Zustände aufweisen, die jeweils dazu dienen, am Ende eine Abschnitts alle Parameter neu zu setzen, so dass der folgende Abschnitt befahren werden kann, und am Ende der planmäßigen Fahrt eines Zuges diesen von der Strecke nimmt. In der Praxis kann dies das Verfahren des Zuges auf ein sonst nicht benutztes Nebengleis oder in ein Depot sein.

Die Zustandsmaschinen des MFG dagegen beschreiben nur einen Teilabschnitt einer Fahrt von einer Zuglaufmeldestelle zur folgenden.

Im folgenden sind für jeden Zustandsübergang des Zugführer- und Zugleiter-Prozesstyps im PROMELA-Modell die Statements aufgelistet, die für den jeweiligen Übergang abgearbeitet werden. Zusammen mit der Semantik-Definitionen von PROMELA und den sich daraus die Werte der Variablen im jeweils folgenden Zustand, und die Zustände, in denen eine Nachricht gesendet oder empfangen wurde.

Es sei jeweils: F der betrachtete Zug, A die Zuglaufmeldestelle, an der der betrachtete Fahrtabschnitt beginnt, $\mathrm{Next}(F,A) = B$ die Zuglaufmeldestelle, an der der betrachtete Fahrtabschnitt endet.

Die folgende Kurzschreibweise wird in den Tabelle verwendet:

inA — `inZ[`F`].station[`A`] == true`

inB — `inZ[`F`].station[`B`] == true`

zw — `zw[`F`].previous[`A`].next[`B`] == true`

S:FA — Das Statement `t2c[`F`]!tFA,`F`,`B wurde abgearbeitet

S:FE — Das Statement `c2t[`F`]!tFE,`F`,`B wurde abgearbeitet

S:AFE — Das Statement `c2t[`F`]!tAFE,`F`,`B wurde abgearbeitet

S:AM — Das Statement `t2c[`F`]!tAM,`F`,`B wurde abgearbeitet

R:FA — Das Statement `t2c[`F`]?tFA,`F`,`B wurde abgearbeitet

R:FE — Das Statement c2t[F]?tFE, F, B wurde abgearbeitet

R:AFE — Das Statement c2t[F]?tAFE, F, B wurde abgearbeitet

R:AM — Das Statement t2c[F]?tAM, F, B wurde abgearbeitet

ZV — Der Zug F belegt den Streckenabschnitt zwischen A und B unter zentraler Verantwortung

Statements, die nur dem Control-Flow dienen, und die keine Zustände beeinflussen, werden nicht betrachtet.

Zugführer

D38 → D7

```
do
:: true ->
```

Statement	inA	inB	zw	S:FA	R:FE	R:AFE	S:AM
Initial	✓	-	-	-	-	-	-
true	✓	-	-	-	-	-	-

D7 → D6

```
d_step {
   ZLM.type = tFA;
   ZLM.train = train;
   ZLM.destination = Next[train];
} /* end d_step */
```

Statement	inA	inB	zw	S:FA	R:FE	R:AFE	S:AM
Initial	✓	-	-	-	-	-	-
Zuweisung Nachricht	✓	-	-	-	-	-	-

D6 → D8

```
t2c[train]!ZLM;
```

Statement	inA	inB	zw	S:FA	R:FE	R:AFE	S:AM
Initial	✓	-	-	-	-	-	-
Senden Nachricht	✓	-	-	✓	-	-	-

D8 → D13

```
c2t[train]?ZLM;
```

Statement	inA	inB	zw	S:FA	R:FE	R:AFE	S:AM
Initial	✓	-	-	✓	-	-	-
Empfangen Nachr.	✓	-	-	✓	-	-	-

Die Nachricht ist zwar schon empfangen, aber noch nicht ausgewertet, daher ist noch nicht bestimmt, ob die empfangene Nachricht die Fahrerlaubnis oder die Ablehnung der Fahrerlaubnis ist.

D13 → D8

```
:: ZLM.type == tAFE ->
```

Statement	inA	inB	zw	S:FA	R:FE	R:AFE	S:AM
Initial	✓	-	-	✓	-	-	-
`ZLM.type==tAFE`	✓	-	-	✓	-	✓	-

Das Empfangen der Ablehnung der Fahrerlaubnis hat keine Auswirkung, Die Zustandsmaschine kehrt wieder in den Zustand D8 zurück, in dem weiterhin auf die Erteilung der Fahrerlaubnis gewartet wird. Der Zweck der Ablehnung der Fahrerlaubnis für menschliche Kommunikation ist, dass bei noch nicht vorliegenden Bedingungen für Erteilung der Erlaubnis, trotzdem eine Rückmeldung zu haben, so dass nicht unnötig nachgefragt wird und/oder eine Störung des Funkverkehrs vermutet wird.

D13 → D16

```
:: ZLM.type == tFE ->
```

Statement	inA	inB	zw	S:FA	R:FE	R:AFE	S:AM
Initial	✓	-	-	✓	-	-	-
`ZLM.type==tFE`	✓	-	-	✓	✓	-	-

D16 → D19

```
skip;
```

10.7.9 Beziehungen zwischen MFG, PROMELA-Modell und SPARK-Code

Es wurde nicht der SPARK-Code in das Modell übertragen, sondern die kommunizierenden Zustandsautomaten der Abstraktion der Ebene 3.

Um Aussagen über den SPARK-Quellcode und den Object-Code treffen zu können, ist eine genauere Betrachtung der Zusammenhänge notwendig.

Abbildung 10.12 zeigt, wie sich der Message-Flow-Graph jeweils zum PROMELA-Modell und zum SPARK-Source-Code verhält.

