Systemanforderungsanalyse von Bahnbetriebsverfahren mit Hilfe der Ontological Hazard Analysis am Beispiel des Zugleitbetriebs nach FV-NE

Dissertation zur Erlangung des Doktorgrades der Ingenieurwissenschaften (Dr.-Ing.)

an der Technischen Fakultät der Universität Bielefeld

Bernd Manfred Sieker

April 2010

Eine Eisenbahn ist ein Unternehmen, gerichtet auf wiederholte Fortbewegung von Personen oder Sachen über nicht ganz unbedeutende Raumstrecken auf metallener Grundlage, welche durch ihre Konsistenz, Konstruktion und Glätte den Transport großer Gewichtsmassen beziehungsweise die Erzielung einer verhältnismäßig bedeutenden Schnelligkeit der Transportbewegung zu ermöglichen bestimmt ist, und durch diese Eigenart in Verbindung mit den außerdem zur Erzeugung der Transportbewegung benutzten Naturkräften — Dampf, Elektrizität, tierischer oder menschlicher Muskeltätigkeit, bei geneigter Ebene der Bahn auch schon durch die eigene Schwere der Transportgefäße und deren Ladung usf. — bei dem Betriebe des Unternehmens auf derselben eine verhältnismäßig gewaltige, je nach den Umständen nur bezweckterweise nützliche oder auch Menschenleben vernichtende und menschliche Gesundheit verletzende Wirkung zu erzeugen fähig ist.

Reichsgericht (Urteil vom 17. März 1879; RGZ 1, 247, 252)

Ich danke meinem Betreuer und Gutachter, Prof. Peter Bernard Ladkin PhD für die langjährige gute Zusammenarbeit in seiner Arbeitsgruppe *Rechnernetze und Verteilte Systeme*, viele Anregungen, Ideen und Bemerkungen.

Besonderer Dank gilt auch meinem Zweitgutachter, Dr.-Ing. Ralf Pinger, der sich trotz vollen Terminkalenders die Zeit für eine detaillierte Auseinandersetzung mit meiner Arbeit genommen hat.

Ebenso danke ich dem Vorsitzenden meines Prüfungsausschusses, Prof. Dr. Franz Kummert, sowie den weiteren Ausschussmitgliedern Prof. Dr. Ipke Wachsmuth und Dr. Thorsten Schneider.

Ganz besonderer Dank geht an dieser Stelle an meine Familie: meine Frau Nicole und unsere Kinder Daniel, Susanna und Tabita, die mir Rückhalt gegeben, aber immer wieder auch Freiräume gelassen haben, an meiner Promotion zu arbeiten; sowie an meine Eltern, die mir eine gute Bildung ermöglicht haben und mich auf meiner akademischen Laufbahn unterstützt und motiviert haben.

Gedruckt auf alterungsbeständigem Papier °° ISO 9706

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einführung

Eisenbahnregelwerke sind traditionell in natürlicher Sprache verfasst. Dies führt zu zahlreichen Problemen beim Erstellen und Verifizieren von Lasten- und Pflichtenheften und bei der unzweideutigen Spezifikation von Systemen für ein bestimmtes Betriebsverfahren. Die Ontological Hazard Analysis ist eine Methode, mit der sich durch formale Verfeinerung („Formal Refinement“) vollständige Spezifikationen der Safety-Requirements erstellen lassen. Mit Hilfe des Model-Checkers SPIN sollen weitere Eigenschaften des Systems geprüft werden, die keine Sicherheitsanforderungen sind. Die Methode, die unter dem Namen *Ontological Analysis* von Ladkin [Lad05] beschrieben wurde, ist noch sehr jung, und ihre Anwendung auf Bahnbetriebsverfahren ist neu. Sie wurde bisher in etwas anderer Form angewendet von Stuphorn [Stu05] für ein Automobil-Kommunikationsbussystem, und vom Autor zusammen mit Stuphorn [SSL09] für Security-Anforderungen für Softwareübertragung auf Automobilkomponenten.

Ziel dieser Arbeit soll sein, *OHA*-Verfahren für diesen Bereich zu entwickeln. Der Zugleitbetrieb bietet sich hierfür besonders an, da er relativ überschaubar sind, aber dennoch in der Praxis Anwendung findet. Dass viele der Betriebsabläufe sich auf menschliches Verhalten stützen, stellt eine besondere Herausforderung bei der Formalisierung dar.

1.1 Methoden

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

1.1.1 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. Es kann vereinbart werden, dass Objekte der Sorte *Fahrzeug* stets als F, $F1$ oder $F2$ bezeichnet werden, d. h. das Prädikat $Fahrzeug(F)$ ist wahr.

Bedeutungspostulate (*Meaning Postulates*)[Car52] beschreiben die Bedeutung von Prädikaten (beispielsweise bei einem Verfeinerungsschritt der OHA) mit bereits bestehenden Elementen der Systembeschreibung. Beispielsweise beim Schritt vom allgemeinen Streckenabschnitt zur Definition der Zuglaufmeldestellen und eines Gleises im Bahnhof kann der Streckenabschnitt durch die neuen Objekte und Prädikate definiert werden, und damit gezeigt, dass es sich um eine Verfeinerung handelt.

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. Sicherheits-Axiome können aus der Systembeschreibung abgeleitet werden, und die Ontological Hazard Analysis kann deren (relative) Vollständigkeit garantieren.

1.1.2 Implementation in SPARK

Die Methode erlaubt bei Verwendung entsprechender Correct-by-Construction-Methoden eine Nachvollziehbarkeit (*Traceability*) der Sicherheitsaxiome von der einfachsten Abstraktion bis zum Source-Code.

1.1.3 Model-Checking

Liveness-Anforderungen an das entwickelte Protokoll können mit einem Model-Checker überprüft werden, der sicherstellt, dass dies neben den Sicherheitsanforderungen auch Betriebsanforderungen erfüllen kann.

Kapitel 2

Herstellung einer Systemdefinition

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.

2.1 Systemgrenzen

Die Grenzen des betrachteten Systems ergeben sich bei der Ontological Hazard Analysis bei jeder Stufe erneut aus den betrachteten Objekten. Alle Objekte, die benötigt werden, um das Verfahren auf der jeweiligen Abstraktionsebene zu beschreiben, sind Teil des Systems

Dies entspricht einer flussbasierten Systemabgrenzung (*„fluxional" boundary*), bei der die Auswahl der Objekte, die zum System gehören, so erfolgt, dass der Informationsfluss zwischen Objekten innerhalb und außerhalb des Systems minimiert wird. Objekte, zu denen keine oder nur sehr geringe Interaktion stattfindet, gehören nach dieser Definition nicht zum System. Bei den Verfeinerungsschritten wird sich jeweils zeigen, dass man Relationen zu Objekten benötigt, die bisher noch nicht Teil des Systems waren, in der der nachfolgenden Ebene muss dann entschieden werden, ob die Interaktion zu diesen Objekten groß genug ist, um sie als Teil des Systems aufgenommen werden, oder ob sie als Teil der Umgebung zu be-

trachten sind.

Eine weitere offensichtliche Systemabgrenzung bei Eisenbahnen ist die soziale Abgrenzung („social" boundary), mit der Unterscheidung, wer welches System hergestellt hat oder betreut. Wichtigste Komponenten sind hier z. B. der Eigentümer des Streckennetzes, der Betreiber der Strecke, der Hersteller der Fahrzeuge, der Betreiber der Fahrzeuge. Diese Unterscheidung ist bei der Ontological Hazard Analysis jedoch nicht relevant, und die entstehende Systemspezifikation wird eine flussbasierte Begrenzung haben.

Kapitel 3

Ebene 0 („UrSpec“)

3.1 Sorten (*sorts*)

Auf der ersten Ebene betrachten wir sehr verallgemeinerte Objekte, siehe Tabelle 3.1. Das Ziel der ersten Stufe kann und soll nicht sein, den Betrieb in Details zu beschreiben, sondern, möglichst einfach zu sein, so dass sich Sicherheitsanforderungen formulieren lassen, die *offensichtlich* richtig sind. Diese Beschreibung mag trivial erscheinen, aber es ist notwendig, dass diese Beschreibung so einfach ist, dass es keinen Zweifel geben kann, dass sie sowohl korrekt als auch vollständig ist.

Es wurde die Sorte *Fahrzeug* gewählt, anstelle der möglichen Alternativen *Zug* bzw. *Fahrt*, die eine bestimmte Zugfahrt (d. h. eine Fahrt nach Fahrplan) implizieren. Dies wäre nicht vollständig, da Schienenfahrzeuge auch anders bewegt werden als in planmäßigen Zugfahrten. Die Definition als generisches *Fahrzeug* schließt also Rangierfahrten und Sperrfahrten ein.

Denkbar gewesen wäre auch eine Eigenschaft $frei(S)$ nur als Prädikat eines Streckenabschnitts zu definieren, jedoch würde dies zu kurz greifen, da ein Streckenabschnitt möglicherweise für einen Zug befahrbar ist, aber für einen anderen Zug nicht, beispielsweise eine nicht elektrifizierte Strecke mit einem Elektrotriebfahrzeug.

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

Tabelle 3.1: Sorten auf Ebene 0 („Level 0“)

3.2 Relationen

3.2.1 Unäre Relationen

Zur Beschreibung auf dieser Stufe sind keine unären Relationen notwendig.

3.2.2 Binäre Relationen

Die Tabelle in Tabelle 3.2 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 3.2: Binäre Relationen auf Ebene 0

3.3 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 $\mathrm{LV2}$, $\mathrm{in2}$ und $\mathrm{ZV2}$ für einen zweiten Zug $F2$. Hierdurch werden außerdem die Prädikate in einfache Aussagenvariablen umgewandelt, so dass sich die nachfolgenden Umformungen und Folgerungen nach den Regeln der Aussagenlogik durchführen lassen. Dies ist möglich, da die Prädikate jeweils nur für feste Variablen $F1$, $F2$, S betrachetet werden, und nicht quantifiziert ($\forall$, $\exists$) sind.

3.3.1 Ein Zug

Für einen festen Streckenabschnitt S ergeben sich so drei atomare Aussagen über einen bestimmten Zug $F1$, nämlich $\mathrm{LV1}$, $\mathrm{in1}$, $\mathrm{ZV1}$. 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 $\mathrm{TF}_n = \text{wahr}$.

Um alle diese möglichen Sicherheitsaxiome zu untersuchen, und damit alle tatsächlichen zu erhalten, benutzt man den Satz, dass es zu allen aussagenlogischen Formeln eine logisch äquivalente Formel (eine Formel, die dieselbe Wahrheitsfunktion beschreibt) in konjunktiver Normalform (KNF) gibt, das heißt, in der Form:

$$(L_{11} \vee L_{12} \vee \cdots) \wedge (L_{21} \vee L_{22} \vee \cdots) \wedge \cdots$$

wobei L_{ij} ein Literal ist, also eine atomare Aussage oder die Negation einer atomaren Aussage.

Wenn man nun nach Axiomen (Prinzipien) sucht, und $(A \wedge B)$ ein Axiom ist, das man zusichern will, so kann man dies als zwei getrennte Axiome, A und B, zusichern. Daher braucht man nur die nicht-konjunktiven Aussagen in der Form $(L_{11} \vee L_{12} \vee \cdots)$ zu betrachten, um die Axiome abzuleiten.

Aussagen der Form $(L_{11} \vee L_{12} \vee \cdots L_{1n})$ sind logisch äquivalent zu Aussagen der Form

$$(\neg L_{11} \wedge \neg L_{12} \wedge \cdots \neg L_{1x} \Rightarrow L_{1,x+1} \vee L_{1,x+2} \vee \cdots L_{1n})$$

Auch hier ist eine Formel der Form $\neg L_{1k}$ äquivalent zu zu einem Literal, denn falls L_{1k} eine atomare Aussage ist, ist $\neg L_{1k}$ per Definition auch ein Literal, und falls L_{1k} gleich $\neg A_{1k}$ ist, wobei A_{1k} eine atomare Aussage ist, dann ist $\neg L_{1k}$ logisch äquivalent zu A_{1k}, und A_{1k} ist per Definition eine atomare Aussage. Wählt man $x = (n-1)$ müssen also nur noch Formeln der Form

$$L_{11} \wedge L_{12} \wedge \cdots L_{1,n-1} \Rightarrow L_{1n}$$

betrachtet werden.

Zunächst betrachtet man nur Formeln dieser Art mit $\mathrm{LV1}$, $\mathrm{in1}$ und $\mathrm{ZV1}$ als atomaren Aussagen. Zunächst mit nur jeweils einer atomaren Aussage links, danach mit zwei atomaren Aussagen links.

Lemma 1. *Alle Formeln mit drei atomaren Aussagen links sind Tautologien, Widersprüche oder Zusicherung des Konsequens*

Als erstes ist festzustellen, dass keine Formel ohne atomare Aussagen auf der linken Seite ein Sicherheitsaxiom ist. Diese Formeln sichern nur die jeweils rechts allein stehende atomare Aussage zu, und es ist leicht zu sehen, dass keine davon im Zugleitbetrieb stets wahr ist, geschweige denn ein Sicherheitsaxiom darstellt.

Aussagen mit einer atomaren Aussage links. Man muss für alle atomaren Aussagen, die links nicht auftreten, und deren Negation, überprüfen, ob die gesamte Aussage ein Sicherheitsaxiom darstellt.

$\mathrm{LV1} \Rightarrow ??$	Keine Sicherheitsaxiome, wie die Überprüfung gegen $\mathrm{ZV1}$ $\neg\,\mathrm{ZV1}$, $\mathrm{in1}$ und $\neg\,\mathrm{in1}$ zeigt.
$\mathrm{in1} \Rightarrow ??$	Ebenfalls keine Sicherheitsaxiome, entsprechend wie oben

$$\mathrm{ZV1} \Rightarrow \neg\,\mathrm{LV1} \qquad \text{(S0.a)}$$

Als nächstes sind Formeln mit zwei atomaren Aussagen auf der linken Seite zu betrachten:

$\mathrm{LV1} \wedge \mathrm{in1} \Rightarrow \neg\,\mathrm{ZV1}$		(S0.b)
$\mathrm{LV1} \wedge \neg\,\mathrm{in1} \Rightarrow ??$	Kein Sicherheitsaxiom	
$\neg\,\mathrm{LV1} \wedge \mathrm{in1} \Rightarrow \mathrm{ZV1}$		(S0.c)
$\neg\,\mathrm{LV1} \wedge \neg\,\mathrm{in1} \Rightarrow ??$	Kein Sicherheitsaxiom	

$\mathrm{LV1} \wedge \mathrm{ZV1}$	Widerspruch zu (S0.a)
$\mathrm{LV1} \wedge \neg\,\mathrm{ZV1} \Rightarrow ??$	Kein Sicherheitsaxiom
$\neg\,\mathrm{LV1} \wedge \mathrm{ZV1} \Rightarrow ??$	Kein Sicherheitsaxiom
$\neg\,\mathrm{LV1} \wedge \neg\,\mathrm{ZV1} \Rightarrow ??$	Kein Sicherheitsaxiom

---

$\mathrm{in1} \wedge \mathrm{ZV1} \Rightarrow \neg\,\mathrm{LV1}$		(S0.d)
$\mathrm{in1} \wedge \neg\,\mathrm{ZV1} \Rightarrow \mathrm{LV1}$		(S0.e)
$\neg\,\mathrm{in1} \wedge \mathrm{ZV1} \Rightarrow ??$	Kein Sicherheitsaxiom	
$\neg\,\mathrm{in1} \wedge \neg\,\mathrm{ZV1} \Rightarrow ??$	Kein Sicherheitsaxiom	

Das Axiom (S0.d) ist logisch äquivalent zu (S0.b), und (S0.e) ist logisch äquivalent zu (S0.c).

Also ergeben sich für einen Zug die folgenden drei Sicherheits-Axiome:

$$\mathrm{ZV1} \Rightarrow \neg\,\mathrm{LV1} \quad \text{(S0.1)}$$

$$\neg\,\mathrm{LV1} \wedge \mathrm{in1} \Rightarrow \mathrm{ZV1} \quad \text{(S0.2)}$$

$$\mathrm{in1} \wedge \mathrm{ZV1} \Rightarrow \neg\,\mathrm{LV1} \quad \text{(S0.3)}$$

3.3.2 Zwei Züge

Für die Betrachtung von 2 Zügen ergeben sich sechs atomare Aussagen: LV1, LV2, in1, in2, ZV1, ZV2. Die systematische Vorgehensweise ist wie bei einem einzelnen Zug, das heißt, von den Formeln der Form

$$L_{11} \wedge L_{12} \wedge \cdots \Rightarrow L_{1n}$$

werden zunächst diejenigen betrachtet, die nur eine atomare Aussage auf der linken Seite haben, dann jene mit 2, und so fort. Dabei werden nur „gemischte“ Formeln betrachtet, die sowohl atomare Aussagen über Zug 1 als auch über Zug 2 enthalten. Es muss außerdem sichergestellt sein, dass die sich jeweils entsprechenden Aussagen zu Zug 1 und Zug 2 verschieden sind, d. h. jeder der folgenden Formeln muss ein

$$F1 \neq F2 \Rightarrow$$

vorangestellt sein.

Für Formeln mit einer atomaren Aussage links ergeben sich vier neue Sicherheitsaxiome. (Es gibt jeweils auch den symmetrischen Fall, bei dem Zug 1 und 2 vertauscht werden, aber da die Variablen frei sind, stellen jeweils beide dieselbe Bedingung dar.)

Für Formeln mit einer atomaren Aussage links ergeben sich folgende neue Sicherheits-Axiome:

$$\mathrm{LV1} \Rightarrow \neg\,\mathrm{ZV2} \tag{S0.f}$$

$$\mathrm{in1} \Rightarrow \neg\,\mathrm{ZV2} \tag{S0.g}$$

$$\mathrm{ZV1} \Rightarrow \neg\,\mathrm{ZV2} \tag{S0.h}$$

$$\mathrm{ZV1} \Rightarrow \neg\,\mathrm{in2} \tag{S0.i}$$

Hier ist das symmetrische Gegenstück zu (S0.g) ($\mathrm{in2} \Rightarrow \neg\,\mathrm{ZV1}$) logisch äquivalent zu (S0.i).

Für zwei atomare Aussagen links ergibt sich nur ein neues Sicherheits-Axiom, das sich nicht bereits durch die vorherigen Betrachtungen ergeben hätte:

$$\mathrm{in1} \wedge \mathrm{in2} \Rightarrow \mathrm{LV1} \tag{S0.j}$$

Es stellt sich allerdings heraus, dass dieses Axiom aus (S0.2) und (S0.g) hergeleitet werden kann, wie folgt: Sei $(F1 \neq F2) \wedge \mathrm{in1} \wedge \mathrm{in2}$, daraus folgt $\neg\,\mathrm{ZV2}$ nach (S0.g). Die Kontraposition von (S0.2) für $F2$ ist:

$$\neg\,\mathrm{ZV2} \Rightarrow \neg(\neg\,\mathrm{LV2} \wedge \mathrm{in2})$$

Durch die DeMorgansche Regeln und die Regel der doppelten Negation ergibt dies:

$$\neg\,\mathrm{ZV2} \Rightarrow \neg\,\mathrm{in2} \vee \mathrm{LV2}\,.$$

Durch Modus Ponens folgt: $\neg\,\mathrm{in2} \vee \mathrm{LV2}$ und dies ist logisch äquivalent zu $\mathrm{in2} \Rightarrow \mathrm{LV2}$ woraus (Modus Ponens) folgt:

$$\mathrm{LV2}\,.$$

Die Betrachtungen zu drei, vier und fünf atomaren Aussagen links ergeben keine neuen Sicherheits-Axiome.

3.3.3 Mehr als zwei Züge

Die Fahrdienstvorschrift für nichtbundeseigene Eisenbahnen beschreibt ausdrücklich nur die Betriebsverfahren für eingleisige Strecken. Für zweigleisige Strecken werden im Einzelfalle spezielle Regeln festgelegt [FVN, §1(3)], dies soll daher im Rahmen dieser Arbeit nicht betrachtet werden.

Da eine eingleisige Strecke für Züge eindimensional ist, kann es immer nur zwischen maximal zwei Zügen einen gemeinsamen Interaktionspunkt geben. Also kann man alle gefährlichen Interaktionen mit mehr als zwei Zügen immer auch auf gefährliche Sub-Interaktionen mit zwei Zügen zurückführen.

Damit ist durch vollständige Aufzählung garantiert, dass alle Sicherheits-Axiome für diese Verfeinerungsebene betrachtet wurden. Insgesamt sind diese in Tabelle 3.3 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 3.3: Sicherheitsaxiome der Ebene 0

3.4 Verfeinerung (*Refinement*)

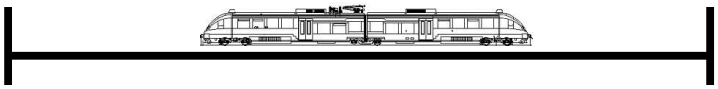
Die Ontological Hazard Analysis schlägt als Methode zur Festlegung, welche Objekte wie verfeinert werden sollen, HAZOP[HAZ01] vor. Damit sollen Abweichungen vom geplanten Betrieb festgestellt werden, und bei denjenigen Abweichungen, die sich nicht in der aktuellen Ontologie ausdrücken lassen, diese um 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*, [FVN]) 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.

Kapitel 4

Ebene 1



4.1 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 4.1 zeigt eine erweiterte Liste der Sorten.

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 4.1: Sorten auf Ebene 1 („Level 1“)

Es gibt keine zusätzlichen Eigenschaften der Objekte auf Ebene 1, siehe Tabelle 4.2.

Tabelle 4.3 zeigt die binären, und Tabelle 4.4 die ternären Relationen der Ebene 1, neu sind hier entsprechende Relationen zwischen Fahrzeugen

Eigenschaft	(informelle) Beschreibung
EISENBAHNFAHRZEUG	
—	
STRECKENABSCHNITT	
—	
ZUGLAUFMELDESTELLE	
—	
GLEIS IM BAHNHOF	
—	

Tabelle 4.2: Eigenschaften der Objekte auf Ebene 1

und Zuglaufmeldestellen bzw. Gleisen.

4.1.1 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.

Da in diesem Falle nur die Sorte *Strecke* verfeinert wurde, ist dies einfach zu zeigen. dabei sind $\mathrm{Streckenabschnitt}(x)$, $\mathrm{Zuglaufmeldestelle}(x)$ und $\mathrm{Gleis}(x)$ die Sorten-Prädikate, die die Zugehörigkeit des Objektes zur jeweiligen Sorte (engl. *sort*) bedeuten.

Die Schreibweise ist die von Lamport in [Lam94] vorgeschlagene übersichtliche Anordnung von Konjunktionen und Disjunktionen mit der Verwendung des Junktionszeichens als Präfix-Operator.

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
$\mathrm{LV}(F, S)$	Fahrzeug F darf den Streckenabschnitt S unter lokaler Verantwortung belegen
EISENBAHNFAHRZEUG, ZUGLAUFMELDESTELLE	
$\mathrm{inZ}(F, A)$	Fahrzeug F befindet sich in der Zuglaufmeldestelle A
EISENBAHNFAHRZEUG, GLEIS	
$\mathrm{inG}(F, G)$	Fahrzeug F befindet sich auf Gleis G
$\mathrm{ZG}(F, G)$	Fahrzeug F darf Gleis G unter zentraler Verantwortung belegen

Tabelle 4.3: Binäre Relationen auf Ebene 1

Eigenschaft	(informelle) Beschreibung
STRECKENABSCHNITT, 2$\times$ ZUGLAUFMELDESTELLE	
$\mathrm{begr}(S, A, B)$	Zwei Zuglaufmeldestellen A und B begrenzen den Streckenabschnitt S
EISENBAHNFAHRZEUG, 2$\times$ 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 4.4: Ternäre Relationen auf Ebene 1

$$\text{Streckenabschnitt}(S) \Leftrightarrow \begin{cases} \vee \wedge \exists A, B \\ \quad \wedge \text{Zuglaufmeldestelle}(A) \\ \quad \wedge \text{Zuglaufmeldestelle}(B) \\ \quad \wedge \text{begr}(S, A.B) \\ \vee \text{Gleis}(S) \end{cases} \tag{M1.1}$$

$$\left.\begin{array}{l} \wedge \text{Streckenabschnitt}(S) \\ \wedge \text{Fahrzeug}(F) \\ \wedge \text{inA}(F, S) \end{array}\right\} \Leftrightarrow \begin{cases} \vee \wedge \exists A, B \\ \quad \wedge \text{Zuglaufmeldestelle}(A) \\ \quad \wedge \text{Zuglaufmeldestelle}(B) \\ \quad \wedge \text{begr}(S, A.B) \\ \quad \wedge \text{zw}(F, A, B) \\ \vee \wedge \text{Gleis}(S) \\ \quad \wedge \text{inG}(F, S) \end{cases} \tag{M1.2}$$

$$\left.\begin{array}{l} \wedge \text{Streckenabschnitt}(S) \\ \wedge \text{Fahrzeug}(F) \\ \wedge \text{ZV}(F, S) \end{array}\right\} \Leftrightarrow \begin{cases} \vee \wedge \exists A, B \\ \quad \wedge \text{Zuglaufmeldestelle}(A) \\ \quad \wedge \text{Zuglaufmeldestelle}(B). \\ \quad \wedge \text{begr}(S, A.B) \\ \quad \wedge \text{ZZ}(F, A, B) \\ \vee \wedge \ \mathit{Gleis}(S) \\ \quad \wedge \text{ZG}(F, S) \end{cases} \tag{M1.3}$$

4.2 Sicherheitsaxiome

Da gegenüber der Beschreibung auf Ebene 0 alle bis dahin existierenden Elemente beibehalten wurden, bleiben alle dort gefundenen Sicherheitsaxiome gültig. Die zusätzlichen Axiome ergeben sich direkt aus einer jeweiligen Ersetzung der Aussagen über einen Streckenabschnitt durch ihre Entsprechungen für Zuglaufmeldestellen und Gleise (im Bahnhof).

4.2.1 Neue Aussagen

Analog zum Verfahren auf Ebene 0 werden folgende Aussagen für zwei Züge $F1$, $F2$, zwei Zuglaufmeldestellen A, B und ein Gleis G definiert:

$$\begin{aligned}
\mathrm{ZW1} &= \mathrm{zw}(F1, A, B) \\
\mathrm{ZW2} &= \mathrm{zw}(F2, A, B) \\
\mathrm{G1} &= \mathrm{inG}(F1, G) \\
\mathrm{G2} &= \mathrm{inG}(F2, G) \\
\mathrm{ZZ1} &= \mathrm{ZZ}(F1, A, B) \\
\mathrm{ZZ2} &= \mathrm{ZZ}(F2, A, B) \\
\mathrm{ZG1} &= \mathrm{ZG}(F1, G) \\
\mathrm{ZG2} &= \mathrm{ZG}(F2, G)
\end{aligned}$$

Für alle Axiome der Ebene 0, in denen Aussagen über einen Streckenabschnitt vorkommen, erhält man jeweils zwei neue, je nachdem, ob es sich um einen Abschnitt der freien Strecke zwischen zwei Zuglaufmeldestellen oder um ein Gleis im Bahnhof handelt.

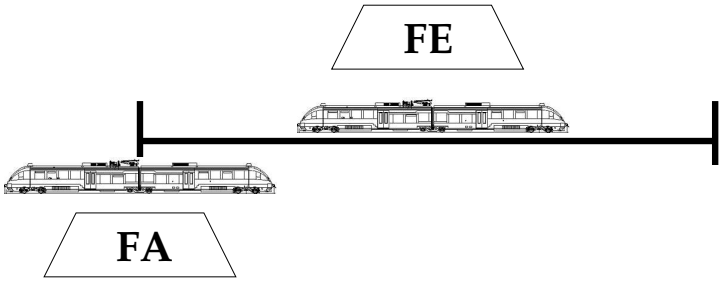
Damit ergeben sich in Abbildung 4.1 die neuen Axiome.

$$\mathrm{ZZ1} \Rightarrow \neg\,\mathrm{LV1} \tag{S1.1}$$
$$\mathrm{ZG1} \Rightarrow \neg\,\mathrm{LV1} \tag{S1.2}$$
$$\neg\,\mathrm{LV1} \wedge \mathrm{ZW1} \Rightarrow \mathrm{ZZ1} \tag{S1.3}$$
$$\neg\,\mathrm{LV1} \wedge \mathrm{G1} \Rightarrow \mathrm{ZG1} \tag{S1.4}$$
$$\mathrm{ZW1} \wedge \mathrm{ZZ1} \Rightarrow \neg\,\mathrm{LV1} \tag{S1.5}$$
$$\mathrm{G1} \wedge \mathrm{ZG1} \Rightarrow \neg\,\mathrm{LV1} \tag{S1.6}$$
$$(F1 \neq F2) \Rightarrow (\mathrm{LV1} \Rightarrow \neg\,\mathrm{ZZ2}) \tag{S1.7}$$
$$(F1 \neq F2) \Rightarrow (\mathrm{LV1} \Rightarrow \neg\,\mathrm{ZG2}) \tag{S1.8}$$
$$(F1 \neq F2) \Rightarrow (\mathrm{ZW1} \Rightarrow \neg\,\mathrm{ZZ2}) \tag{S1.9}$$
$$(F1 \neq F2) \Rightarrow (\mathrm{G1} \Rightarrow \neg\,\mathrm{ZG2}) \tag{S1.10}$$
$$(F1 \neq F2) \Rightarrow (\mathrm{ZZ1} \Rightarrow \neg\,\mathrm{ZZ2}) \tag{S1.11}$$
$$(F1 \neq F2) \Rightarrow (\mathrm{ZG1} \Rightarrow \neg\,\mathrm{ZG2}) \tag{S1.12}$$

Abbildung 4.1: Zusätzliche Sicherheitsaxiome auf Ebene 1

Kapitel 5

Ebene 2



5.1 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 5.1

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 5.1: Sorten auf Ebene 2

5.2 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, s. Tabelle 5.2.

Eigenschaft	(informelle) Beschreibung
EISENBAHNFAHRZEUG	
—	
STRECKENABSCHNITT	
—	
ZUGLAUFMELDESTELLE	
—	
GLEIS IM BAHNHOF	
—	

Tabelle 5.2: Eigenschaften der Objekte auf Ebene 2

Neben diesen unären Relationen gibt es auch binäre Relationen. Zur Beschreibung des Zugleitbetriebs auf dieser Ebene, sind die in Tabelle 5.3 gezeigten Relationen erforderlich.

Es gibt einige Relationen zwischen drei Objekten, siehe Tabelle 5.4.

5.3 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 5.1.

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
$\mathrm{LV}(F, S)$	Fahrzeug F darf den Streckenabschnitt S unter lokaler Verantwortung belegen
EISENBAHNFAHRZEUG, ZUGLAUFMELDESTELLE	
$\mathrm{inZ}(F, A)$	Fahrzeug F befindet sich in der Zuglaufmeldestelle A
FAHRZEUG, ZUGLAUFMELDESTELLE	
$\mathrm{Next}(F, A)$	Diese Funktion bezeichnet die nächste fahrplanmäßige Zuglaufmeldestelle für Fahrzeug F nach der Zuglaufmeldestelle A

Tabelle 5.3: Binäre Relationen auf Ebene 2

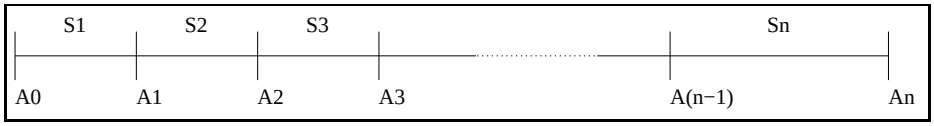


Abbildung 5.1: Schematische Zugleitstrecke

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 5.4: Ternäre Relationen auf Ebene 2

5.3.1 Als Pseudo-Code

Dies lässt sich in der Darstellung als Pseudo-Code als Schleife darstellen, mit einem Durchlauf pro Abschnitt zwischen jeweils zwei Zuglaufmeldestellen (s. Abbildung 5.2.

```
FOR ( i = 1 TO N )
⋆     Zug F in A_{i-1}
      Fahranfrage-F-A_i
Bed: IF    ( Kein-Hindernis(A_{i-1}, A_i) ) THEN
           Fahrerlaubnis-F-A_i
           F fährt von A_{i-1} nach A_i
      ELSE
           Verweigerung-Fahrerlaubnis-F-A_i
           GOTO Bed
      ENDIF
ENDFOR
⋆ bedeutet, dass diese Bedingung bei jedem Schleifendurchlauf vorausgesetzt wird.
```

Abbildung 5.2: Pseudo-Code

5.3.2 Als Zustandsautomat (Predicate Action Diagram)

Der Pseudocode des auf der einfachsten Ebene beschriebenen Verfahrens für eine Fahrt im Zugleitbetrieb lässt sich auch als Zustandsmaschine darstellen, siehe Abbildung 5.3.

Die Zustände sind in Tabelle 5.5 exakt beschrieben, hier folgt eine informelle Beschreibung in natürlicher Sprache:

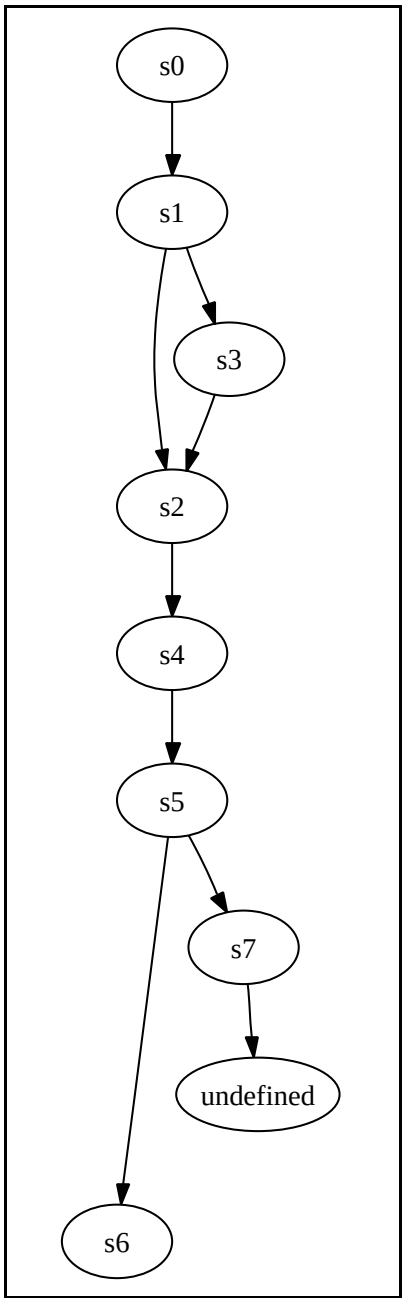


Abbildung 5.3: Zustandsmaschine

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

Tabelle 5.5: Zustände des Automaten in Abb. 5.3

- 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.

Diese Beschreibungen (bzw. deren formale Entsprechung in Abbildung 5.5) sind die Axiome zur Durchführung des normalen Betriebes bei einer Fahrt von einer Zuglaufmeldestelle zur nächsten.

Sie kann umgesetzt werden in die Formulierung eines Zustandsautomaten in PROMELA, der Beschreibungssprache des Model-Checkers SPIN. Dies stellt eine ausführbare Semantik dar, die mit Hilfe des Model Checkers verifiziert werden kann.

Ebenso kann diese Beschreibung nach Lamport [Lam95] direkt in eine formal strenge Beschreibung in der Temporal Logic of Actions (TLA) übersetzt werden.

5.3.3 In PROMELA

PROMELA (*PROtocol/PROcess MEta LAnguage*) ist die Beschreibungssprache des Model-Checkers SPIN (*Simple Promela INterpreter*)

Model Checkers. In PROMELA kann das Verhalten von Systemen abstrakt beschrieben werden, und bestimmte Eigenschaften des Systems, z. B. Safety-Requirements, können überprüft werden.

Abbildung 5.4 zeigt die Beschreibung einer Zugfahrt im Zugleitbetrieb in der Sprache des Model-Checkers SPIN. Obwohl es Ähnlichkeiten mit einer eine Programmiersprache hat, beschreibt diese Sprache Zustandsautomaten; der Verifier von SPIN kann diese Automaten effizient überprüfen.

5.3.4 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.

5.3.5 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.

5.4 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.

```
 1: /*
 2:    Beschreibung einer Zugfahrt ueber eine Zugleitstrecke,
 3:    mit mehrere Zuglaufmeldestellen.
 4: */
 5:
 6: /* Anzahl der Zuglaufmeldestellen der betrachteten Zugfahrt */
 7: #define Anzahl_ZMS 6
 8:
 9: byte Naechste_ZMS;
10: byte Abschnitte = Anzahl_ZMS - 1;
11: bool Fahranfrage[Anzahl_ZMS] = false;
12: bool Fahrerlaubnis[Anzahl_ZMS] = false;
13: bool Kein_Hindernis[Anzahl_ZMS];
14: bool inA[Anzahl_ZMS] = false;
15: bool inZ[Anzahl_ZMS] = false;
16:
17: active proctype Zugfahrt () {
18:     Naechste_ZMS = 0;
19:     inZ[Naechste_ZMS] = true;
20:     do
21:     ::
22:         if
23:         :: ( Naechste_ZMS < ( Abschnitte ) ) ->
24:             Naechste_ZMS++;
25:         :: else ->
26:             break;
27:         fi;
28:         Fahrerlaubnis[Naechste_ZMS] = false;
29:         Fahranfrage[Naechste_ZMS] = true;;
30: Bed:    if
31:         :: Kein_Hindernis[Naechste_ZMS] = true;
32:         :: Kein_Hindernis[Naechste_ZMS] = false;
33:         fi;
34:         if
35:         :: ( Kein_Hindernis[Naechste_ZMS] ) ->
36:             d_step {
37:                 Fahrerlaubnis[Naechste_ZMS] = true;
38:                 Fahranfrage[Naechste_ZMS] = false;
39:             }
40:             d_step {
41:                 inZ[Naechste_ZMS - 1] = false;
42:                 inA[Naechste_ZMS] = true;
43:             }
44:             d_step {
45:                 inA[Naechste_ZMS] = false;
46:                 inZ[Naechste_ZMS] = true;
47:                 Fahrerlaubnis[Naechste_ZMS] = false;
48:             }
49:         :: else ->
50:             Fahrerlaubnis[Naechste_ZMS] = false;
51:             goto Bed;
52:         fi;
53:     od;
54: }
```

Abbildung 5.4: Spezifikation einer Zugfahrt in PROMELA

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 5.5 dargestellt.

5.5 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:

Beweis von (S0.1). $\mathrm{ZV1}$ impliziert $\mathrm{KH1}$ (nach (M2.1)), $\mathrm{KH1}$ 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)). □

Beweis von (S0.3). $(in1 \wedge \mathrm{ZV1})$ impliziert $(\mathrm{KH1} \wedge \mathrm{FE1})$ (M2.1). $(\mathrm{KH1} \wedge \mathrm{FE1})$ impliziert, dass sich F1 in Zustand s5 befindet (nach Definition des Zustandsautomaten). Zustand s5 impliziert $\neg\,\mathrm{LV1}$ (nach Definition des Zustandsautomaten). □

Beweis von (S0.4). $((F1 \neq F2) \wedge \neg\,\mathrm{LV1})$ impliziert, dass sich dass sich F1 in Zustand s5 befindet (nach Definition des Zustandsautomaten). Zustand s5 impliziert $(\mathrm{FE1} \wedge \mathrm{KH1})$ (nach Definition des Zustandsautomaten). $\mathrm{FE1}$ impliziert $\neg\,\mathrm{FE2}$ (nach (S2.2)). $\neg\,\mathrm{FE2}$ impliziert $\neg\,\mathrm{ZV2}$ (nach (M2.2)). □

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

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

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

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 ($\Box$: 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 5.5: Sicherheitsaxiome auf Ebene 2

Beweis von (S0.5). $((F1 \neq F2) \wedge \mathrm{in1})$ impliziert, dass sich dass sich F1 in Zustand s5 befindet (nach Definition des Zustandsautomaten). Zustand s5 impliziert $(\mathrm{FE1} \wedge \mathrm{KH1})$ (nach Definition des Zustandsautomaten). $\mathrm{FE1}$ im-

pliziert $\neg$ FE2 (nach (S2.2)). $\neg$ FE2 impliziert $\neg$ ZV2 (nach (M2.2)). □

Beweis von (S0.6). $((F1 \neq F2) \wedge \mathrm{ZV1})$ impliziert FE1 (nach (M2.2)). FE1 impliziert $\neg$ FE2 (nach (S2.2)). $\neg$ FE2 impliziert $\neg$ ZV2 (nach Kontraposition von (M2.2)). □

5.6 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:

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

Kapitel 6

HAZOP

Zur systematischen Identifizierung der Hazards wird das HAZOP-Verfahren (HAZard and Operability studies) verwendet. Die damit gefundenen Abweichungen („Deviations“) vom vorgesehenen Verhalten können verwendet werden, um Liveness- und Safety-Requirements für den Zugleitbetrieb zu formulieren, und mit Hilfe eines Model Checkers zu überprüfen.

Bei diesem Verfahren werden alle Parameter des untersuchten Verfahrens mit allen Guide-Words aus einem vorher festgelegten Set kombiniert, und überprüft, ob sie eine plausible Abweichung vom gewünschten Systemverhalten beschreiben, und ob diese Abweichung einen Hazard darstellt, oder eine Einschränkung des normalen und effizienten Betriebes.

6.1 Guide Words

Als Satz von Guide-Words bieten sich die in der Norm IEC 61882 [HAZ01] vorgeschlagenen an:

Guide Word	allg. Bedeutung	Bedeutung im Bahnbetrieb
Basic Guide Words		
NO or NOT	Vollständige Negierung des vorgesehenen Funktion	Zugfahrt findet nicht statt oder Nachricht wird nicht übermittelt
MORE	Quantitative Steigerung	Längerer Zug, längere Nachricht
LESS	Quantitative Verminderung	Kürzerer Zug, kürzere Nachricht
AS WELL AS	Qualitative Änderung / Steigerung	Ein zusätzlicher Zug, eine zusätzliche Nachricht
PART OF	Qualitative Änderung / Verminderung	Nur ein Teil des Zuges / der Nachricht
REVERSE	Logisches Gegenteil der vorgesehenen Funktion	Zug fährt in Gegenrichtung, Gegenteilige Nachricht
OTHER THAN	Vollständige Ersetzung	Ein anderer Zug, eine andere Nachricht
Zusätzliche Guide-Words zu Zeit und Reihenfolge		
EARLY	Früher bzgl. absoluter Uhrzeit	Zug fährt zu früh ab, Nachricht wird zu früh übermittelt
LATE	Später bzgl. absoluter Uhrzeit	Zug fährt zu spät ab, Nachricht wird zu spät übermittelt
BEFORE	Früher bzgl. der Reihenfolge	Falsche Zugreihenfolge, falsche Reihenfolge bei Nachrichten
AFTER	Später bzgl. der Reihenfolge	Falsche Zugreihenfolge, falsche Reihenfolge bei Nachrichten

6.2 Anwendung der HAZOP-Guide Words auf die ZLB-Parameter

In der Beschreibung des HAZOP-Verfahrens in der IEC 61882 sind zwei grundsätzlich verschiedene Vorgehensweisen vorgeschlagen: entweder kombiniert man nacheinander alle Guide-Words jeweils mit demselben Design-Parameter, und geht erst dann zum nächsten Design-Parameter über, oder man kombiniert immer alle Design-Parameter mit demselben Guide-Word, und tut anschließend dasselbe mit dem nächsten Guide-Word.

Hier erscheint der Ansatz, jeweils mit jedem Parameter jedes Guide-Word durchzugehen, angemessen. Einer der ersten Schritte nach der Zusammenstellung der Guide Words und Parameter, ist, festzustellen, ob zwei verschiedene dieser Kombinationen dasselbe Probleme beschreiben.

In einzelnen Tabellen folgen die einzelnen Deviations (Abweichungen) durch Zusammenstellung der Relationen und Guide Words. Für Zusammenstellungen, die keinen Sinn ergeben, steht in der Beschreibung ein „—“. In der Spalte „H/O“ stehen jeweils ein Häkchen oder ein Strich, je nachdem, ob für den beschriebenen Fall eine Gefährdung (Hazard) oder eine Einschränkung des Betriebsablaufs (Operability Problem) vorliegt.