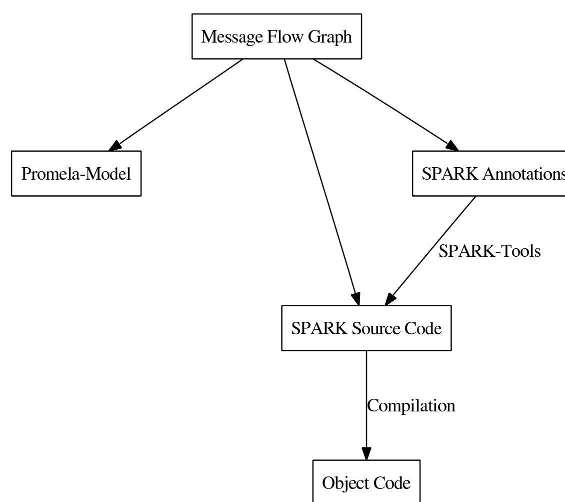


Abbildung 10.12: Beziehungen zwischen Message-Flow-Graph, SPARK Code und Promela-Modell

MFG — SPARK Source Code

In Kapitel 10.6 wurde gezeigt, wie Thornleys SPARK-Code den Message Flow Graph implementiert. Implementation bedeutet in diesem Fall, dass die Zustände der kommunizierenden Zustandsautomaten in Variablen in Ada beziehungsweise SPARK abgebildet werden. Dies gewährleistet, dass der Algorithmus im SPARK-Code, wann welche Zuglaufmeldungen von wem an wen abgegeben werden, und wann daraufhin

welcher Zug von welcher Station zu welcher anderen Station fahren kann, mit den im Message-Flow-Graph beschriebenen Verfahren übereinstimmt.

Das bedeute, algorithmisch erfüllt auch der SPARK-Source-Code die Sicherheitsanforderungen. Wenn es ein Problem mit den Computern in einer möglichen Anwendung dieses Verfahrens gibt, so können diese nicht am Verfahren an sich liegen. Dieses Verfahren kann auch in für Menschen verständlicher Schriftform verfasst werden, vorstellbar als Handbuch für den Triebfahrzeugführer. Dieser kann jeweils anhand der empfangenen Nachrichten und des Zustands seines Fahrzeugs darin die auszuführenden Schritte nachlesen und genau befolgen.

Unwägbarkeiten bleiben hier, wie bei einer Computerumsetzung, in der Zuverlässigkeit, mit der der Algorithmus ausgeführt wird.

An dieser Stelle kann durch diese Entsprechung allein keine Aussage darüber gemacht werden, welche anderen Bedingungen der SPARK-Source sonst erfüllt, oder ob der daraus übersetzte Object-Code weitere Bedingungen erfüllt, wie Freiheit von Laufzeitfehlern.

MFG — SPARK Annotations

Die Zustände der Automaten im MFG werden entsprechend den verwendeten Variablen in SPARK in Pre- und Postconditions übertragen. Diese müssen so formuliert sein, dass sie die Übergänge des Predicate-Action-Diagrams und der daraus abgeleiteten Message-Flow-Graphs abbilden.

MFG — PROMELA-Modell

In diesem Kapitel ist die Entsprechung von Message-Flow-Graph und dem PROMELA-Modell dargestellt worden, damit ist gewährleistet, dass auch der PROMELA-Code algorithmisch dem Message-Flow-Graphen entspricht. Das Model-Checking hat gezeigt, dass dieser Algorithmus einen Betrieb sicherstellt, das heißt, liveness-Eigenschaften hat. Da sowohl der SPARK-Code als auch das PROMELA-Modell auf demselben Algorithmus basieren, ist auch für den SPARK-Code sichergestellt, dass dieser die gewünschten Liveness-Eigenschaften hat. Weitere Aussagen lassen sich an dieser Stelle nicht über den SPARK-Code treffen.

SPARK Annotations — SPARK Source-Code

Aufgrund der besonderen Eigenschaften von SPARK kann man unter Voraussetzung von funktionierenden Compilern bestimmte Eigenschaften des Object-Codes anhand des Source-Codes nachweisen. Dies sind insbesondere wichtige Klassen von Laufzeit-Fehlern wie Division durch Null, Einhaltung von Array-Grenzen und Wertebereichen von Variablen, sowie Daten- und Informationsflussbedingungen. Diese Eigenschaften werden mit Hilfe des SPARK-Toolkits, bestehend aus dem SPARK-Examiner, dem Simplifier und dem Interactive Proof Checker automatisch oder interaktiv nachgewiesen. [1] gibt eine detaillierte Beschreibung der Benutzung und Fähigkeiten der SPARK-Tools.

SPARK-Quellcode — Object Code

Der verifizerte SPARK-Source-Code wird mit einem Ada-Compiler (und Linker) in ausführbaren Object-Code übersetzt. Algorithmische Korrektheit wurde durch den Nachweis der Entsprechung von MFG und Source-Code sowie der Überprüfung der Pre- und Postconditions gezeigt. Anhand der Data- und Information-Flow-Analyse durch die SPARK-Tools und die starke Typisierung der Programmiersprache Ada (auf der SPARK basiert) kann Abwesenheit von wesentlichen Klassen von Laufzeitfehlern nachgewiesen werden. Die Übersetzung durch die Compiler-Tools bleibt damit neben der Hardwarezuverlässigkeitsebene als Unsicherheitsfaktor für das fehlerfreie Funktionieren einer computergestützten Umsetzung des hier vorgestellten Protokolls.