In der Spalte „beschr.“ steht jeweils ein Häkchen, wenn die jeweilige Abweichung im aktuellen System beschrieben werden kann, ein Strich, wenn es nicht relevant ist, da die Kombination keine sinnvolle Abweichung beschreibt, und ein Kreuz, wenn sich eine Abweichung nicht beschreiben lässt.

6.2.1 LV(F, S)

Nr.	Guide Word	Spezifische Bedeutung	H/O	beschr.
1.a	NOT	Zug fährt nicht auf Sicht	✓/✓	✓
1.b	MORE	—	-/-	-
1.c	LESS	—	-/-	-
1.d	AS WELL AS	Ein weiterer Zug fährt auf Sicht	-/✓	×
1.e	PART OF	—	-/-	-
1.f	REVERSE	Zug fährt nicht auf Sicht	✓/✓	✓
1.g	OTHER THAN	Ein anderer Zug als vorgesehen fährt auf Sicht	✓/✓	
1.h	EARLY	Zug fährt eher als geplant auf Sicht	-/✓	×
1.i	LATE	Zug fährt erst später als geplant auf Sicht	✓/✓	×
1.j	BEFORE	Zug fährt fälschlicherweise eher als ein anderer auf Sicht	-/✓	×
1.k	AFTER	Zug fährt fälschlicherweise später als ein anderer auf Sicht	-/✓	×

6.2.2 $\mathrm{inA}(F, S)$

Nr.	Guide Word	Spezifische Bedeutung	H/O	beschr.
2.a	NOT	Zug ist nicht im Abschnitt	-/✓	✓
2.b	MORE	—	-/-	-
2.c	LESS	—	-/-	-
2.d	AS WELL AS	Ein weiterer Zug ist im Abschnitt	✓/✓	×
2.e	PART OF	Nur Teil des Zuges ist im Abschnitt	✓/✓	✓
2.f	REVERSE	Zug fährt in falscher Richtung im Abschnitt	✓/✓	×
2.g	OTHER THAN	Ein anderer Zug als vorgesehen fährt in den Abschnitt	✓/✓	×
2.h	EARLY	Zug fährt eher als geplant in den Abschnitt	✓/✓	✓
2.i	LATE	Zug fährt später als geplant in den Abschnitt	-/✓	✓
2.j	BEFORE	Zug fährt fälschlicherweise eher als ein anderer in den Abschnitt	✓/✓	×
2.k	AFTER	Zug fährt fälschlicherweise eher als ein anderer in den Abschnitt	✓/✓	×

6.2.3 ZV(F, S)

Nr.	Guide Word	Spezifische Bedeutung	H/O	beschr.
3.a	NOT	Der Fahrweg ist nicht frei	✓/✓	✓
3.b	MORE	—	-/-	-
3.c	LESS	—	-/-	-
3.d	AS WELL AS	—	-/-	-
3.e	PART OF	—	-/-	-
3.f	REVERSE	Der Fahrweg ist nicht frei	✓/✓	✓
3.g	OTHER THAN	—	-/-	-
3.h	EARLY	Abschnitt ist früher frei als geplant	-/-	✓
3.i	LATE	Abschnitt ist später frei als geplant	-/✓	✓
3.j	BEFORE	—	-/-	-
3.k	AFTER	—	-/-	-

6.2.4 $\mathrm{inZ}(F, A)$

Nr.	Guide Word	Spezifische Bedeutung	H/O	beschr.
4.a	NOT	Zug ist nicht auf der vorgesehenen Zuglaufmeldestelle	-/✓	✓
4.b	MORE	—	-/-	-
4.c	LESS	—	-/-	-
4.d	AS WELL AS	Ein weiterer Zug ist auf der Zuglaufmeldestelle	-/✓	×
4.e	PART OF	Nur ein Teil des Zuges ist auf der Zuglaufmeldestelle	✓/✓	✓
4.f	REVERSE	Zug ist nicht auf der vorgesehenen Zuglaufmeldestelle	-/✓	✓
4.g	OTHER THAN	Ein anderer Zug als vorgesehen ist auf der Zuglaufmeldestelle	-/✓	×
4.h	EARLY	Zug ist eher als geplant auf der Zuglaufmeldestelle	✓/✓	✓
4.i	LATE	Zug ist später als geplant auf der Zuglaufmeldestelle	-/✓	✓
4.j	BEFORE	Zug ist fälschlicherweise eher als ein anderer auf der Zuglaufmeldestelle	✓/✓	×
4.k	AFTER	Zug ist fälschlicherweise später als ein anderer auf der Zuglaufmeldestelle	✓/✓	×

6.2.5 FA(F, B)

Nr.	Guide Word	Spezifische Bedeutung	H/O	beschr.
5.a	NOT	Es wurde wider Erwarten keine Fahranfrage gestellt	-/✓	✓
5.b	MORE	—	-/-	-
5.c	LESS	—	-/-	-
5.d	AS WELL AS	—	-/-	-
5.e	PART OF	—	-/-	-
5.f	REVERSE	—	-/-	-
5.g	OTHER THAN	Fahranfrage für eine andere Zuglaufmeldestelle wurde gestellt	-/✓	×
5.h	EARLY	Fahranfrage wurde früher als geplant gestellt	✓/✓	✓
5.i	LATE	Fahranfrage wurde später als geplant gestellt	-/✓	✓
5.j	BEFORE	Fahranfrage wurde vor Erreichen der Zuglaufmeldestelle gestellt	✓/✓	✓
5.k	AFTER	Fahranfrage wurde erst nach der Fahrerlaubnis gestellt	✓/✓	✓

6.2.6 $\mathrm{FE}(F, B)$

Nr.	Guide Word	Spezifische Bedeutung	H/O	beschr.
6.a	NOT	Es wurde wider Erwarten keine Fahrerlaubnis erteilt	-/✓	✓
6.b	MORE	—	-/-	-
6.c	LESS	—	-/-	-
6.d	AS WELL AS	—	-/-	-
6.e	PART OF	—	-/-	-
6.f	REVERSE	Fahrerlaubnis wurde verweigert	-/✓	✓
6.g	OTHER THAN	Fahrerlaubnis zu einer anderen Zuglaufmeldestelle wurde erteilt	✓/✓	×
6.h	EARLY	Fahrerlaubnis wurde früher als geplant erteilt	✓/✓	✓
6.i	LATE	Fahrerlaubnis wurde später als geplant erteilt	-/✓	✓
6.j	BEFORE	Fahrerlaubnis wurde vor der Fahranfrage erteilt	✓/✓	✓
6.k	AFTER	Fahrerlaubnis wurde erst nach der Abfahrt erteilt	-/-	✓

6.2.7 $\mathrm{AFE}(F, A, B)$

Nr.	Guide Word	Spezifische Bedeutung	H/O	beschr.
7.a	NOT	Es wurde wider Erwarten die Fahrerlaubnis nicht verweigert	✓/✓	✓
7.b	MORE	—	-/-	-
7.c	LESS	—	-/-	-
7.d	AS WELL AS	—	-/-	-
7.e	PART OF	—	-/-	-
7.f	REVERSE	Fahrerlaubnis wurde erteilt	✓/✓	✓
7.g	OTHER THAN	—	-/-	-
7.h	EARLY	Verweigerung der Fahrerlaubnis wurde früher als geplant gegeben	-/✓	✓
7.i	LATE	Verweigerung der Fahrerlaubnis wurde später als geplant gegeben	✓/✓	✓
7.j	BEFORE	Verweigerung der Fahrerlaubnis wurde vor der Fahranfrage gegeben	✓/✓	-
7.k	AFTER	Verweigerung der Fahrerlaubnis wurde erst nach der Abfahrt gegeben	✓/✓	-

6.2.8 $\mathrm{KH}(F, A, B)$

Nr.	Guide Word	Spezifische Bedeutung	H/O	beschr.
8.a	NOT	Es sind Hindernisse bekannt	-/-	-
8.b	MORE	—	-/-	-
8.c	LESS	—	-/-	-
8.d	AS WELL AS	—	-/-	-
8.e	PART OF	—	-/-	-
8.f	REVERSE	Es sind Hindernisse bekannt	-/-	-
8.g	OTHER THAN	—	-/-	-
8.h	EARLY	—	-/-	-
8.i	LATE	Es wird zu spät bekannt, dass keine Hindernisse vorliegen	-/✓	×
8.j	BEFORE	—	-/-	-
8.k	AFTER	Es wird erst nach der Fahranfrage bekannt, dass keine Hindernisse vorliegen	-/✓	×

6.2.9 $\mathrm{Next}(F, A) = B$

Nr.	Guide Word	Spezifische Bedeutung	H/O	beschr.
9.a	NOT	B ist nicht die nächste vorgesehene Zuglaufmeldestelle	-/-	✓
9.b	MORE	—	-/-	-
9.c	LESS	—	-/-	-
9.d	AS WELL AS	—	-/-	-
9.e	PART OF	—	-/-	-
9.f	REVERSE	—	-/-	-
9.g	OTHER THAN	Eine andere Zuglaufmeldestelle als B ist die nächste	-/✓	✓
9.h	EARLY	—	-/-	-
9.i	LATE	—	-/-	-
9.j	BEFORE	—	-/-	-
9.k	AFTER	—	-/-	-

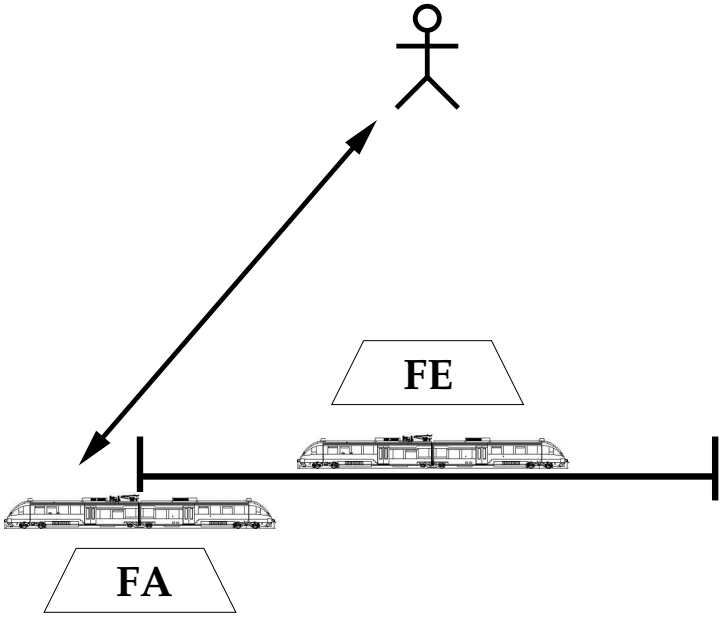
6.3 Verschiebung der HAZOP-Studien

Die hier gezeigten Tabellen sollen einen kleinen Eindruck geben, welchen Umfang diese Analyse bereits in dieser frühen Verfeinerungsebene erreichen können. Die HAZOP-Studien werden hier noch nicht durchgeführt, da es zunächst sinnvoll ist, eine vollständige Spezifikation einer Zugfahrt im Zugleitbetrieb zu erstellen. Insbesondere soll hier eine vollständige Spezifikation in streng formaler Form und mit rigoroser Semantik erstellt werden.

Nach einer weiteren Verfeinerung (Ebene 3) wird sich zeigen, dass fast alle der hier festgestellten potentiellen Hazards durch das Betriebsverfahren vermieden werden.

Kapitel 7

Ebene 3



7.1 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 [Lam95] vorgestellten Predicate-Action-Diagrammen[Lam95], die sich vollständig auf eine Spezifikation in TLA abbilden lassen. Vollständig heißt

in diesem Falle, dass in allen vom Anfangszustand beginnenden Pfade im Endzustand enden. Durch die Analyse der Predicate-Action-Diagramme lässt sich eine Vervollständigung der Verfahren vorschlagen, dieses Verfahren wurde von Ladkin in [Lad95] vorgeschlagen.

Zur weiteren detaillierten Beschreibung der Verfahren im Zugleitbetrieb bietet es sich an, die Kommunikation zu analysieren. Dabei soll das wesentliche am Verfahren des Zugleitbetriebs, der Austausch von Zuglaufmeldungen, besonders betrachtet werden. In der Informatik sind Kommunikationsmethoden bereits seit mehreren Jahrzehnten Gegenstand der Forschung. Hier sind insbesondere Protokolle entwickelt worden, die eine zuverlässige Kommunikation über ein unzuverlässiges Medium garantieren. Das am weitesten verbreitete Protokoll ist wohl TCP (zuverlässig) auf IP (unzuverlässig).

Diese im Internet verwendeten Standardprotokolle sind ausführlich in zahlreichen Werken beschrieben, stellvertretend sei hier nur [PMRL04] genannt.

Andere Techniken wie beispielsweise das synchrone Aktualisieren von Daten in verteilten Systemen mittels *two-phase commit* [SS83] sind lange bekannt und ihre Relevanz für Kommunikation zwischen Zugleiter und Zugführern ist zu untersuchen. Eine darauf basierende computerunterstützte Kommunikationstechnik mit vollständig verifizierten Algorithmen und Implementationen könnte insbesondere die Kommunikationssicherheit erheblich erhöhen. Misslungene Kommunikation zwischen beteiligten Menschen war trotz vorgeschriebener Wortlaute bereits Ursache für Unfälle.

7.1.1 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. Diese Nachrichten enthalten jeweils verschiedene Parameter, FA und FE haben jeweils Zugname/Zugnummer und die nächste Zuglaufmeldestelle als Parameter. Die Ablehnung der Fahrerlaubnis hat keine Parameter (erkennbar am Wortlaut „Nein, warten.“, in dem keine Zugnummer und keine Zuglaufmeldestelle

vorkommen.) Die Ankunftmeldung hat Zugname/Zugnummer und die aktuelle Zuglaufmeldestelle als Parameter. Zu beachten ist, dass diese Nachrichten *nicht* dasselbe sind wie die Prädikate FA und FE aus Ebene 2. Diese bezeichnen den Zustand, dass die Fahranfrage korrekt und vollständig gestellt wurde, bzw. dass tatsächlich die Fahrerlaubnis vorliegt. Die Abkürzungen hier bezeichnen die Nachricht, die übermittelt wird.

$\mathrm{Sent}(N, \mathrm{Parameter})$ Nachricht N wurde gesendet

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

Diese definieren jeweils folgendermaßen die entsprechenden Prädikate aus Ebene 2:

7.1.2 Bedeutungspostulate

$$\left.\begin{array}{l} \wedge\, \mathrm{inZ}(Z, A) \\ \wedge\, \mathrm{Sent}(\mathrm{FA}, Z, \mathrm{Next}(Z, A)) \\ \wedge\, \mathrm{Recd}(\mathrm{FA}, Z, \mathrm{Next}(Z, A)) \end{array}\right\} \Rightarrow \mathrm{FA}(Z, A, \mathrm{Next}(Z, A)) \qquad \text{(M3.1)}$$

$$\left.\begin{array}{l} \wedge\, \mathrm{inZ}(Z, A) \\ \wedge\, \mathrm{Sent}(\mathrm{FE}, Z, \mathrm{Next}(Z, A)) \\ \wedge\, \mathrm{Recd}(\mathrm{FE}, Z, \mathrm{Next}(Z, A)) \end{array}\right\} \Rightarrow \mathrm{FE}(Z, A, \mathrm{Next}(Z, A)) \qquad \text{(M3.2)}$$

$$\left.\begin{array}{l} \wedge\, \mathrm{inZ}(Z, A) \\ \wedge\, \mathrm{Sent}(\mathrm{AFE}) \end{array}\right\} \Rightarrow \mathrm{AFE}(Z, A, \mathrm{Next}(Z, A)) \qquad \text{(M3.3)}$$

7.1.3 Auflösung

Im dem Falle, dass eine Nachricht nicht korrekt empfangen wurde (in dieser Betrachtung bedeutet dies, dass eine falsche Zugnummer, oder eine falsche Zuglaufmeldestelle empfangen wurde), gibt es, je nachdem, welches Fehler vorliegt, verschiedene Möglichkeiten der Auflösung. Diese Auflösungsmöglichkeiten sind in den Diagrammen der Zustandsautomaten jeweils mit einer gestrichelten Linie dargestellt.

Watchdog-Timer

In bestimmten Fällen wird eine bestimmte Meldung spätestens nach Ablauf einer bestimmten Frist erwartet. Läuft diese Frist ab, ohne dass die entsprechende Nachricht empfangen wurde, so kann diese beispielsweise erneut angefordert werden. Zustandsübergänge, in denen dieses Verfahren angewendet wird, erhalten die Bezeichnung „Watchdog“ mit einer entsprechenden Bezeichnung.

Auflösung durch Einholen einer Bestätigung

Für einen bestimmten Zug in einer bestimmten Zuglaufmeldestelle ist die nächste fahrplanmäßige Zuglaufmeldestelle immer festgelegt. Wird eine Nachricht empfangen mit der richtigen Zugnummer, aber einer falschen Zuglaufmeldestelle, so fragt der Empfänger nach Klärung, und bekommt die Nachricht erneut übermittelt. Diese Art der Problemlösung wird am Zustandsübergang im Diagramm des Automaten mit „Resolution“ bezeichnet.

7.2 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. Das verwendete Modell zur Kommunikation basiert auf asynchroner Kommunikation, d. h. Senden und Empfangen einer Nachricht sind zwei getrennte Vorgänge und es gibt keinen Mechanismus, der die korrekte Reihenfolge von Nachrichten gewährleistet. Auch wenn typischerweise Zugfunk eingesetzt wird (der synchrone Kommunikation ermöglicht) für die die Zugfolge regelnden Zuglaufmeldungen, so sind doch auch andere Verfahren als Rückfallebene möglich, wo dies nicht der Fall ist.

Betrachtet werden hier idealisierte Kommunikationsteilnehmer, die empfangene Nachrichten stets korrekt beurteilen, und sich immer an alle Vorschriften halten. Es wird nur untersucht, wie sich die Verstümmelung von Nachrichten während der Übertragung auf das Verfahren auswirken kann.

Dazu werden unter anderem die sogenannten *Rational Cognitive Models* (*RCM*) der beteiligten Personen untersucht. Diese bezeichnen das Modell der Wirklichkeit, das diese jeweils hätten, wenn sie idealisierte Roboter wären. Das bedeutet insbesondere, die Untersuchung menschlichen Fehlverhaltens, oder eine Risikoabschätzung, ist nicht Teil dieser Arbeit. Wenn die *RCM*s der beteiligten Personen nicht mit der Realität und/oder nicht mit den *RCM*s der andere Personen übereinstimmen, kann dies einen Hazard bedeuten.

7.2.1 Beschreibung der Zustände

Gegenüber der Ebene 2 ist dieser Graph um so vieles größer geworden, dass er in einzelne Teilbereiche aufgeteilt wird. Die Zustände des ersten Teilbereich sind in Abbildung 7.1 dargestellt.

$$
\begin{aligned}
s0 = {} & \mathrm{inZ}(Z, A) \\
\\
s1 = {} & \wedge \mathrm{inZ}(Z, A) \\
& \wedge \mathrm{Sent}(\mathrm{FA}, Z, \mathrm{Next}(Z, A)) \\
\\
s2 = {} & \wedge \mathrm{inZ}(Z, A) \\
& \wedge \mathrm{Sent}(\mathrm{FA}, Z, \mathrm{Next}(Z, A)) \\
& \wedge \mathrm{Recd}(\mathrm{FA}, Z, \mathrm{Next}(Z, A)) \\
& \wedge \mathrm{FA}(Z, A, \mathrm{Next}(Z, A)) \\
\\
s3 = {} & \wedge \mathrm{inZ}(Z, A) \\
& \wedge \mathrm{Sent}(FA, Z, \mathrm{Next}(Z, A)) \\
& \wedge \neg \mathrm{Recd}(FA, Z, \mathrm{Next}(Z, A))
\end{aligned}
$$

Abbildung 7.1: Zustände des Automaten in Abb. 7.2

Der zugehörige Automat ist in Abbildung 7.2 dargestellt, die Zusammenfassung dieses Automaten ist der Übergang vom Zustand $s0$ in den

Zustand $s2$. Im Ausgangszustand $s0$ und im gewünschten Endzustand $s2$ stimmen die *Rational Cognitive Models* aller Beteiligten überein: Die Fahranfrage ist gestellt worden, und sowohl Zugleiter als auch Zugführer wissen darum.

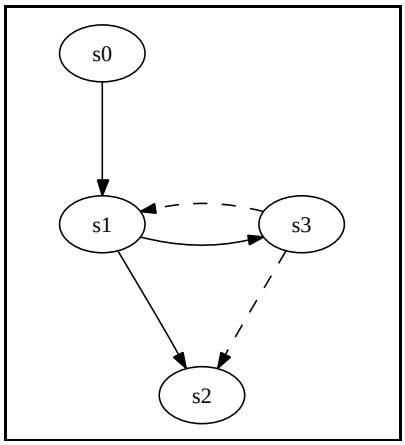


Abbildung 7.2: Zustandsmaschine 3.1

Besondere Aufmerksamkeit verdient der Zustand $s3$, da bei ihm eine Fallunterscheidung getroffen werden muss, was es im Einzelnen bedeutet, dass die Nachricht $(\mathrm{FA}, Z, \mathrm{Next})$ nicht empfangen wurde. Verschiedene Fälle von Verstümmelung der Nachricht sind vorstellbar, die verschieden gehandhabt werden müssen.

Die Formulierung $\neg\,\mathrm{Recd}(FA, Z, \mathrm{Next}(F, A))$ bedeutet zunächst nur, dass die Nachricht nicht korrekt empfangen wurde, realistisch mögliches Verstümmelungen betreffen die Zugnummer und die genannte Zuglaufmeldestelle.

Abbildung 7.3 zeigt die formale Darstellung der unterschiedlichen Fälle, in denen die Nachricht verstümmelt sein könnte.

3.1 : In diesem Falle ist eine falsche Zugnummer, nämlich die eines anderen Zuges Z_1 empfangen worden. In diesem Fall ist es irrelevant, welche Aktion, und welche Zielzuglaufmeldestelle empfangen wurde. Dies stellt einen Hazard dar, da, falls Zug Z_1 existiert, das RCM des Zugleiters nun das Prädikat (z. B.) $\mathrm{FA}(Z_1, A, (\mathrm{Next}(Z, A))$ enthält. Die Situation wird sich nach Ablauf eines Watchdog-Timers

beim Zugführer von Z lösen, da dieser entweder eine Erteilung oder Ablehnung der Fahrerlaubnis erwartet, diese aber nicht erhält, und rückfragt.

$s3.2$: Die Aktion, d. h. in diesem Falle, die Art der Zuglaufmeldung ist nicht korrekt übertragen worden. Da der Zugleiter eine Fahranfrage von dem Zug erwartet (Zugnummer und Zuglaufmeldestelle seien korrekt übermittelt worden), wird er sich erkundigen, ob eine Fahranfrage gemeint war, und die Fahranfrage wird erneut gestellt werden, was eine Rückkehr in Zustand $s1$ bedeutet.

$s3.3$: Dies beschreibt den Fall, dass das Ziel der Fahranfrage nicht korrekt empfangen wurde. Da sowohl Zugleiter als auch Zugführer (unter der Annahme perfekter Roboter im Sinne der Rational Cognitive Models) immer wissen, welches das Ziel einer Fahranfrage für einen bestimmten Zug in einer bestimmten Zuglaufmeldestelle ist, wird der Zugleiter rückfragen, und sich das korrekte Ziel bestätigen lassen.

Der nächste Teilautomat lässt sich zusammenfassen als der Übergang von $S2$ nach $s6$, d. h. vom Zustand, in dem die Fahranfrage gestellt worden ist, bis zu dem Zustand, in dem die Fahrerlaubnis erteilt wurde. Siehe Abbildungen 7.4 und 7.5.

Anmerkung zu AFE Dieser Fall einer Nachricht wird etwas anders behandelt, als die anderen. Es wird davon ausgegangen, da aufgrund des besonderen Wortlauts, („Nein, Warten.“) und des Fehlens der bei den anderen Nachrichten vorkommenden Parameter *Zug* und *Ziel*, diese Nachricht nicht verstümmelt empfangen wird.

Der Unterzustand, in dem die Nachricht verstümmelt empfangen wurde, ($s7$) wird wieder gesondert betrachtet, siehe Abbildung Substates-3.2.

Die dritte Phase betrifft die eigentliche Fahrt des Zuges nach Erteilung der Fahrerlaubnis, von Zuglaufmeldestelle A nach Zuglaufmeldestelle $\mathrm{Next}(Z, A)$. Siehe Abbildungen 7.7 und 7.8.

Der Unterzustand, in dem die Nachricht verstümmelt empfangen wurde, ($s13$) wird wieder gesondert betrachtet, siehe Abbildung 7.9.

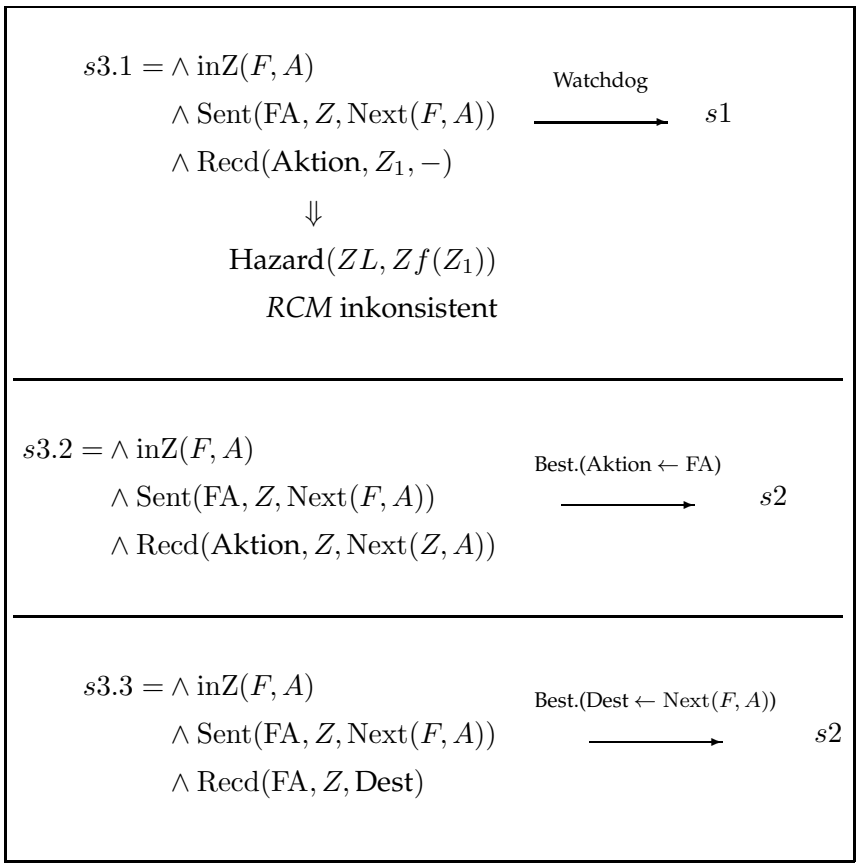


Abbildung 7.3: Unterzustände von $s3$ des Automaten in Abb. 7.2

7.3 Sicherheitsaxiome

Durch eine geeignete Formulierung der in der FV-NE beschriebenen Kommunikation könnte es möglich sein, dass das Sicherheitsaxiom (S2.2) zu beweisen. Ebenso denkbar ist, dass die Durchführung der Kommunikation, „nach Vorschrift“ die Erfüllung von (S2.2) *nicht* gewährleisten kann, und somit ein Problem mit der Vorschrift identifiziert (und bewiesen!) ist.

Sicherheitsaxiom (S2.1) wird durch ein ähnliches Axiom auf Ebene 3 (S3.1) gewährleistet werden.

Das einzige Problem, das direkt hier auftaucht, ist die Transition von $s10$ in einen undefinierten Zustand. Es ist im Zugleitbetriebsver-

$$
\begin{aligned}
s2 = {} & \wedge \mathrm{inZ}(Z, A) \\
& \wedge \mathrm{FA}(Z, A, \mathrm{Next}(Z, A))
\end{aligned}
$$

$$
\begin{aligned}
s4 = {} & \wedge \mathrm{inZ}(Z, A) \\
& \wedge \mathrm{FA}(Z, A, \mathrm{Next}(Z, A)) \\
& \wedge \mathrm{KH}(Z, A, \mathrm{Next}(Z, A)) \\
& \wedge \mathrm{Sent}(\mathrm{FE}, Z, \mathrm{Next}(Z, A))
\end{aligned}
$$

$$
\begin{aligned}
s5 = {} & \wedge \mathrm{inZ}(Z, A) \\
& \wedge \mathrm{FA}(Z, A, \mathrm{Next}(Z, A)) \\
& \wedge \neg\, \mathrm{KH}(Z, A, \mathrm{Next}(Z, A)) \\
& \wedge \mathrm{Sent}(\mathrm{AFE})
\end{aligned}
$$

$$
\begin{aligned}
s6 = {} & \wedge \mathrm{inZ}(Z, A) \\
& \wedge \mathrm{FA}(Z, A, \mathrm{Next}(Z, A)) \\
& \wedge \mathrm{KH}(Z, A, \mathrm{Next}(Z, A)) \\
& \wedge \mathrm{Sent}(\mathrm{FE}, Z, \mathrm{Next}(Z, A)) \\
& \wedge \mathrm{Recd}(\mathrm{FE}, Z, \mathrm{Next}(Z, A))
\end{aligned}
$$

$$
\begin{aligned}
s7 = {} & \wedge \mathrm{inZ}(Z, A) \\
& \wedge \mathrm{FA}(Z, A, \mathrm{Next}(Z, A)) \\
& \wedge \mathrm{KH}(Z, A, \mathrm{Next}(Z, A)) \\
& \wedge \mathrm{Sent}(FE, Z, \mathrm{Next}(Z, A)) \\
& \wedge \neg\, \mathrm{Recd}(FE, Z, \mathrm{Next}(Z, A))
\end{aligned}
$$

Abbildung 7.4: Zustände des Automaten in Abb. 7.5

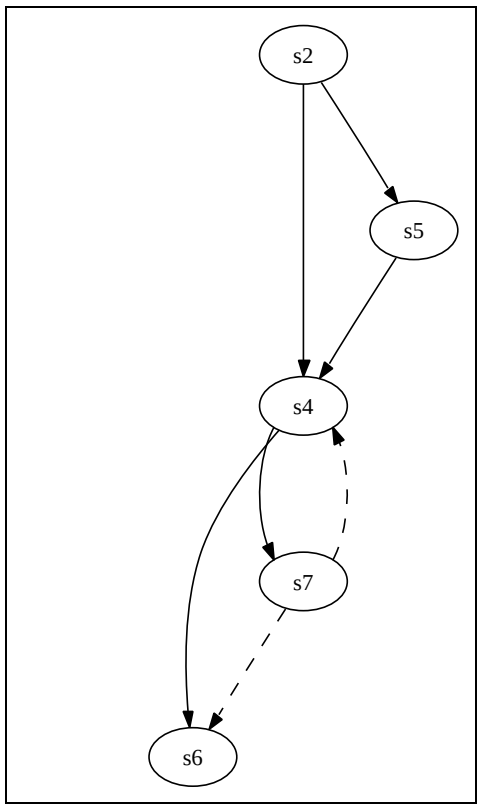


Abbildung 7.5: Zustandsmaschine 3.2

fahren nicht klar, was gemacht werden soll, wenn die Bedingung, die zur Erteilung der Fahrerlaubnis notwendig war, nämlich, dass keine bekannten Hindernisse der Fahrt entgegenstehen, *während* der Fahrt wegfällt, indem ein Hindernis entdeckt wird.

Analog zu Abschnitt 5.4 ist Abbildung 7.10 Sicherheitsaxiom, da es verlangt, dass neben den im Predicate-Action-Diagramm beschriebenen Schritten, nichts anderes stattfinden kann. Darauf gründet sich die Analyse.

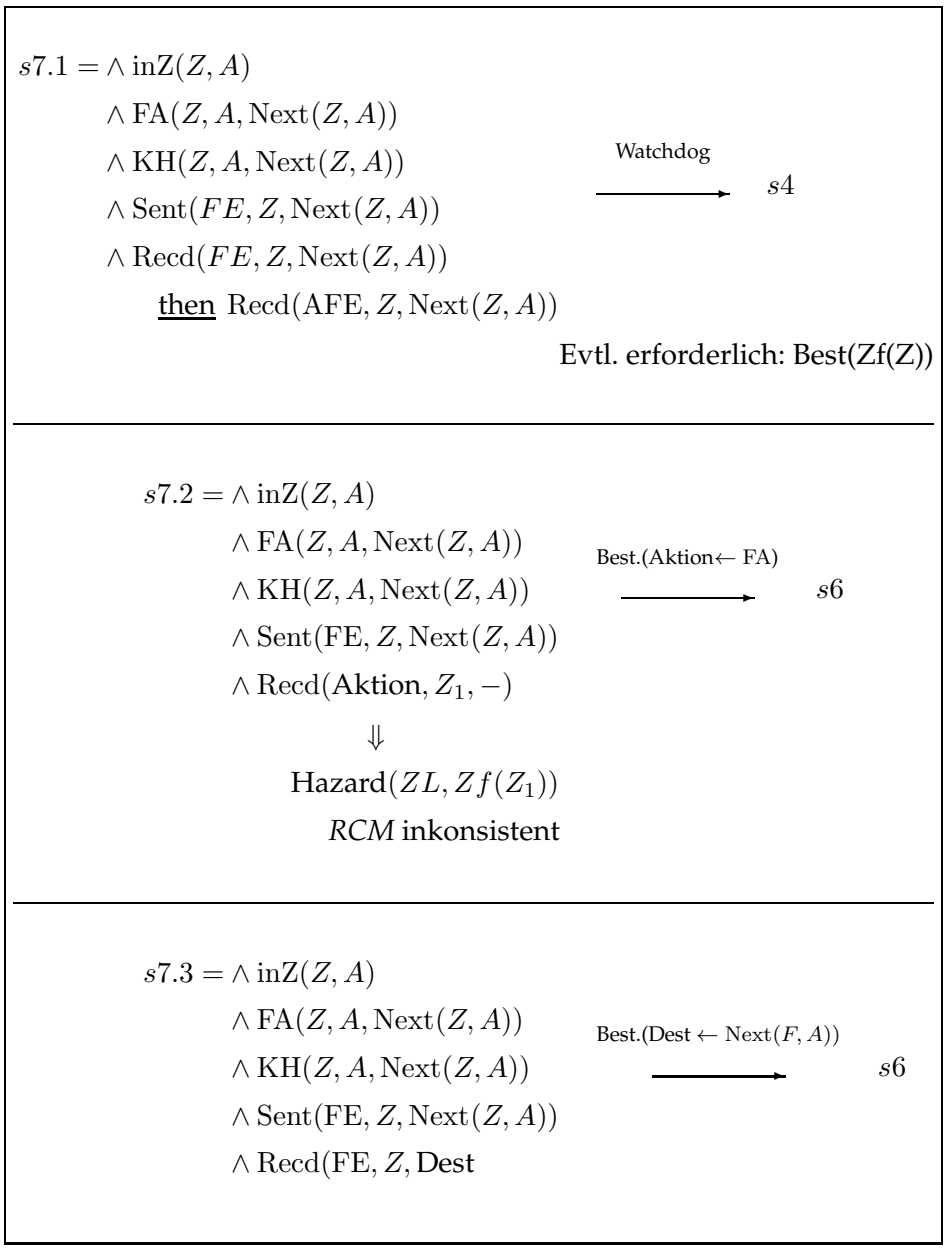


Abbildung 7.6: Unterzustände von $s3$ des Automaten in Abb. 7.5

7.4 Beweis der Verfeinerung

Mit Hilfe der Bedeutungspostulate der Ebene 3 lässt sich zeigen, dass sich das Predicate-Action-Diagramm aus Ebene 2 (Abbildungen 5.3 und 5.5) auf das von Ebene 3 abbilden lässt. Dabei entsprechen jeweils mehrere Zwischenzustände und zusätzliche Übergänge auf Ebene 3 einem Zustand auf Ebene 2. Diese Eigenschaft wird auch als *stuttering* (Engl. „Stottern“) bezeichnet.

$$
\begin{aligned}
s6 = {} & \wedge \mathrm{inZ}(Z, A) \\
& \wedge \mathrm{FE}(Z, A, \mathrm{Next}(Z, A)) \\
& \wedge \mathrm{KH}(Z, A, \mathrm{Next}(Z, A))
\end{aligned}
$$

$$
\begin{aligned}
s8 = {} & \wedge \mathrm{zw}(Z, A, \mathrm{Next}(Z, A)) \\
& \wedge \mathrm{FE}(Z, A, \mathrm{Next}(Z, A)) \\
& \wedge \mathrm{KH}(Z, A, \mathrm{Next}(Z, A)) \\
& \wedge \neg \mathrm{LV}(Z)
\end{aligned}
$$

$$
\begin{aligned}
s9 = {} & \wedge \mathrm{inZ}(Z, \mathrm{Next}(Z, A)) \\
& \wedge \mathrm{FE}(Z, A, \mathrm{Next}(Z, A))
\end{aligned}
$$

$$
\begin{aligned}
s10 = {} & \wedge \mathrm{zw}(Z, A, \mathrm{Next}(Z, A)) \\
& \wedge \mathrm{FE}(Z, A, \mathrm{Next}(Z, A)) \\
& \wedge \neg \mathrm{KH}(Z, A, \mathrm{Next}(Z, A)) \\
& \wedge \neg \mathrm{LV}(Z)
\end{aligned}
$$

$$
\begin{aligned}
s11 = {} & \wedge \mathrm{inZ}(Z, \mathrm{Next}(Z, A)) \\
& \wedge \mathrm{FE}(Z, A, \mathrm{Next}(Z, A)) \\
& \wedge \mathrm{Sent}(\mathrm{AM}, Z, \mathrm{Next}(Z, A))
\end{aligned}
$$

$$
\begin{aligned}
s12 = {} & \wedge \mathrm{inZ}(Z, \mathrm{Next}(Z, A)) \\
& \wedge \mathrm{FE}(Z, A, \mathrm{Next}(Z, A)) \\
& \wedge \mathrm{Sent}(\mathrm{AM}, Z, \mathrm{Next}(Z, A)) \\
& \wedge \mathrm{Recd}(\mathrm{AM}, Z, \mathrm{Next}(Z, A)) \\
= {} & s0
\end{aligned}
$$

$$
\begin{aligned}
s13 = {} & \wedge \mathrm{inZ}(Z, \mathrm{Next}(Z, A)) \\
& \wedge \mathrm{FE}(Z, A, \mathrm{Next}(Z, A)) \\
& \wedge \mathrm{Sent}(\mathrm{AM}, Z, \mathrm{Next}(Z, A)) \\
& \wedge \neg \mathrm{Recd}(\mathrm{AM}, Z, \mathrm{Next}(Z, A))
\end{aligned}
$$

Abbildung 7.7: Zustände des Automaten in Abb. 7.5

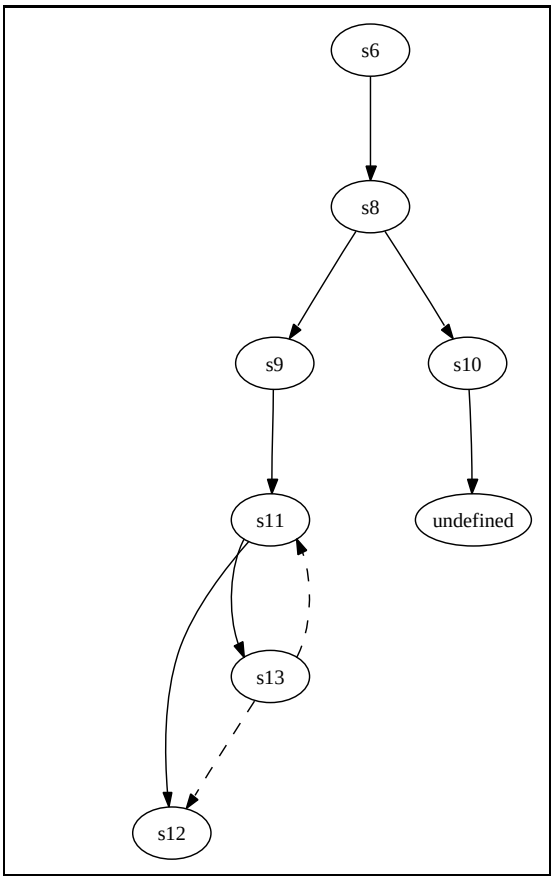


Abbildung 7.8: Zustandsmaschine 3.3

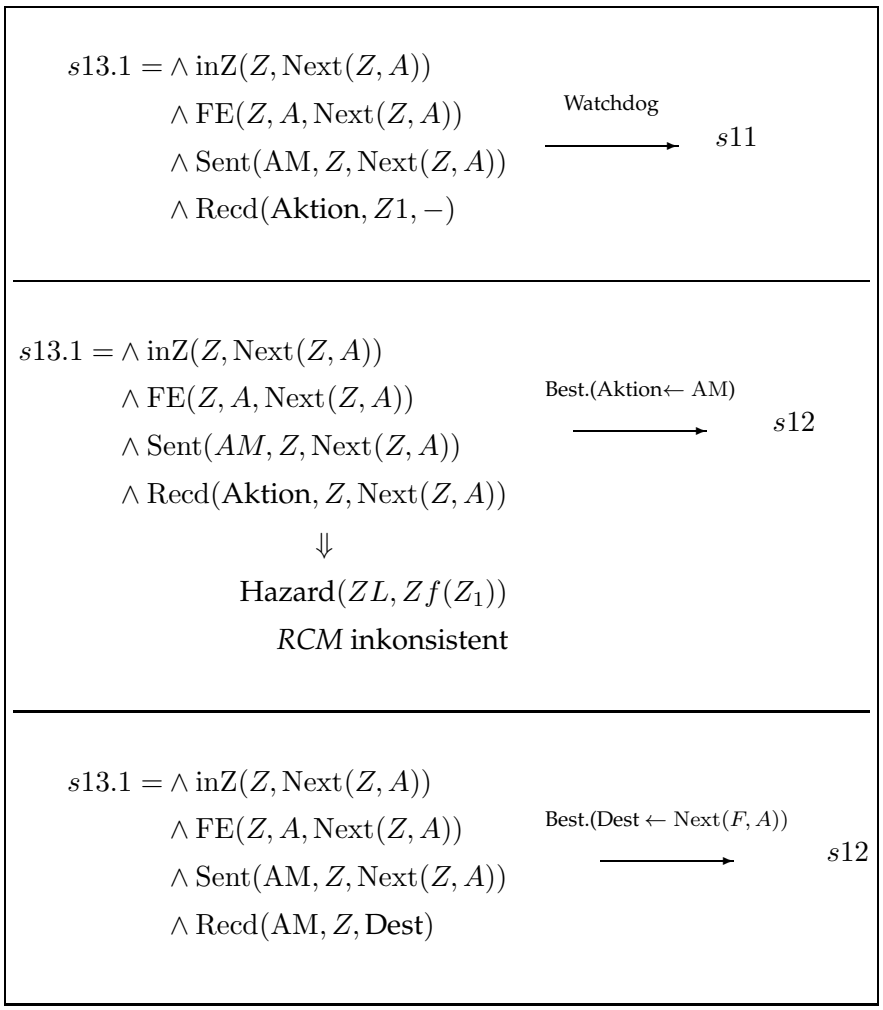


Abbildung 7.9: Unterzustände von $s3$ des Automaten in Abb. 7.8

$$
\begin{aligned}
&\Box \vee (s0 \wedge s1') \qquad \text{(S3.1)}\\
&\quad \vee (s1 \wedge s2')\\
&\quad \vee (s1 \wedge s3')\\
&\quad \vee (s3 \wedge s1')\\
&\quad \vee (s3 \wedge s2')\\
&\quad \vee (s2 \wedge s4')\\
&\quad \vee (s2 \wedge s5')\\
&\quad \vee (s5 \wedge s4')\\
&\quad \vee (s4 \wedge s6')\\
&\quad \vee (s4 \wedge s7')\\
&\quad \vee (s7 \wedge s6')\\
&\quad \vee (s7 \wedge s4')\\
&\quad \vee (s6 \wedge s8')\\
&\quad \vee (s8 \wedge s9')\\
&\quad \vee (s8 \wedge s10')\\
&\quad \vee (s9 \wedge s11')\\
&\quad \vee (s11 \wedge s13')\\
&\quad \vee (s11 \wedge s12')\\
&\quad \vee (s13 \wedge s11')\\
&\quad \vee (s13 \wedge s12')\\
&\quad \vee (s10 \wedge undefined')
\end{aligned}
$$

Abbildung 7.10: Sicherheitsaxiom auf Ebene 3

E3	E2	
$s0$	$s0$	$\mathrm{inZ}(Z, A) \Rightarrow \mathrm{inZ}(Z, A)$
$s2$	$s1$	$\begin{pmatrix} \wedge\, \mathrm{inZ}(Z, A) \\ \wedge\, \mathrm{Sent}(\mathrm{FA}, Z, \mathrm{Next}(Z, A)) \\ \wedge\, \mathrm{Recd}(\mathrm{FA}, Z, \mathrm{Next}(Z, A)) \end{pmatrix} \Rightarrow \begin{pmatrix} \wedge\, \mathrm{inZ}(Z, A) \\ \wedge\, \mathrm{FA}(Z, A, \mathrm{Next}(Z, A)) \end{pmatrix}$ (M3.1)
$s4$	$s2$	$\begin{pmatrix} \wedge\, \mathrm{inZ}(Z, A) \\ \wedge\, \mathrm{FA}(Z, A, \mathrm{Next}(Z, A)) \\ \wedge\, \mathrm{KH}(Z, A, \mathrm{Next}(Z, A)) \\ \wedge\, \mathrm{Sent}(\mathrm{FE}, Z, \mathrm{Next}(Z, A)) \end{pmatrix} \Rightarrow \begin{pmatrix} \wedge\, \mathrm{inZ}(Z, A) \\ \wedge\, \mathrm{FA}(Z, A, \mathrm{Next}(Z, A)) \\ \wedge\, \mathrm{KH}(Z, A, \mathrm{Next}(Z, A)) \end{pmatrix}$
$s5$	$s3$	$\begin{pmatrix} \wedge\, \mathrm{inZ}(Z, A) \\ \wedge\, \mathrm{FA}(Z, A, \mathrm{Next}(Z, A)) \\ \wedge\, \neg\, \mathrm{KH}(Z, A, \mathrm{Next}(Z, A)) \\ \wedge\, \mathrm{Sent}(\mathrm{AFE}) \end{pmatrix} \Rightarrow \begin{pmatrix} \wedge\, \mathrm{inZ}(Z, A) \\ \wedge\, \mathrm{FA}(Z, A, \mathrm{Next}(Z, A)) \\ \wedge\, \neg\, \mathrm{KH}(Z, A, \mathrm{Next}(Z, A)) \\ \wedge\, \mathrm{AFE}(Z, A, \mathrm{Next}(Z, A)) \end{pmatrix}$ (M3.3)
$s6$	$s4$	$\begin{pmatrix} \wedge\, \mathrm{inZ}(Z, A) \\ \wedge\, \mathrm{FA}(Z, A, \mathrm{Next}(Z, A)) \\ \wedge\, \mathrm{KH}(Z, A, \mathrm{Next}(Z, A)) \\ \wedge\, \mathrm{Sent}(\mathrm{FE}, Z, \mathrm{Next}(Z, A)) \\ \wedge\, \mathrm{Recd}(\mathrm{FE}, Z, \mathrm{Next}(Z, A)) \end{pmatrix} \Rightarrow \begin{pmatrix} \wedge\, \mathrm{inZ}(Z, A) \\ \wedge\, \mathrm{FA}(Z, A, \mathrm{Next}(Z, A)) \\ \wedge\, \mathrm{KH}(Z, A, \mathrm{Next}(Z, A)) \\ \wedge\, \mathrm{FE}(Z, A, \mathrm{Next}(Z, A)) \end{pmatrix}$ (M3.2)
$s8$	$s5$	$\begin{pmatrix} \wedge\, \mathrm{zw}(Z, A, \mathrm{Next}(Z, A)) \\ \wedge\, \mathrm{FE}(Z, A, \mathrm{Next}(Z, A)) \\ \wedge\, \mathrm{KH}(Z, A, \mathrm{Next}(Z, A)) \\ \wedge\, \neg\, \mathrm{LV}(Z) \end{pmatrix} \Rightarrow \begin{pmatrix} \wedge\, \mathrm{zw}(Z, A, \mathrm{Next}(Z, A)) \\ \wedge\, \mathrm{FE}(Z, A, \mathrm{Next}(Z, A)) \\ \wedge\, \mathrm{KH}(Z, A, \mathrm{Next}(Z, A)) \\ \wedge\, \neg\, \mathrm{LV}(Z) \end{pmatrix}$
$s12$	$s6$	wie $s0$
$s10$	$s7$	$\begin{pmatrix} \wedge\, \mathrm{zw}(Z, A, \mathrm{Next}(Z, A)) \\ \wedge\, \mathrm{FE}(Z, A, \mathrm{Next}(Z, A)) \\ \wedge\, \neg\, \mathrm{KH}(Z, A, \mathrm{Next}(Z, A)) \\ \wedge\, \neg\, \mathrm{LV}(Z) \end{pmatrix} \Rightarrow \begin{pmatrix} \wedge\, \mathrm{zw}(Z, A, \mathrm{Next}(Z, A)) \\ \wedge\, \mathrm{FE}(Z, A, \mathrm{Next}(Z, A)) \\ \wedge\, \neg\, \mathrm{KH}(Z, A, \mathrm{Next}(Z, A)) \\ \wedge\, \neg\, \mathrm{LV}(Z) \end{pmatrix}$

Abbildung 7.11: Beweis der Verfeinerungen Ebene 3 → Ebene 2

Transition	Paragraphen/Abschnitte in der FV-NE
$s0 \rightarrow s1$	§§10(3), 10(4)
$s1 \rightarrow s2$	§§10(3), 10(4)
$s1 \rightarrow s3$	(*Anlage 9, II.(7)*)
$s3 \rightarrow s1$	?
$s3 \rightarrow s2$	Anlage 9, II.(7)
$s2 \rightarrow s4$	§§10(4), 17(7)
$s2 \rightarrow s5$	§10(4)
$s5 \rightarrow s4$	§10(4)
$s4 \rightarrow s6$	§§10(3), 10(4)
$s4 \rightarrow s7$	(*Anlage 9, II.(7)*)
$s7 \rightarrow s6$	Anlage 9, II.(7)
$s7 \rightarrow s4$	?
$s6 \rightarrow s8$	§§17(6), 42(6)
$s8 \rightarrow s9$	§17(1)
$s8 \rightarrow s10$	§44(13)
$s9 \rightarrow s11$	§§10(3), 10(5)
$s11 \rightarrow s13$	(*Anlage 9, II.(7)*)
$s11 \rightarrow s12$	Anlage 9, II.(7)
$s13 \rightarrow s11$	§§10(12), 10(13)
$s13 \rightarrow s12$	Anlage 9, II.(7)
$s10 \rightarrow undefined$	?

Abbildung 7.12: Entsprechung der Transitionen in der FV-NE

Kapitel 8

HAZOP-Studien

Die Auflistung der potentiellen Hazards in Kapitel 6 erfolgte „naiv“, d. h. ohne Berücksichtigung des tatsächlichen Verfahrens wurde abgewogen, ob eine Abweichung vom beabsichtigten Verhalten einen Hazard oder ein Betriebsproblem darstellen könnte. Eine genauere Untersuchung jedes einzelnen Falls anhand des in Ebene 3 beschriebenen Verfahrens ergibt, dass die HAZOP-Studien *keine* neuen Hazards liefern. Eine genauere Untersuchung der Betriebsprobleme (engl. *Operability Problems*) soll an dieser Stelle nicht erfolgen, es genügt, sicherzustellen, dass der Betrieb *sicher* durchgeführt werden kann.

Ein Nachweis, dass der Betrieb mit dem entwickelten Protokoll durchgeführt werden kann, wird durch das Model-Checking in Kapitel 11 erfolgen.

8.1 Hazards $\mathrm{LV}(F, S)$

1a, 1f Der Zug fährt nicht auf Sicht, obwohl er auf Sicht fahren sollte.

Unter Ausschluss menschlicher Fehler (wie in vorherigen Kapiteln seien perfekte Roboter vorausgesetzt, da hier nur das Verfahren analysiert wird, und keine Risikoabschätzung gemacht werden soll) kann dieser Zustand nicht eintreten, da das auf Ebene 3 beschriebene Verfahren die Sicherheitsaxiome der Ebene 0 erfüllt, die dies ausschließen.

1g Ein anderer Zug als vorgesehen fährt unter lokaler Verantwortung.

Naiv könnte dies implizieren, dass der Zug, der eigentlich unter lokaler Verantwortung hätte fahren sollen, dies nicht tut. Die Sicherheitsaxiome und das Verfahren lassen dies jedoch nicht zu.

1i Der Zug sollte ab einer bestimmten Stelle im Abschnitt unter lokaler Verantwortung fahren, fährt aber erst ab einer späteren Stelle unter lokaler Verantwortung.

Da die Sicherheitsaxiome der Ebene 0 aber sicherstellen, dass nie ein weiterer Zug im Abschnitt sein kann, wenn *nicht* auf Sicht gefahren wird, kann auch dies nicht eintreten.

8.2 Hazards $\mathrm{inA}(F, S)$

2d Ein weiterer Zug ist im Abschnitt, zusätzlich zu dem betrachteten Zug.

Durch die Sicherheitsaxiome der Ebene 0 ist sichergestellt, dass dieser Fall nicht eintreten kann.

2e Nur ein Teil des Zuges ist im Abschnitt, das heißt, es hat eine Zugtrennung stattgefunden.

Dieser Fall wird dadurch vermieden, dass die Bedingung für die Ankunftmeldung beinhaltet, dass das Eintreffen mit Zugschluss festgestellt wurde.

Dies wird im Rahmen dieser Arbeit nicht explizit behandelt, da es nicht spezifisch für das Protokoll ist.

2f Zug fährt im Abschnitt in die falsche Richtung

Wenn der Zug nach Befolgung des Verfahrens im Abschnitt ist, dann ist er dort der einzige, oder er fährt auf Sicht, daher stellt auch dies keinen Hazard dar.

2g Ein anderer Zug als vorgesehen fährt in den Abschnitt

Dies kann infolge einer verstümmelten Nachricht geschehen, doch werden diese Verstümmelungen durch das Nachrichtenprotokoll behoben, so dass auch hier kein Hazard vorliegt.

2h,2i,2j,2k All diese „Deviations“ beschreiben fehlerhafte Zeitabfolgen

Diese können bei Befolgung des vorgeschriebenen Verfahrens wie in Ebene 3 beschrieben, keinen Hazard darstellen, da stets sichergestellt ist, dass sich nur ein Zug im Abschnitt befindet.

8.3 Hazards ZV(F, S)

3a,3f Der Abschnitt ist nicht frei, obwohl er frei sein sollte.

Aufgrund des Betriebsverfahrens stellt dies keinen Hazard dar, da in einen nicht-freien Abschnitt kein Zug eingelassen wird.

8.4 Hazards inZ(F, A)

4e Nur ein Teil des Zuges ist in der Zuglaufmeldestelle

Wird verhindert durch die Betriebsvorschriften, siehe auch 2e.

4h,4j,4k All diese „Deviations“ beschreiben fehlerhafte Zeitabfolgen

Unabhängig von Timing-Betrachtungen gewährleistet die Beschreibung des Verfahrens auf Ebene 3 auch hier stets einen sicheren Betrieb

8.5 Hazards FA(F, B)

5g Es wurde für eine andere als die im Fahrplan vorgesehene Zuglaufmeldestelle die Fahranfrage gestellt.

Dies wird durch das Kommunikationsprotokoll vermieden, da nach den Rational Cognitive Models stets alle an der Kommunikation beteiligten wissen, welche Zuglaufmeldestelle für einen spezifischen Zug die nächste ist.

5h,5j,5k All diese „Deviations“ beschreiben fehlerhafte Zeitabfolgen

Unabhängig von Timing-Betrachtungen gewährleistet die Beschreibung des Verfahrens auf Ebene 3 auch hier stets einen sicheren Betrieb

8.6 Hazards $\mathrm{FE}(F, B)$

6g Es wurde für eine andere als die im Fahrplan vorgesehene Zuglaufmeldestelle die Fahrerlaubnis erteilt

Dies wird durch das Kommunikationsprotokoll vermieden, da nach den Rational Cognitive Models stets alle an der Kommunikation beteiligten wissen, welche Zuglaufmeldestelle für einen spezifischen Zug die nächste ist.

6h,6j,6k All diese „Deviations“ beschreiben fehlerhafte Zeitabfolgen

Unabhängig von Timing-Betrachtungen gewährleistet die Beschreibung des Verfahrens auf Ebene 3 auch hier stets einen sicheren Betrieb

8.7 Hazards $\mathrm{AFE}(F, B)$

7a,7f Wider Erwarten wurde die Fahrerlaubnis nicht abgelehnt, oder die Fahrerlaubnis erteilt

Dies kann durch fehlerhaftes menschliches Verhalten geschehen, durch eine idealisierte Darstellung der handelnden Menschen wird dies ausgeschlossen. Dies ist nicht ein einfaches „Wegdefinieren“ des Problems, denn diese Fehlerquelle kann leicht durch eine Ersetzung der menschlichen Kommunikation durch eine computerbasierte Kommunikation beseitigt werden.

7i,7j,7k All diese „Deviations“ beschreiben fehlerhafte Zeitabfolgen

Unabhängig von Timing-Betrachtungen gewährleistet die Beschreibung des Verfahrens auf Ebene 3 auch hier stets einen sicheren Betrieb

8.8 Hazards $\mathrm{FE}(F, B)$

8f,8k Dies stellt einen echten Hazard dar, der sich allerdings nicht durch ein Betriebsverfahren vollständig ausschließen läßt, der er auch Ereignisse höherer Gewalt einschließt, wie beispielsweise einen Baum, der vor dem Zug auf die Strecke stürzt.

8.9 Hazards $\mathrm{Next}(F, A) = B$

—

Kapitel 9

Message Flow Graphs

9.1 Message Flow Graphs vs. Message Sequence Charts

Die hier verwendeten Diagramme zur Veranschaulichung der Kommunikation zwischen einzelnen Zustandsautomaten. Jeder Teilnehmer am Verfahren, und damit an der Kommunikation, ist ein eigener Zustandsautomat, der separat implementiert werden kann.

9.2 Zugleiter und ein Zug

9.2.1 Vertrauenswürdige Kommunikation

Wenn die Kommunikation vertrauenswürdig ist, das bedeutet, dass beispielsweise durch ein computerbasiertes Verfahren mit C oder SHA1-Checksum garantiert wird, dass Meldungen unverstümmelt übertragen werden.

9.2.2 Nicht vertrauenswürdige Kommunikation

Wenn die tatsächliche menschliche Kommunikation betrachtet wird mit der Möglichkeit von unerkannt verstümmelten Nachrichten, sieht das Diagramm stellenweise ein wenig anders aus. Siehe Abbildung 9.2.

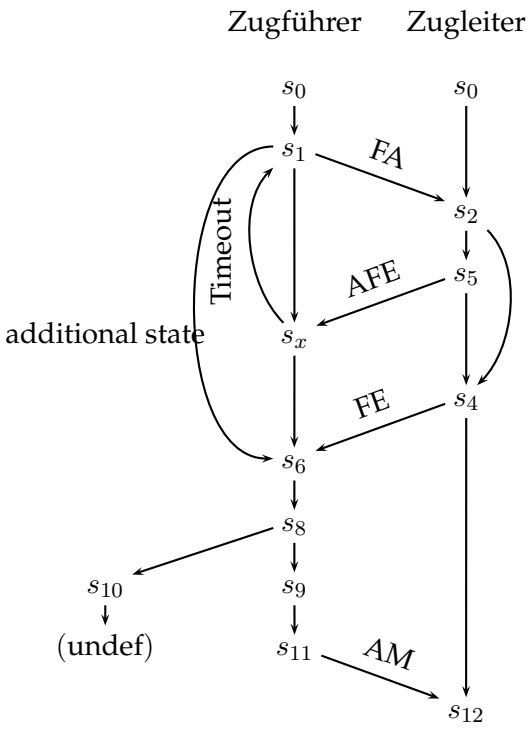


Abbildung 9.1: Message Flow Graph

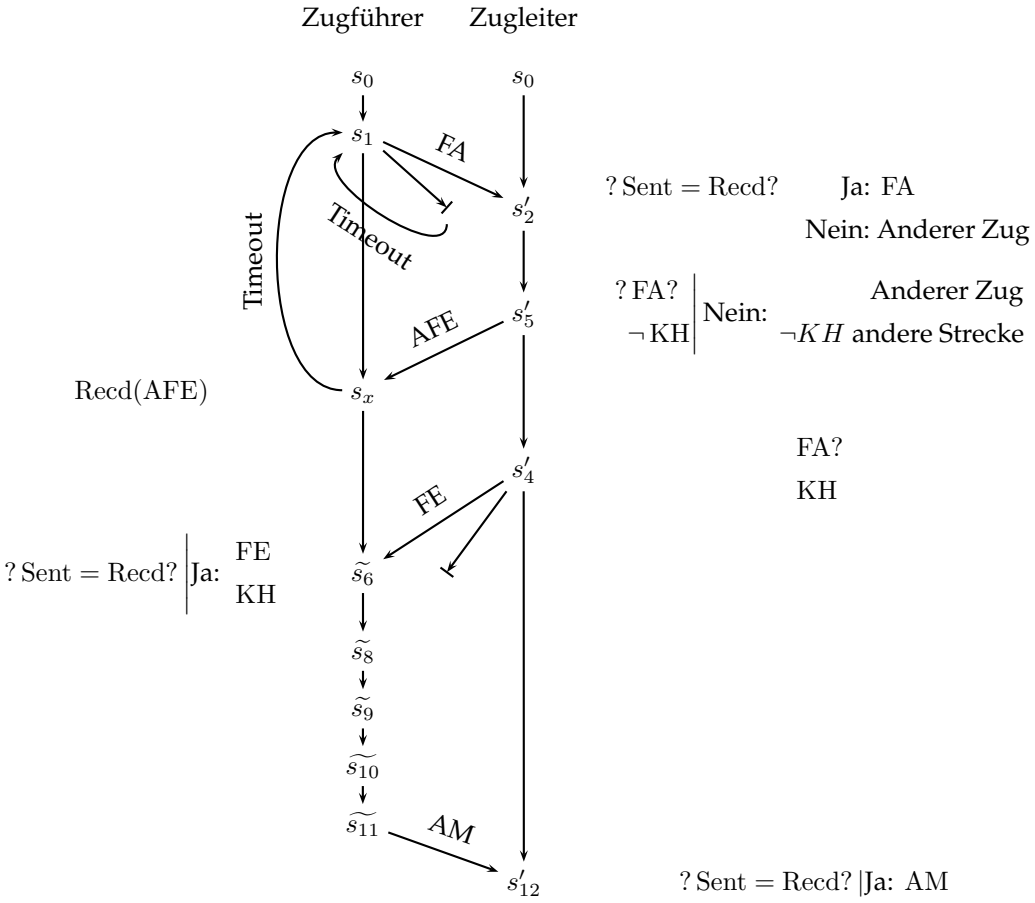


Abbildung 9.2: MFG mit unzuverlässiger Kommunikation

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.

9.3 Zwei Züge

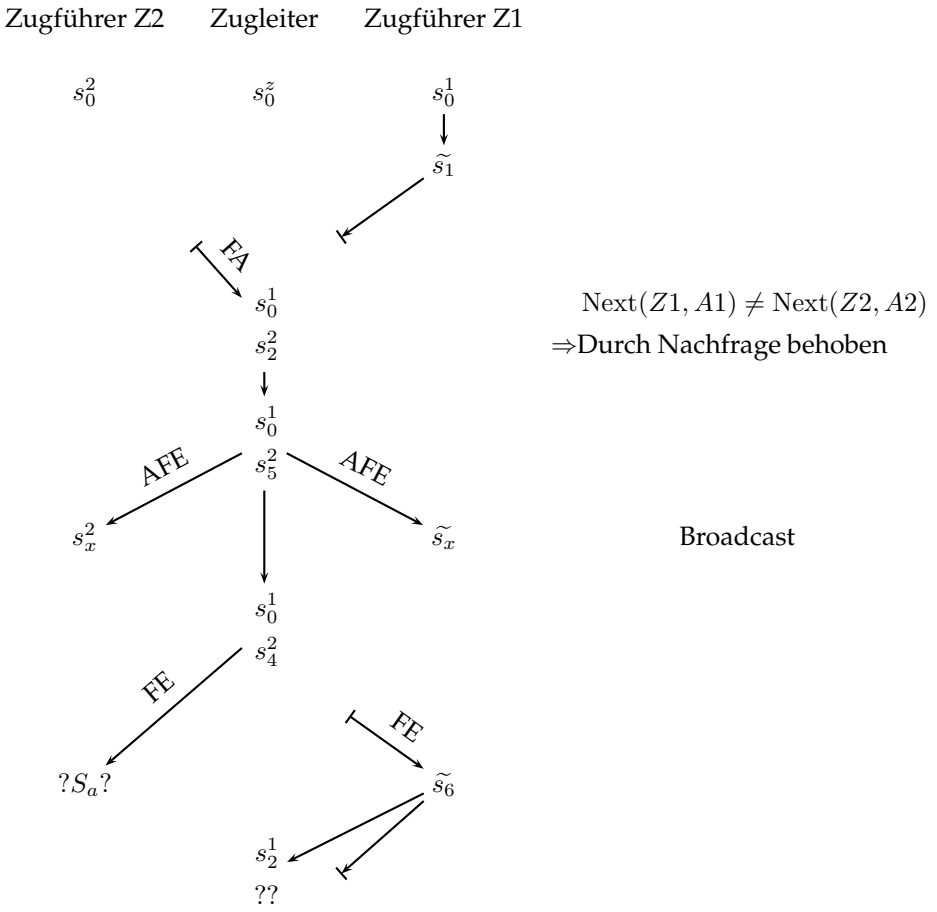


Abbildung 9.3: MFG mit zwei Zügen

Abbildung 9.3 zeigt den modifizierten Message Flow Graph für zwei Züge. Es zeigt sich, dass in diesem Falle ein neuer Hazard auftauchen kann für denn Fall, dass zwei verschiedene Züge dasselbe Ziel haben (d. h. $\mathrm{Next}(Z1, A1) = \mathrm{Next}(Z2, A2)$.

Es sei z. B. folgende Nachricht falsch übertragen worden:

$$\mathrm{Sent}(\mathrm{FE}, Z1, \mathrm{Next}(Z1, A1))$$
$$\mathrm{Recd}(\mathrm{FE}, Z2, \mathrm{Next}(Z2, A2))$$

im Zustand s_1^2. Falls dann gilt: $\neg\,\mathrm{KH}(Z2, \mathrm{Next}(Z2, A2))$, besteht ein Hazard, weil ein Zug in eine Strecke einfahren könnte, für die nicht festgestellt wurde, dass keine Hindernisse vorliegen.

Falls sich die beiden Ziele der Züge unterscheiden, wird dieser Konflikt durch die Nachfrage im Nachrichtenprotokoll behoben (wieder unter der Voraussetzung perfekter Roboter).

9.4 Entsprechung von MFG and PAD

Nach Ladkin/Leue[LL95] wird in Abbildung 9.4 gezeigt, dass der Message Flow Graph den Zustandsautomaten aus Ebene 3 erzeugt. Da in diesem Falle nur Kommunikation mit unverstümmeltem Nachrichteninhalt betrachtet wird, tauchen diejenigen Zustände, die nur bei einer verstümmelten Nachricht erreicht werden können, nicht auf. Dies sind s_3, s_7 und s_{13}.

9.5 Implementation, Test Harness

In Zusammenarbeit mit Praxis High Integrity Systems wurde eine erste Implementation realisiert, die in einer simulierten Umgebung getestet werden kann.

Eine spätere Version kann in separaten Computern implementiert werden, die auf einer Modelleisenbahntestanlage jeweils einen Zug steuern, bzw. die Rolle des Zugleiters übernehmen. Die Kommunikation erfolgt dabei mit geeigneten digitalen Verfahren, die eine Integrität der Nachrichten mit einer beliebig hohen Zuverlässigkeit gewährleisten können.

MFG-Trans.	Driver State	Controller State	Global State	PAD state
Initial State s_0	$\text{inZ}(F, A)$	—	$\text{inZ}(F, A)$	s_0
$s_0 \rightarrow s_1$	$\wedge \text{inZ}(F, A)$ $\wedge \text{Sent}\langle \text{FA}, F, \text{Next}(F, A)\rangle$	—	$\wedge \text{inZ}(F, A)$ $\wedge \text{Sent}\langle \text{FA}, F, \text{Next}(F, A)\rangle$	s_1
$s_1 \rightarrow s_2$	—	$\text{Recd}\langle \text{FA}, F, \text{Next}(F, A)\rangle$	$\wedge \text{inZ}(T, A)$ $\wedge \text{Sent}\langle \text{FA}, F, \text{Next}(F, A)\rangle$ $\wedge \text{Recd}\langle \text{FA}, F, \text{Next}(F, A)\rangle$	s_2
$s_2 \rightarrow s_5$	—	$\wedge\neg \text{KH}(F, A, \text{Next}(F, A))$ $\wedge \text{Sent}\langle \text{AFE}\rangle$	$\wedge \text{inZ}(F, A)$ $\wedge \text{FA}(F, A, \text{Next}(F, A))$ $\wedge\neg \text{KH}(F, A, \text{Next}(F, A))$ $\wedge \text{Sent}\langle \text{AFE}\rangle$	s_5
$s_2 \rightarrow s_4$	—	$\wedge \text{KH}(F, A, \text{Next}(F, A))$ $\wedge \text{Sent}\langle \text{FE}, F, \text{Next}(F, A)\rangle$	$\wedge \text{inZ}(F, A)$ $\wedge \text{FA}(F, A, \text{Next}(F, A))$ $\wedge \text{KH}(F, A, \text{Next}(F, A))$ $\wedge \text{Sent}\langle \text{FE}, F, \text{Next}(F, A)\rangle$	s_4
$s_4 \rightarrow s_6$	$\wedge \text{Recd}\langle \text{FE}, F, \text{Next}(F, A)\rangle$	—	$\wedge \text{inZ}(F, A)$ $\wedge \text{FA}(F, A, \text{Next}(F, A))$ $\wedge \text{KH}(F, A, \text{Next}(F, A))$ $\wedge \text{Sent}\langle \text{FE}, F, \text{Next}(F, A)\rangle$ $\wedge \text{Recd}\langle \text{FE}, F, \text{Next}(F, A)\rangle$	s_6
$s_6 \rightarrow s_8$	$\wedge \text{zw}(F, A, \text{Next}(F, A))$ $\wedge\neg \text{LV}(F, A)$	—	$\wedge \text{zw}(F, A, \text{Next}(F, A))$ $\wedge \text{FE}(F, A, \text{Next}(F, A))$ $\wedge \text{KH}(F, A, \text{Next}(F, A))$ $\wedge\neg \text{LV}(F, A)$	s_8
$s_8 \rightarrow s_9$	$\text{inZ}(F, \text{Next}(F, A))$	—	$\wedge \text{inZ}(F, \text{Next}(F, A))$ $\wedge \text{FE}(F, A, \text{Next}(F, A))$	s_9
$s_9 \rightarrow s_{11}$	$\wedge \text{Sent}\langle \text{AM}, F, \text{Next}(F, A)\rangle$	—	$\wedge \text{inZ}(F, \text{Next}(F, A))$ $\wedge \text{FE}(F, A, \text{Next}(F, A))$ $\wedge \text{Sent}\langle \text{AM}, F, \text{Next}(F, A)\rangle$	s_{11}
$s_{11} \rightarrow s_{12}$	—	$\wedge \text{Recd}\langle \text{AM}, F, \text{Next}(F, A)\rangle$	$\wedge \text{inZ}(F, \text{Next}(F, A))$ $\wedge \text{FE}(F, A, \text{Next}(F, A))$ $\wedge \text{Sent}\langle \text{AM}, F, \text{Next}(F, A)\rangle$ $\wedge \text{Recd}\langle \text{AM}, F, \text{Next}(F, A)\rangle$	s_{12}
$s_8 \rightarrow s_{10}$	$\wedge\neg \text{KH}(F, A, \text{Next}(F, A))$ $\wedge\neg \text{LV}(F, A)$	—	$\wedge \text{zw}(F, A, \text{Next}(F, A))$ $\wedge \text{FE}(F, A, \text{Next}(F, A))$ $\wedge\neg \text{KH}(F, A, \text{Next}(F, A))$ $\wedge\neg \text{LV}(F, A)$	s_{10}

Abbildung 9.4: MFG / PAD

Kapitel 10

Implementation in SPARK

10.1 SPARK

SPARK ist eine Programmiersprache und ein Spezifikationssystem [Bar03] 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 Praxis HIS entwickelte Software-Systeme haben die geringsten gemessenen Fehlerraten von im Einsatz befindlichen Software-Systemen.

10.2 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.

10.2.1 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 10.1: Beispiel einer Prozedur-Spezifikation in SPARK

Abbildung 10.1 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 (`*`) 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.

10.3 Zugführer

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

Abbildung 10.2 zeigt den Zusammenhang zwischen den Prozeduren des SPARK-Codes und den Übergängen im Zustandsautomaten des Fahrers.

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 10.2: Prozeduren für Zugführer

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.

Der hier behandelte SPARK-Code behandelt nicht den Übergang von s_8 nach s_{10}, weil dieser ein Erkennen eines Hindernisses während der Fahrt unter zentraler Verantwortung im Abschnitt beschreibt. Da dies nicht vom Computerprogramm erkannt werden kann, insbesondere nicht bei einer Modelleisenbahn-Simulationsanlage, kann dieser Fall nicht auftreten.

10.3.1 Die Beweis-Sprache FDL

Es werden hier vor allem die automatisch generierten Beweisfunktionen zu Record-Datentypen verwendet, die mit den Präfixen `fdl_`, `upf_`, und `mk_` beginnen.

Sei A eine Variable eines Record-Typs `T` und e ein Feld vom Typ `t`

dieses Record-Typs. Sei x eine Variable des Typs `t` und A' eine Variable des Record-Typs `T`, für die gilt: $A'.e = x$ und alle andere Felder f gilt: $A'.f = A.f$. So sind die Funktionen `fdl_e` und `upf_e` wie folgt definiert:

$$\texttt{fld\_e} : \texttt{T} \rightarrow \texttt{t}$$
$$\texttt{fld\_e}(A) = A.e$$

$$\texttt{upf\_e} : \texttt{T} \times \texttt{t} \rightarrow \texttt{T}$$
$$\texttt{upf\_e}(A, x) = A'$$

Die Tilde (˜)

Variablen, die in den Postconditions der SPARK-Annotationen mit einer Tilde versehen werden (z. B. `DS˜`), bezeichnen den ursprünglichen Wert der Variablen em vor der Ausführung der Prozedur. Unmarkierte Variablennamen bezeichnen den aktuellen Wert.

10.3.2 SPARK-Datenstrukturen

Zum Nachweis der Pre- und Postconditions muss ein Teil der in den SPARK-Programmen verwendeten Datenstrukturen näher betrachtet werden.

`Move_Data`

Dieser Datentyp ist ein Record und hat die beiden Felder `Active` vom Typ Boolean, und `Next_St` vom Typ `Track.Station_ID`.

`Driver_State`

Dieser Typ ist ein Record mit den Feldern `My_Unit` vom Typ `Trains.Train_ID`, `My_State` vom Typ `D_States` und `Move` vom Typ `Move_Data`.

`D_States` ist ein Aufzählungstyp, der die möglichen Zustände der Zugführer-Zustandsmachine des MFG enthält.

10.3.3 Meaning Postulates

$$\left.\begin{array}{l} \wedge\ \texttt{DS.MyUnit.ID == } F \\ \wedge\ \texttt{DS.MyData.At\_Station == } A \\ \wedge\ \texttt{DS.Move.active == true} \\ \wedge\ \texttt{DS.Move.Next\_St == } B \end{array}\right\} \Rightarrow \left\{\begin{array}{l} \wedge\ \neg\,\mathrm{inZ}(F, A) \\ \wedge\ \mathrm{zw}(F, A, B) \end{array}\right. \tag{10.1}$$

$$\left.\begin{array}{l} \wedge\ \texttt{DS.MyUnit.ID == } F \\ \wedge\ \texttt{DS.MyData.At\_Station == } A \\ \wedge\ \texttt{DS.Move.active == false} \end{array}\right\} \Rightarrow \mathrm{inZ}(F, A) \tag{10.2}$$

$$\left.\begin{array}{l} \wedge\ \texttt{DS.MyUnit.ID == } F \\ \wedge\ \texttt{DS.MyData.At\_Station == } A \\ \wedge\ \texttt{DS.Move.Next\_St == } B \end{array}\right\} \Rightarrow \mathrm{Next}(F, A) = B \tag{10.3}$$

`Send_FA`-Prozedur beendet $\Rightarrow \mathrm{Sent}(\mathrm{FA}, F, A)$ (10.4)

`Process_FE`-Prozedur beendet $\Rightarrow \mathrm{FE}(F, A)$ (10.5)

`Send_AM`-Prozedur beendet $\Rightarrow \mathrm{Sent}(\mathrm{AM}, F, A)$ (10.6)

10.3.4 Beweis

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. Die Spezifikationen und der SPARK-Quelltext, sowie die Beweis-Rule-Files (`.rlu` und die Zusammenfassung aller Beweisschritte (`Code.sum` sind im Anhang A zu finden.

Zu zeigen ist hier, dass die Pre- und Postconditions in den SPARK-Annotations sicherstellen, dass die Zustandsautomaten des MFG implementiert werden.

Sei F ein Zug, B und A Zuglaufmeldestellen. und seien `DS.MyUnit.ID =` F, `DS.MyData.At_Station =` A und `DS.Move.active = false`. Sei ferner `DS.Move.Next_St =` B

D. h. die Bedingungen am Beginn einer Strecke: Nach Meaning Postulates ist dann $\mathrm{inZ}(F, A)$, und $\mathrm{Next}(F, A) = B$.

Der SPARK-Code benutzt das Feld `My_State` in der Fahrer-Zustandsvariablen `DS`, deren Inhalt dem Zustand des Automaten auf dem MFG entspricht. Zu zeigen ist, dass die Veränderungen dieses Feldes jeweils genau dann stattfinden, wenn die Bedingungen, unter denen der Zustandsübergang im MFG-Automaten stattfindet, gegeben sind.

Dieses Feld Variable habe zu Beginn den Wert `D_S0`.

$s_0 \rightarrow s_1$:

Zu zeigen ist, dass dieser Übergang erfüllt wird von der Funktion `Send_FA` aus dem SPARK-Module `Drivers`.

Die Postcondition besagt, dass die Beweisfunktion `To_S1(DS~, DS)` nach Ende der Prozedur wahr ist. Laut Rule-File `drivers.rlu` kann die Funktion `To_S1(DS~, DS)` ersetzt werden durch `DS = upf_my_state(DS~,d_s1`. Nach Definition der `upf_`-Funktionen ist danach DS (Der Zustand des Zugführers nach der Prozedur) gleich dem Zustand vor der Prozedur, mit dem Unterschied, dass das Feld `My_State` nun den Wert `D_S1` hat.

Laut Meaning Postulate (10.4) ist nach Ausführen der Prozedur `Send_FA` $\mathrm{Sent}(\mathrm{FA}(Z, A, \mathrm{Next}(F, A))$ wahr, wie in Zustand $s1$ des Zugführer-Zustandsautomaten.

$s_1 \rightarrow s_x$:

Da die Nachricht der Ablehnung der Fahrerlaubnis (AFE) nicht die Zugnummer enthält, kann sie in jedem beliebigen Zustand eintreffen. Die Prozedur `Process_AFE` hat daher keine Precondition. Allerdings wird nach Definition des Zustandsautomaten die Meldung in allen Zuständen außer s_1 ignoriert.

Der Automat des Zugführers geht in den Zustand s_x über, wenn er sich vorher in s_1 befand, andernfalls passiert nichts. Der globale Zustand entspricht weiterhin s_5.

Die Beweisfunktion `D_State(DS)` in der Postcondition der Prozedur `Process_AFE` kann laut Rules-File `drivers.rlu` ersetzt werden durch `fld_my_state(DS)`, dies liefert den Wert der Feldes `My_State` des Zustands-Records `Driver_State` des betreffenden Zugführers. Wenn diese Zustandsvariable den Zustand `D_S1` liefert, verlangt die Postcondition, dass das Zustandsfeld `My_State` sich nach `D_Sx` ändert, falls `D_State` nicht `D_S1` liefert, soll der Zugführer-Zustand unverändert bleiben.

Durch das Empfangen der Ablehnung der Fahrerlaubnis ändert sich keines der Prädikate des globalen Predicate-Action-Diagrams, daher existiert für diese Prozedur auch kein Meaning Postulate.

$s_1 \rightarrow s_6$ **und** $s_x \rightarrow s_6$:

Da sich zwischen den Zuständen s_x und s_1 nur das Prädikat $\mathrm{AFE}(Z, A, \mathrm{Next}(Z, A))$ ändert, und dessen Wert für weitere Übergänge im Zustandsautomaten irrelevant ist, Kann der Übergang in den Zustand s_6 von beiden möglichen Vorgängerzuständen aus gemeinsam betrachtet werden.

Die Precondition der Prozedur `Process_FE` gewährleistet entsprechend, dass diese Prozedur nur ausgeführt werden kann, wenn das `D_State`-Feld des Zugführer-Zustandsrecords einen der Werte `D_S1` oder `D_Sx` hat.

Die Postcondition besagt, dass die Beweisfunktion `To_S6(DS~, DS)` nach Ende der Prozedur wahr ist. Laut Rule-File `drivers.rlu` kann die Funktion `To_S6(DS~, DS)` ersetzt werden durch `DS = upf_my_state(DS~,d_s6`. Nach Definition der `upf_`-Funktionen ist danach DS (der Zustand des Zugführers nach der Prozedur) gleich

dem Zustand vor der Prozedur, mit dem Unterschied, dass das Feld `My_State` nun den Wert `D_S6` hat.

Laut Tabelle 9.4 ist die Bedingung für den Übergang in Zustand $s6$: $\mathrm{Recd}(\mathrm{FE}, F, \mathrm{Next}(F, A))$. Da vor diesem Übergang global immer auch $\mathrm{Sent}(\mathrm{FE}, F, \mathrm{Next}(F, A))$ gelten muss, ist zusammen mit Meaning Postulate (M3.2) auch $\mathrm{FE}(F, A, \mathrm{Next}(F, A))$ wahr.

$s_6 \rightarrow s_8$:

Bei diesem Übergang wird das Prädikat $\mathrm{zw}(Z, A, \mathrm{Next}(Z, A))$ wahr, und das Prädikat $\mathrm{inZ}(Z, A)$ wird falsch.

Die Precondition der Prozedur `Start_Move` gewährleistet dass diese Prozedur nur ausgeführt werden kann, wenn das `My_State`-Feld des Zugführer-Zustandsrecords den Wert `D_S6` hat.

Im Rule-File `drivers.rlu` ist die Postcondition der Prozedur `Start_Move` so definiert, dass das Feld `My_State` nach der Prozedur den Wert `D_S8` hat, und außerdem das Feld `Active` im Record des Feldes `Move` der Zugführerzustandsvariablen den Wert `True` hat. Nach Meaning Postulate (10.1) und Definition des Automaten ist damit der Zugführerzustandsautomat in Zustand s_8.

$s_8 \rightarrow s_9$:

Bei diesem Übergang wird das Prädikat $\mathrm{inZ}(Z, \mathrm{Next}(Z, A))$ wahr, und das Prädikat $\mathrm{zw}(Z, A, \mathrm{Next}(Z, A))$ wird falsch.

Die Precondition der Prozedur `Check_End_Move` gewährleistet dass diese Prozedur nur ausgeführt werden kann, wenn das `My_State`-Feld des Zugführer-Zustandsrecords den Werte `D_S8` hat.

Die Postcondition der Prozedur `Check_End_Move` gewährleistet, dass sich an der Zugführerzustandsvariable nichts, wenn die Funktion `Completed_Move` des aktuellen Zustands und des Zugsensors falsch ist.

Andernfalls wird die Beweisfunktion `To_S9` wahr. Diese ist in `drivers.rlu` so definiert, dass folgende Felder der Zugführerzustandsvariablen aktualisiert werden: Der Wert der aktuellen Position im Feld At_Station wird auf den Wert des Feldes `Next_St` vor der Prozedur gesetzt und das Feld `active` in

`Move_Data` wird wahr; damit ist nach Meaning Postulate (10.2) das Prädikat $\mathrm{inZ}(F, \mathrm{Next}(Z, A))$ wahr. Das Feld `My_State` wird nach der Postcondition auf den Wert `D_S9` gesetzt.

$s_9 \rightarrow s_{11}$:

Laut der Precondition kann die Prozedur `Send_AM` nur aufgerufen werden, wenn das Feld `My_State` der Zugführerzustandsvariablen den Wert `D_S9` hat.

Laut der Postcondition der Prozedur hat das Feld `My_State` der Zugführerzustandsvariablen nach der Prozedur den Wert `D_S11`.

Laut Meaning Postulate (10.6) ist nach Ausführen der Prozedur `Send_AM` das Prädikat $\mathrm{Sent}\langle AM(Z, \mathrm{Next}(Z, A))\rangle$ wahr, was dem Zustand s_{11} im Zustandsautomaten entspricht.

10.4 Zugleiter

10.4.1 Verschiedene Züge

Das Predicate-Action-Diagram gilt jeweils für einen Zug und einen Zugleiter, d. h. bei mehreren Zügen gibt es eine getrennt zu untersuchende Zugleiterzustandsmaschine für jeden Zug.

Untersucht wird die Zugleiterzustandsmaschine für einen bestimmten, festen Zug Z.

10.4.2 Meaning Postulates

Für die Beweisfunktion `Postion(`Z`)` aus `control.rlu` gilt: Sei Z ein Zug und A eine Zuglaufmeldestelle, so gilt:

$$\texttt{Position(}Z\texttt{)} = A \Rightarrow \mathrm{inZ}(Z, A) \tag{10.7}$$

Für die Beweisfunktion `Next(`Z`)` aus `control.rlu` gilt: Sei Z ein Zug und A und B Zuglaufmeldestelle, so gilt:

$$\texttt{Next(}Z\texttt{) = }B \wedge \texttt{Position(}Z\texttt{) = }A \Rightarrow \mathrm{Next}(Z, A) = B \quad (10.8)$$

Für die Funktion `KH(`Z`)` aus `control.rlu` gilt: Sei Z ein Zug und A und B Zuglaufmeldestellen, so gilt:

$$\left.\begin{array}{l} \wedge\ \texttt{KH(}Z\texttt{) = True} \\ \wedge\ \texttt{Position(}Z\texttt{) = }A \\ \wedge\ \texttt{Next(}Z\texttt{) = }B \end{array}\right\} \Rightarrow \mathrm{KH}(Z, A, B) \quad (10.9)$$

10.4.3 Beweis

$s_0 \rightarrow s_2$:

In der Postcondition kann `To_S2` nach den Regeln der in `control.rlu` so ersetzt werden, dass des Feld `current_state` der Zustandsvariable des Zugleiters für den Zug Z den Wert `C_S2` annimmt. Nach Meaning Postulate wird durch das Ausführen der Prozedur `Process_FA` das Prädikat $\mathrm{Recd}\langle FA, Z, \mathrm{Next}(Z, A)\rangle$ wahr. Dies entspricht dem Übergang der Zugleiterzustandsmaschine in den Zustand s_2.

(Der Zustand der Zugleiterzustandsmaschinen für alle anderen Züge bleibt unverändert.)

$s_2 \rightarrow s_4$:

Laut Precondition kann die Prozedur `Check_Waiting` in `control.ads` im Zustand s_2 oder s_5 aufgerufen werden.

In der Postcondition wird je nach Anfangszustand unterschieden: Wenn der Anfangszustand s_2 war, und `KH(`Z`)` wahr ist, dann impliziert dies laut Postcondition, dass die Beweisfunktion `To_S4` wahr ist.

In der Postcondition kann `To_S4` nach den Regeln der in `control.rlu` so ersetzt werden, dass des Feld `current_state` der Zustandsvariable des Zugleiters für den Zug Z den Wert `C_S4` annimmt, und damit der Zugleiterzustandsautomat für den Zug Z in Zustand s_4 ist.

$s_2 \rightarrow s_5$:

Laut Precondition kann die Prozedur `Check_Waiting` in `control.ads` im Zustand s_2 oder s_5 aufgerufen werden.

In der Postcondition wird je nach Anfangszustand unterschieden: Wenn der Anfangszustand s_2 war, und `KH(`Z`)` falsch ist, dann impliziert dies laut Postcondition, dass die Beweisfunktion `To_S5` wahr ist.

In der Postcondition kann `To_S5` nach den Regeln der in `control.rlu` so ersetzt werden, dass des Feld `current_state` der Zustandsvariable des Zugleiters für den Zug Z den Wert `C_S5` annimmt, und damit der Zugleiterzustandsautomat für den Zug Z in Zustand s_5 ist.

$s_5 \rightarrow s_4$:

Laut Precondition kann die Prozedur `Check_Waiting` in `control.ads` im Zustand s_2 oder s_5 aufgerufen werden.

In der Postcondition wird je nach Anfangszustand unterschieden: Wenn der Anfangszustand s_5 war, und `KH(`Z`)` wahr ist, dann impliziert dies laut Postcondition, dass die Beweisfunktion `To_S4` wahr ist.

In der Postcondition kann `To_S4` nach den Regeln der in `control.rlu` so ersetzt werden, dass des Feld `current_state` der Zustandsvariable des Zugleiters für den Zug Z den Wert `C_S5` annimmt, und damit der Zugleiterzustandsautomat für den Zug Z in Zustand s_4 ist.

$s_4 \rightarrow s_{12}$:

Laut Precondition kann die Prozedur `Process_AM` in `control.ads` nur aufgerufen werden, wenn sich der Automat in Zustand s_4.

In der Postcondition kann `To_S12` nach den Regeln der in `control.rlu` so ersetzt werden, dass des Feld `current_state` der Zustandsvariable des Zugleiters für den Zug Z den Wert `C_S12` annimmt, und damit der Zugleiterzustandsautomat für den Zug Z in Zustand s_{12} ist.

Kapitel 11

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[Hol03] 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.

11.1 Der SPIN-Model-Checker

11.1.1 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 [HS00], Betriebssysteme [DJ95], und auch bereits für Bahnsysteme [GLL+00]. Das Softwaretool prüft logische Konsistenz und meldet Deadlocks, Race Conditions und andere Probleme.

SPIN kann die spezifizierten Algorithmen anhand von Korrektheits-Anforderungen in linear-lime temporal logic (LTL) testen, und kann partielle und vollständige Beweise liefern durch zufällige (random), interaktive oder erschöpfende Überprüfung.

SPIN kann verwendet werden als

- **Simulator**, in dem zum *Rapid Prototyping* zufällige, geleitete (*guided*) oder interaktive Simulationen durchgeführt werden
- **Verifier**, der durch erschöpfende Tests die Korrektheit des Systems anhand der Requirements rigoros beweisen kann.
- **Beweisapproximationssystem**, das auch sehr große Systeme durch größtmögliche Ausschöpfung des Zustandsraumes validieren kann.

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

11.1.2 Die Modellierungssprache PROMELA

PROMELA (PROcess MEta Language) ist eine nicht-deterministische Sprache, ähnlich Dijkstras *guarded command language* [Dij75], erweitert um die Ein-/Ausgabefunktionen von Hoares CSP-Sprache [Hoa78].

11.2 LTL

Linear Temporal Logic (oder *Linear Time Temporal Logic* ist eine Modallogik,

in der sich die zusätzlichen Modalitäten *möglich* und *notwendig* auf die Zeit beziehen. Üblicherweise wird *notwendigerweise* dabei als *immer*, und *möglicherweise* als *schließlich* (engl.: *finally/eventually*.

Die modalen Operatoren der LTL sind:

Nϕ (auch: $\circ\phi$): ϕ ist im nächsten Zustand wahr. Dieser Operator ist nicht stotterinvariant, daher wird er in dieser Betrachtung nicht verwendet. (*Next*) Siehe hierzu auch [Lam83], warum Lamport ausdrücklich bewusst auf die Einführung eines *Next*-Operators verzichtet hat.

Auch SPIN kennt den *Next*-Operator nur dann, wenn er als Nicht-Standard-Option bei der Übersetzung des Quelltextes aktiviert wird.

Gϕ (auch: $\Box\phi$): ϕ ist immer wahr. (*globally/always*)

Fϕ (auch: $\Diamond\phi$): ϕ wird irgendwann in der Zukunft wahr. (*Finally/Eventually*)

ϕUψ (auch: $\phi\mathcal{U}\psi$): Ab dem aktuellen oder einem späteren Zeitpunkt ist ψ wahr, bis dahin ist ϕ wahr. Ab dann muss ϕ nicht mehr wahr sein. (*Until*)

ϕRψ (auch: $\phi\mathcal{R}\psi$): ψ ist wahr bis zu dem ersten Zeitpunkt, an dem ϕ wahr ist (oder für immer, falls ϕ nie wahr ist.) (*Release*)

Wichtige Anwendung für die F- und G-Operatoren sind Sicherheits- und Liveness-Bedingungen.

In [LO82] werden folgende Beschreibungen für Sicherheits- und Liveness-Anforderungen vorgeschlagen:

Safety: *etwas Schlechtes passiert* nie, d. h. $\Box\neg B$, wenn B einen Zustand darstellt, der nicht eintreten darf.

Liveness-Bedingungen haben demnach die Form *etwas Gutes passiert* irgendwann, d. h. $\Diamond A$, wenn A einen gewünschten Zustand darstellt.

LTL kann keine Echtzeitabläufe darstellen.

11.3 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.

Die ursprüngliche Idee war, ein an die SPARK-Implementation von Thornley angelehntes Modell zu erstellen:

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.

11.3.1 Zustände und Globale Variablen

Entsprechend der SPARK-Implementation wird beim Model-Checking hier nur zuverlässige Kommunikation betrachtet. SPIN bietet sowohl synchrone als auch asynchrone (gepufferte) Kommunikation. 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.

Zur Initialisierung wird jedem Zug ein Start- und ein Zielbahnhof zugewiesen, und eine Fahrtrichtung. Daraus werden vor Beginn des eigentlichen Model-Checking die jeweils nächsten Bahnhöfe der Strecke ($\mathrm{Next}(F, A)$) erzeugt und in einem Feld abgelegt. Die Abstraktion des System in dieser Ontological Hazard Analysis beschreibt perfekte Roboter, dazu gehört, dass alle Züge und der Zugleiter immer wissen, zu welcher Station sie als nächstes fahren müssen.

Wie in 9.4 gezeigt, findet der Übergang in den jeweiligen Zustand 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 11.11 anhand der von SPIN erzeugten und ausgegebenen Zustandsmaschinen statt.

11.3.2 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.

11.3.3 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.

- Kreuzungen und Überholungen

  Durch das vereinfachte System mit Rückfallweichen sind keine Überholungen möglich. Kreuzungen sind möglich.

- 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.

11.3.4 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.

11.4 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.

11.4.1 Büchi-Automaten

SPIN verwendet für die Assertions Bedingungen in Linear (Time) Temporal Logic (LTL), die von einem zusätzlichen Tool in einen Büchi-Automaten transformiert werden. Dieser Büchi-Automat hat die Form eines *Never-Claims* in der Sprache PROMELA.

In diesem speziellen Fall sind für die Erfüllung der Liveness-Properties jedoch keine Never-Claims erforderlich. Alle PROMELA-Prozesse gehen nur dann in ihren *valid end state* über, wenn die entsprechenden Züge ihr vorgesehenes Ziel erreicht haben. Spin betrachtet den Zustand, den ein Automat beziehungsweise ein Prozess erreicht hat, dann als gültig, wenn das Ende des Codes erreicht wurde. Durch die Implementation des Modells ist dies für Züge genau dann der Fall, wenn die letzte Zuglaufmeldestelle erreicht und die Ankunftmeldung von dieser Zuglaufmeldestelle abgeschickt wurde. Für den Zugleiter ist dies der Fall, wenn die Ankunftmeldung von der letzten Zuglaufmeldestelle im vorgesehenen Zuglauf empfangen wurde.

11.5 Kreisstrecke

Nach der Erstellung des PROMELA-Modells, mehrerer Testläufe und der Erzeugung des Verifiers stellte sich heraus, dass eine vollständige Verifikation des implementierten Modells mit Kreisstrecke nur für eine sehr kleine Anzahl Stationen und Züge durchführbar war. Lediglich bei 2 Zügen und 3 Stationen war eine erschöpfende Durchsuchung des Zustandsraumes möglich. Auf der zur Verfügung stehenden Hardware mussten Verifikationsläufe mit 4 Stationen und 3 Zügen aufgrund erheblich zu hohen Speicherbedarfs abgebrochen werden.

Eine der Ursachen für die Zustandsexplosion ist die Tatsache, dass bei einem Test aller möglichen Verteilungen von Zugfahrten von 3 Zügen der weitaus größte Teil der Zustände, die durch das Protokoll aufgelöst werden müssen, äquivalent zu anderen Zuständen sind. Es gibt nur eine kleine Anzahl wirklich verschiedener Situationen. Diese mehrfachen Situationen werden jedoch vom Model-Checker nicht erkannt und eliminiert.

Eine explizite Auflistung aller möglichen Situationen mit vier Zügen verspricht mehr Erfolg, wenn man manuell alle Wiederholungen und Spiegelungen ausschließt, und die Startpositionen und Richtungen der Zugfahrten explizit auflistet.

11.6 Lineare Strecke

Das PROMELA-File *Four-Train-Situations* im Anhang zeigt den Source-Code dieses zweiten Verfahrens. Die eigentlichen Kernprozesse des Ablaufs sind weitgehend identisch mit denen des ersten Versuchs.

Alle möglichen Ausgangssituationen mit 4 Zügen sollen im Model-Checker getestet werden. Die Bedingungen sind dabei so anzunehmen wie bei der Kreisstrecke: 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.

11.6.1 Situationen

Tabelle 11.1 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.

Bemerkungen zu Tabelle 11.1

1) Dieser Fall muss nicht betrachtet werden, da eine Rahmenbedingung festlegt, dass 2 Züge in derselben Station stets unterschiedliche Fahrtrichtungen haben müssen. Einer der beiden Züge würde daher bereits seinen Bestimmungsort erreicht haben, und aus der Betrachtung herausfallen. Da der Model-Checker eine erschöpfende Zustandsraumsuche durchführt, wird dieser Fall mit 3 Zügen bereits von anderen Plazierungsfällen abgedeckt, beispielsweise wird Fall A bereits abgedeckt von Fall G, wenn in dem Fall Zug 3 nach rechts aus dem betrachteten Abschnitt ausfährt, bevor andere Züge behandelt werden.

2) Fall L ist symmetrisch zu Fall H, und muss daher nicht betrachtet werden.

3) Fall M ist symmetrisch zu Fall D, und muss daher nicht betrachtet werden.

4) Fall P ist symmetrisch zu Fall K, und muss daher nicht betrachtet werden.

5) Fall R ist symmetrisch zu Fall I, und muss daher nicht betrachtet werden.

Stationen	0	1	2	3	4	Bemerkungen
Situation						
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 11.1: Liste aller möglichen Ausgangssituationen für 4 Züge und 5 Stationen

6) Fall S ist symmetrisch zu Fall O, und muss daher nicht betrachtet werden.

7) Fall T ist symmetrisch zu Fall G, und muss daher nicht betrachtet werden.

8) Fall U ist symmetrisch zu Fall C, und muss daher nicht betrachtet werden.

9) Fall V ist symmetrisch zu Fall N, und muss daher nicht betrachtet werden.

10) Fall W ist symmetrisch zu Fall F, und muss daher nicht betrachtet werden.

11) Fall X ist symmetrisch zu Fall B, und muss daher nicht betrachtet werden.

12) Fall Y ist symmetrisch zu Fall A, und muss daher nicht betrachtet werden.

Das Model-Checking ist daher nur noch für die in Tabelle 11.2 durchzuführen. Um eine unnötige Komplizierung des PROMELA-Codes und damit eine Explosion des Speicherbedarfs zu vermeiden, werden die jeweiligen Variablen für jeweils einen Fall in einem getrennten Source-File initialisiert, das jeweils für einen Durchlauf des Model-Checkers importiert wird. Mit dieser Methode ist es möglich, in vertretbarer Zeit für alle 10 unterschiedlichen Fälle eine erschöpfende Zustandsraumsuche durchzuführen. Hierzu wurde für alle diese zu untersuchenden Fälle ein entsprechendes zu importierendes File geschrieben (Abb. 11.6.1). Shell-Skript-gesteuert werden nacheinander diese Files importiert, jeweils der Verifier daraus erzeugt und gestartet. Die Ausgaben des Verifiers werden für jeden Durchlauf in einem gesonderten Protokollfile abgelegt (Abb. 11.5).

11.7 Zustandsmaschinen

Der Model-Checker SPIN erzeugt aus dem PROMELA-Code kommunizierende Zustandsmaschinen. Der Verifier, der aus dem Code erzeugt

```
/* 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 11.1: Beispiel eines Include-Files für den Model-Checkers

wird, kann diese State-Machine nach den ersten Optimierungsschritten ausgeben. Der wichtigste dieser Optimierungsschritte ist Partial Order Reduction [HP95] mit *statement merging* [Hol99]. 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 aller möglichen globalen Zustände würde den Rahmen heute für private Anwendung üblicher Computer sprengen. Daher verwende ich in diesem Fall Graph-Encoding [Gre96], 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 11.2 und 11.3 zeigen die Zustandsautomaten, die SPIN aus den jeweiligen Prozessbeschreibungen erzeugt hat, und die für die Verifikation verwendet werden. Die Nummern in den Zustandsknoten sind

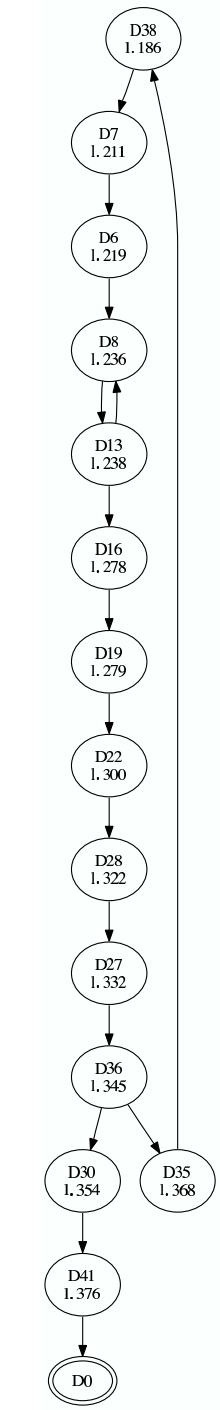


Abbildung 11.2: Reduzierte Zustandsmaschine des PROMELA-Prozesses für den Zugführer

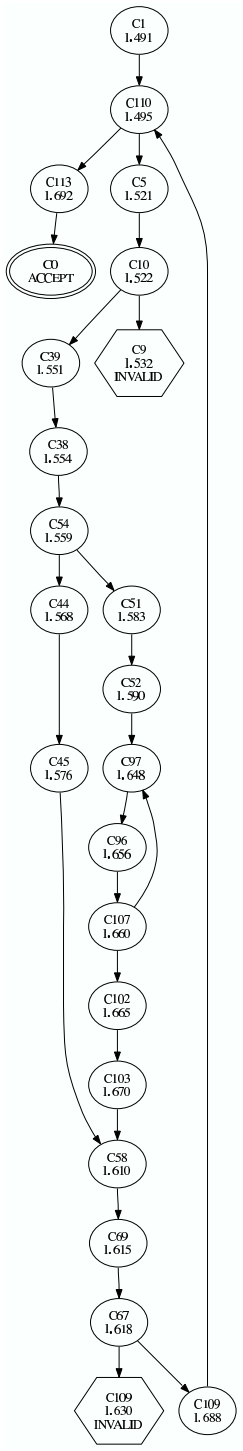


Abbildung 11.3: Reduzierte Zustandsmaschine des PROMELA-Prozesses für den Zugleiter

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

vom Model-Checker vergeben, darunter steht jeweils die Nummer der Code-Zeile im PROMELA-File des Zustands.

11.8 Simulation

Während der Entwicklung des PROMELA-MODELLS aus den kommunizierenden Automaten des Message-Flow-Graphs wurden zur schnellen Überprüfung auf offensichtliche Implementationsfehler Simulationsläufe mit SPIN durchgeführt. Um zu veranschaulichen, wie die Kommunikation zwischen den Prozessen abläuft, kann bei einem solchen Simulation-

slauf, der nur einen einzigen der möglichen Pfade durch den globalen Zustandsautomaten durchläuft, kann SPIN ein einfaches Message-Sequence-Chart in Postscript ausgeben. Abbildung 11.4 zeigt einen Ausschnitt einer solchen Ausgabe.

Die Einzelnen Prozesse sind dabei als senkrechte Linien dargestellt, Senden und Empfangen von Nachrichten, sowie benutzerdefinierte Ausgaben durch `printf`-statements als Rechtecke, die Nachrichtenübermittlung durch diagonale Linien.

Die Texte der Form `t2c[`n`]!tFA,`t`,`d bezeichnen das Senden der Nachricht (hier `tFA`: Fahranfrage) über den Kanal `t2c[`n`]` (hier: train-to-controller, also Zugführer an Zugleiter, Nummer n), betreffend Zug Nummer t für die Fahrt zur Zuglaufmeldestelle d.

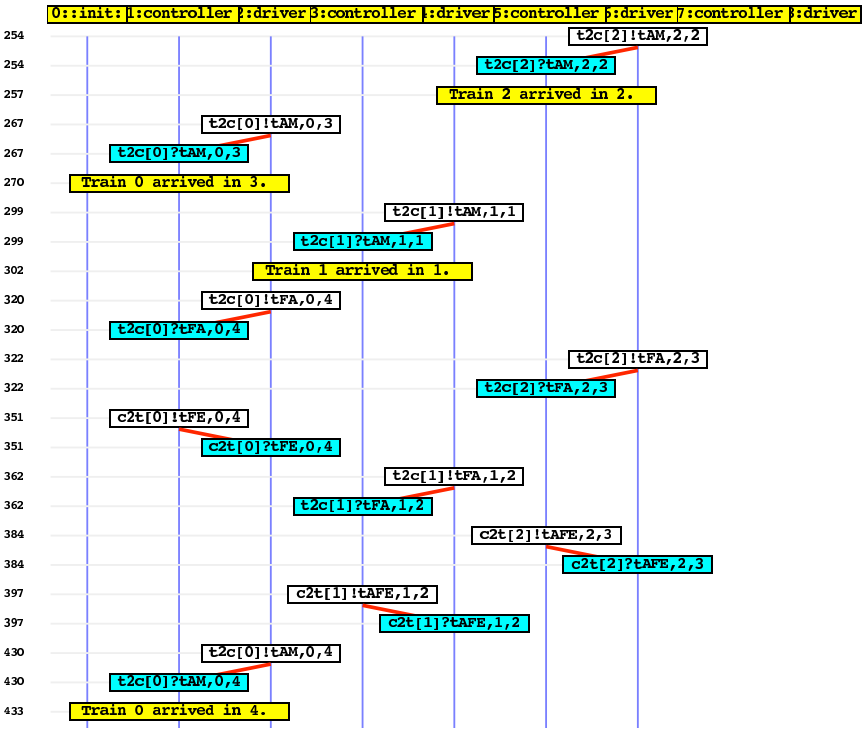


Abbildung 11.4: Ausschnitt aus einem von SPIN generierten Message-Sequence-Chart

11.9 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.

```
 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 11.5: Ausgabe eines SPIN-Verifier-Durchlaufs

Abbildung 11.5 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

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[BSD09] vertretbar.

Tabelle 11.3 zeigt die jeweilige Größe des überprüften globalen Zustandsautomaten, sowie die für die Verifikation benötigten Ressourcen und die erzielte Testrate in Übergängen pro Sekunde.

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 11.3: Benötigte Ressourcen für die Verifikation

11.10 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.

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.

$$\left.\begin{array}{l} \wedge\ \texttt{t2c[}F\texttt{]!ZLM} \\ \wedge\ \texttt{ZLM.type == FA} \\ \wedge\ \texttt{ZLM.train == } F \\ \wedge\ \texttt{ZLM.destination == } A \end{array}\right\} \Rightarrow \mathrm{Sent}(\mathrm{FA}, F, A) \tag{11.1}$$

$$\left.\begin{array}{l} \wedge\ \texttt{c2t[}F\texttt{]!ZLM} \\ \wedge\ \texttt{ZLM.type == FE} \\ \wedge\ \texttt{ZLM.train == } F \\ \wedge\ \texttt{ZLM.destination == } A \end{array}\right\} \Rightarrow \mathrm{Sent}(\mathrm{FE}, F, A) \tag{11.2}$$

$$\left.\begin{array}{l} \wedge\ \texttt{c2t[}F\texttt{]!ZLM} \\ \wedge\ \texttt{ZLM.type == AFE} \\ \wedge\ \texttt{ZLM.train == } F \\ \wedge\ \texttt{ZLM.destination == } A \end{array}\right\} \Rightarrow \mathrm{Sent}(\mathrm{AFE}, F, A) \qquad (11.3)$$

$$\left.\begin{array}{l} \wedge\ \texttt{t2c[}F\texttt{]!ZLM} \\ \wedge\ \texttt{ZLM.type == AM} \\ \wedge\ \texttt{ZLM.train == } F \\ \wedge\ \texttt{ZLM.destination == } A \end{array}\right\} \Rightarrow \mathrm{Sent}(\mathrm{AM}, F, A) \qquad (11.4)$$

$$\left.\begin{array}{l} \wedge\ \texttt{t2c[}F\texttt{]?ZLM} \\ \wedge\ \texttt{ZLM.type == FA} \\ \wedge\ \texttt{ZLM.train == } F \\ \wedge\ \texttt{ZLM.destination == } A \end{array}\right\} \Rightarrow \mathrm{Recd}(\mathrm{FA}, F, A) \qquad (11.5)$$

$$\left.\begin{array}{l} \wedge\ \texttt{c2t[}F\texttt{]?ZLM} \\ \wedge\ \texttt{ZLM.type == FE} \\ \wedge\ \texttt{ZLM.train == } F \\ \wedge\ \texttt{ZLM.destination == } A \end{array}\right\} \Rightarrow \mathrm{Recd}(\mathrm{FE}, F, A) \qquad (11.6)$$

$$\left.\begin{array}{l} \wedge\ \texttt{c2t[}F\texttt{]?ZLM} \\ \wedge\ \texttt{ZLM.type == AFE} \\ \wedge\ \texttt{ZLM.train == } F \\ \wedge\ \texttt{ZLM.destination == } A \end{array}\right\} \Rightarrow \mathrm{Recd}(\mathrm{AFE}, F, A) \qquad (11.7)$$

$$\left.\begin{array}{l} \wedge\ \texttt{t2c[}F\texttt{]?ZLM} \\ \wedge\ \texttt{ZLM.type == AM} \\ \wedge\ \texttt{ZLM.train == } F \\ \wedge\ \texttt{ZLM.destination == } A \end{array}\right\} \Rightarrow \mathrm{Recd}(\mathrm{AM}, F, A) \qquad (11.8)$$

$$\texttt{inZ[}F\texttt{].station[}A\texttt{] == true} \Rightarrow \mathrm{inZ}(F, A) \tag{11.9}$$

$$\left.\begin{array}{l}\text{Sei } n \text{ Anzahl der Züge:}\\ \text{Sei } F_i, i \in \{0, \cdots, n-1\}, \text{ ein Zug:}\\ \forall F_k, k \in \{0, \ldots, n-1\}, k \neq i:\\ \wedge\texttt{inZ[}F_i\texttt{].station[}A\texttt{] == true}\\ \wedge\texttt{ZV[}F_k\texttt{].previous[}A\texttt{].next[Next[}F_i\texttt{]] == false}\\ \wedge\texttt{ZV[}F_k\texttt{].previous[Next[}F_i\texttt{]].next[}A\texttt{] == false}\\ \wedge(\vee\ \texttt{inZ[}F_k\texttt{].station[Next[}F_i\texttt{]] == false}\\ \quad\vee\ \texttt{Next[}F_k\texttt{] == }A)\end{array}\right\} \Rightarrow \mathrm{KH}(F, A) \tag{11.10}$$

$$\texttt{zw[}F\texttt{].previous[}A\texttt{].next[}b\texttt{]} \Rightarrow \mathrm{zw}(F, A, B) \tag{11.11}$$

$$\texttt{ZV[}F\texttt{].previous[}A\texttt{].next[}b\texttt{]} \Rightarrow \mathrm{ZV}(F, A, B) \tag{11.12}$$

$$\left.\begin{array}{l}\wedge\ \texttt{inZ[}F\texttt{].previous[}A\texttt{].next[}B\texttt{] == true}\\ \wedge\ \texttt{ZV[}F\texttt{].previous[}A\texttt{].next[}B\texttt{] == false}\end{array}\right\} \Rightarrow \mathrm{LV}(F, A, B) \tag{11.13}$$

11.11 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.

11.11.1 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;
```

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

D19 → D22

```
d_step {
  inZ[train].station[Current[train]]=false;
  zw[train].previous[Current[train]].next[Next[train]]=true;
} /* end d_step */
```

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

Da die beiden Statements in dieser Transition innerhalb von einem D_STEP-Konstrukt ausgeführt werden, kann hier keine race condition entstehen, da kein anderer Prozess zwischendurch aktiv werden kann.

D22 → D28

```
d_step {
  zw[train].previous[Current[train]].next[Next[train]]=false;
  inZ[train].station[Next[train]] = true;
} /* end d_step */
```

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

D28 → D27

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

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

D27 → D36

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

Statement	inA	inB	zw	S:FA	R:FE	R:AFE	S:AM
Initial	-	✓	-	✓	✓	-	-
t2c!ZLM	-	✓	-	✓	✓	-	✓

Alle weiteren Übergänge dienen dem Neusetzen der Parameter für den nächsten Teilabschnitt, und werden im MFG nicht abgebildet.

11.11.2 Zugleiter

```
t2c[train]?ZLM;
```

C5 → C10

Statement	ZV	R:FA	S:FE	S:AFE	R:AM
Initial	-	-	-	-	-
t2c?ZLM	-	-	-	-	-

Die Nachricht wurde noch nicht ausgewertet, daher ist der Inhalt noch nicht bekannt, und die entsprechenden Variablen noch nicht gesetzt. An dieser Stelle darf keine andere Nachricht als die Fahranfrage empfangen werden. Der Verifikationslauf des Model-Checkers hat gezeigt, dass dies im entwickelten Protokoll nie auftritt.

C10 → C39

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

Statement	ZV	R:FA	S:FE	S:AFE	R:AM
Initial	-	-	-	-	-
`ZLM.type==tFA`	-	✓	-	-	-

Die Übergänge von C10 nach C9 und von C9 nach C39 brauchen nicht betrachtet zu werden, da der Model-Checker-Lauf gezeigt hat, dass diese nie erreicht werden. Dies sind die Fälle, in denen eine andere Nachricht als die erwartete vom Zugleiter empfangen wird, das hier entwickelte Protokoll vermeidet diesen Fehler.

C39 → C38

```
atomic {
  Check_KH(train, Current[train], Next[train]);
```

Statement	ZV	R:FA	S:FE	S:AFE	R:AM
Initial	-	✓	-	-	-
`Check_KH`	-	✓	-	-	-

Dieser Übergang ruft die inline-Funktion auf, die alle anderen Züge auf die Bedingungen für „Kein Hindernis“ überprüft, s. Meaning Postulate (11.10).

C38 → C54

```
ZV[train].previous[Current[train]].next[Next[train]] =
  KH[train].previous[Current[train]].next[Next[train]];
```

Statement	ZV	R:FA	S:FE	S:AFE	R:AM
Initial	-	✓	-	-	-
`ZV = KH`	==KH	✓	-	-	-

C54 → C44

```
:: (ZV[train].previous[Current[train]].next[Next[train]]) ->
```

Statement	ZV	R:FA	S:FE	S:AFE	R:AM
Initial	-	✓	-	-	-
`ZV == true`	✓	✓	-	-	-

C54 → C51

```
:: else ->
```

Statement	ZV	R:FA	S:FE	S:AFE	R:AM
Initial	-	✓	-	-	-
`ZV == false`	-	✓	-	-	-

C44 → C45

```
d_step {
  ZLM.type = tFE;
  ZLM.train = train;
  ZLM.destination = Next[train];
}
```

Statement	ZV	R:FA	S:FE	S:AFE	R:AM
Initial	✓	✓	-	-	-
FE-Nachricht erstellen	✓	✓	-	-	-

C45 → C58

```
c2t[train]!ZLM;
```

Statement	ZV	R:FA	S:FE	S:AFE	R:AM
Initial	✓	✓	-	-	-
`c2t!ZLM`	✓	✓	✓	-	-

C58 → C69

```
atomic {
  previous = Current[train];
  next = Next[train];
}
```

Statement	ZV	R:FA	S:FE	S:AFE	R:AM
Initial	✓	✓	✓	-	-
Zuweisung Hilfsvariablen	✓	✓	✓	-	-

C69 → C67

```
t2c[train]?ZLM;
```

Statement	ZV	R:FA	S:FE	S:AFE	R:AM
Initial	✓	✓	✓	-	-
`t2c?ZLM`	✓	✓	✓	-	-

C67 → C109

```
:: ( ZLM.type == tAM ) ->
```

Statement	ZV	R:FA	S:FE	S:AFE	R:AM
Initial	✓	✓	✓	-	-
`ZLM.type==tAM`	✓	✓	✓	-	✓

C51 → C52

```
d_step {
  ZLM.type = tAFE;
  ZLM.train = train;
  ZLM.destination = Next[train];
}
```

Statement	ZV	R:FA	S:FE	S:AFE	R:AM
Initial	-	✓	-	-	-
Erzeugen AFE-Nachricht	-	✓	-	-	-

C52 → C97

```
c2t[train]!ZLM;
```

Statement	ZV	R:FA	S:FE	S:AFE	R:AM
Initial	-	✓	-	-	-
`c2t!ZLM`	-	✓	-	✓	-

C97 → C96

```
Check_KH(train, Current[train], Next[train]);
```

Statement	ZV	R:FA	S:FE	S:AFE	R:AM
Initial	-	✓	-	✓	-
`Check_KH`	-	✓	-	✓	-

C96 → C107

```
ZV[train].previous[Current[train]].next[Next[train]] =
  KH[train].previous[Current[train]].next[Next[train]]
```

Statement	ZV	R:FA	S:FE	S:AFE	R:AM
Initial	-	✓	-	✓	-
`ZV = KH`	`== KH`	✓	-	✓	-

C107 → C102

```
:: ( ZV[train].previous[Current[train]].next[Next[train]] ) ->
```

Statement	ZV	R:FA	S:FE	S:AFE	R:AM
Initial	-	✓	-	✓	-
`ZV = true`	✓	✓	-	✓	-

C107 → C97

```
:: else ->
```

Statement	ZV	R:FA	S:FE	S:AFE	R:AM
Initial	-	✓	-	✓	-
`ZV = false`	-	✓	-	✓	-

C102 → C103

```
d_step {
  ZLM.type = tFE;
  ZLM.train = train;
  ZLM.destination = Next[train];
};
```

Statement	ZV	R:FA	S:FE	S:AFE	R:AM
Initial	✓	✓	-	✓	-
FE-Nachricht vorbereiten	✓	✓	-	✓	-

C103 → C58

```
c2t[train]!ZLM;
```

Statement	ZV	R:FA	S:FE	S:AFE	R:AM
Initial	✓	✓	-	✓	-
`c2t!ZLM`	✓	✓	✓	-	-

11.11.3 MFG und PROMELA-Maschine für Zugführer

Durch Anwendung der Meaning Postulates kann direkt gezeigt werden, dass jeweils ein Zustandsübergang oder eine Reihe Übergänge im SPIN-Modell einem Zustandsübergang im Zugführerautomaten des MFG entspricht.

Übergang MFG/Zugführer	Übergänge PROMELA/Zugführer
$s_0 \rightarrow s_1$	D38 → D7 → D6 → D8
$s_1 \rightarrow s_6$	D8 → D13 → D16
$s_1 \rightarrow s_x$	D8 → D13
$s_x \rightarrow s_1$	D13 → D8
$s_6 \rightarrow s_8$	D16 → D19 → D22
$s_8 \rightarrow s_9$	D22 → D28
$s_9 \rightarrow s_{11}$	D28 → D27 → D36

Beim Übergang s_9 nach s_{10} im MFG gibt es keine Entsprechung im PROMELA-Modell, weil es sich hier um einen unerwünschten undefinierten Zustand handelt, der im Betrieb, wie er hier entwickelt wurde, nicht vorkommt.

11.11.4 MFG und PROMELA-Maschine für Zugleiter

Auch hier kann durch Anwendung der Meaning Postulates direkt gezeigt werden, dass jeweils ein Zustandsübergang oder eine Reihe Übergänge im SPIN-Modell einem Zustandsübergang im Zugführerautomaten des MFG entspricht.

Übergang MFG/Zugleiter	Übergänge PROMELA/Zugleiter
$s_0 \rightarrow s_2$	C5 $\rightarrow$ C10 $\rightarrow$ C39
$s_2 \rightarrow s_5$	C39 $\rightarrow$ C38 $\rightarrow$ C54 $\rightarrow$ C51 $\rightarrow$ C52 $\rightarrow$ C97
$s_2 \rightarrow s_4$	C39 $\rightarrow$ C38 $\rightarrow$ C54 $\rightarrow$ C44 $\rightarrow$ C45 $\rightarrow$ C58
$s_5 \rightarrow s_4$	C97 $\rightarrow$ C96 $\rightarrow$ C107 $\rightarrow$ C102 $\rightarrow$ C103 $\rightarrow$ C58
$s_4 \rightarrow s_12$	C58 $\rightarrow$ C69 $\rightarrow$ C67 $\rightarrow$ C109

11.12 Ende der Zugfahrten

Bisher wurde gezeigt, dass ein Abschnitt einer Zugfahrt von einer Zuglaufmeldestelle zur nächsten im PROMELA-Modell einen entsprechenden Fahrtabschnitt im MFG implementiert, entweder indem einer Transition im Zustandsautomaten des MFG direkt eine Transition im PROMELA-Prozess entspricht, oder durch eine Anzahl von „Stotterschritten“ [Lam83].

Um zu zeigen, dass alle Züge am Ende an ihrem Bestimmungsort ankommen, muss noch gezeigt werden, dass der akzeptierende Endzustand der Prozessinstanzen im Verifier genau dann erreicht wird, wenn der Zug an seiner Zielzuglaufmeldestelle angekommen ist.

11.12.1 Zugführer

Die Übergangssequenz, die in den Endzustand des Zugführer-Automaten D0 führt, ist D36 $\rightarrow$ D30 $\rightarrow$ D41 $\rightarrow$ D0. Die Variable `destination` enthält innerhalb des Zugführer-Prozesses den Wert des Zielbahnhofs dieses Zuges, `Current[`n`]` enthält den Wert der aktuellen Station.

```
if
:: Current[train] == destination ->
```

Der ganze „Zweig“, der ab Zustand D36 zum akzeptierten Endzustand D0 führt, wird genau dann durchlaufen, wenn die Bedingung oben wahr ist. Damit ist sichergestellt, dass beim Erreichen der Zielstation der Prozess

beendet wird. Die Verifikation stellt sicher, dass der Prozess an keiner anderen Stelle beendet wird.

11.12.2 Zugleiter

Die Übergangssequenz, die in den Endzustand des Zugleiter-Automaten DC führt, ist C110 $\rightarrow$ C113 $\rightarrow$ C0. Die Variable `destination` enthält innerhalb des Zugführer-Prozesses den Wert des Zielbahnhofs dieses Zuges, `Current[`n`]` enthält den Wert der aktuellen Station.

```
:: Current[train] == destination ->
  break;
```

Der ganze „Zweig“, der ab Zustand C110 zum akzeptierten Endzustand C0 führt, wird genau dann durchlaufen, wenn die Bedingung oben wahr ist. Damit ist sichergestellt, dass beim Erreichen der Zielstation der Prozess beendet wird. Die Verifikation stellt sicher, dass der Prozess an keiner anderen Stelle beendet wird.

11.13 Quellcode

Der vollständige Quellcode des PROMELA-Modells findet sich im Anhang.

`Operation_Level3.prom` ist der Quellcode für das Kreisstreckenmodell, das aufgrund der Zustandsexplosion keine sinnvolle Überprüfung des Protokolls zuließ.

`Four-Train-Situations.prom` ist der Source-Code für die Lineare Strecke mit 5 Stationen und 4 Zügen. Jede der 10 möglichen Ausgangssituationen wird in Zeile 129 eingefügt.

Die Situations-Files, die im Anhang jeweils unter Titeln wie `Output-E.txt` wiedergegeben sind, entsprechen den in Tabelle 11.2 gelisteten Situationen.

Dabei werden anhand der Rahmenbedingungen, dass in einer Station nur dann zwei Züge sein können, wenn diese in unterschiedlichen Richtungen starten. Wenn in einer Station nur ein Zug ist, dann startet dieser nichtdeterministisch in einer der möglichen Richtungen. Der Verifier des Model-Checkers garantiert, dass alle möglichen Kombinationen getestet werden, und im Falle eines Durchlaufs ohne *invalid end states* ist gewährleistet, dass

alle Züge alle Stationen anfahren und jeweils am Ende der Strecke ankommen.

11.14 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 11.6 zeigt, wie sich der Message-Flow-Graph jeweils zum PROMELA-Modell und zum SPARK-Source-Code verhält.

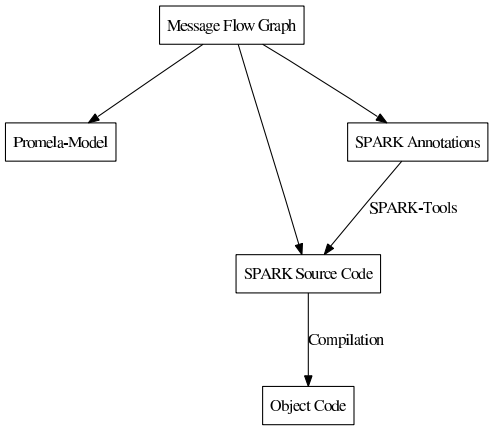


Abbildung 11.6: Beziehungen zwischen Message-Flow-Graph, SPARK Code und Promela-Modell

11.14.1 MFG — SPARK Source Code

In Kapitel 10 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.

11.14.2 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.

11.14.3 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.

11.14.4 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. [Bar03] gibt eine detaillierte Beschreibung der Benutzung und Fähigkeiten der SPARK-Tools.

11.14.5 SPARK-Quellcode — Object Code

Der verifizierte 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.

11.15 Bewertung

Der Model-Checker SPIN hat sich nach anfänglichen Schwierigkeiten als geeignet herausgestellt, Liveness-Properties des hier entwickelten Protokolls nachzuweisen. Die Probleme, die zunächst auftraten, rühren zum Teil daher, dass der Autor vor dieser Arbeit noch keine Erfahrung mit dem praktischen Einsatz eines Model-Checkers hatte.

Andererseits benutzt SPIN zwar eine Reihe fortgeschrittener Techniken zur Suchraumeinschränkung, ohne die Gültigkeit des Ergebnisses zu verändern, dennoch ergibt sich für viele Fälle, so zum Beispiel die zuerst vorgeschlagene Kreisstrecke eine solche Zustandsexplosion, dass die Verifikation nicht praktisch durchführbar gewesen wäre. Erst eine manuelle weitere deutliche Beschränkung des Zustandsraumes durch die Reduktion auf zwei Handvoll Ausgangssituationen erlaubte eine sinnvolle Durchführung der Überprüfung des gesamten Zustandsraumes für diese Situationen.

Die Möglichkeit, alle diese Ausgangssituationen durch nichtdeterministische Konstrukte direkt in ein einzelnes Source-File einzubinden, hätte eine weitere Erhöhung des Speicherbedarfs bedeutet. Diesem hätte man zwar durch eine stärkere Kompression durch Graph-Encoding begegnen können, aber hätte dadurch weiter an Geschwindigkeit eingebüßt, so dass die Gesamtlaufzeit sich weiter deutlich erhöht hätte. Insgesamt stellt die hier verwendete Methode für diesen Fall daher einen guten Kompromiss aus manuellem Aufwand, Nutzen und Computerressourcen-Nutzung dar.

Kapitel 12

Fazit

12.1 Durchgehende logische Nachvollziehbarkeit

Es hat sich gezeigt, dass die Ontological Hazard Analysis in der Lage ist, für ein echtes Bahnbetriebsverfahren, eine nahtlose Nachvollziehbarkeit (*audit trail*) von abstrakten Spezifikationen bis zum Quellcode — abgesehen von möglichen Compilerproblemen auch bis zum Objektcode — zu bieten, und dabei gleichzeitig logische, relative Vollständigkeit der Sicherheitsanforderungen (*Safety Requirements*) zu garantieren.

Mir ist kein anderes Verfahren bekannt, dass dieses tatsächlich leistet; andere Verfahren konzentrieren sich in der Regel auf einzelne Aspekte des Software-Entwicklungsprozesses, können aber keinen ununterbrochenen logisch rigorosen Nachweis erbringen, dass die abstrakten Sicherheitsanforderungen der anfänglich einfachen abstrakten Systembeschreibung vom Quellcode erfüllt werden.

Die Ontological Hazard Analysis hat in diesem Beispiel sehr gut funktioniert, da sich auch ohne die Komplexitätsexplosion, die durch Anwendung von HAZOP eingetreten wäre, alle Verfeinerungen unmittelbar durch das Formal Refinement beweisen ließen. Auf diese Weise gelang eine Konkretisierung der Abstraktionen der vorangegangenen Ebenen bis zur Implementation in SPARK und in einem PROMELA-Modell für SPIN.

Es ist nicht garantiert, dass dies in allen Fällen so gut funktioniert. Stuphorn hat in [Stu05] eine Verfeinerung mittels HAZOP verwendet, und diese hat eine in den ersten Verfeinerungsschritten zu einer erheblichen Zu-

nahme an Objekten und Relationen geführt, doch zeigte sich auch hier, dass bereits nach wenigen Schritten eine Konvergenz zu erkennen war in Richtung einer vollständigen Beschreibbarkeit des System und aller Funktionsabweichungen (*Deviations*) mit der verfügbaren Sprache.

12.2 Implementation

Die einfach durchzuführende Implementation durch einen erfahrenen SPARK-Entwickler zeigt, dass sich das System ohne große Hindernisse in der Praxis implementieren ließe. Dadurch, dass gezeigt werden konnte, dass sowohl der SPARK-Code als auch die PROMELA-Prozesstypen die kommunizierenden Zustandsautomaten des Message-Flow-Graphen implementieren, sowie dass der SPARK-Code die Pre- und Postconditions erfüllt, weiterhin die Safety-Axiome der ersten Abstraktionsebene in enger Kooperation mit Domain-Experten als vollständig und richtig erkannt wurden, und außerdem das Model-Checking gezeigt hat, dass ein verklemmungsfreier Betrieb möglich ist, kann mit hohem Gewissheit davon ausgegangen werden, dass das Protokoll auch in der Praxis einsetzbar ist.

12.3 Praxisanwendung

VDV-Schrift 752[VDV04] schreibt vor, dass unmodifizierter Zugleitbetrieb nicht mehr als Betriebsverfahren für neu zu eröffnende Strecken verwendet werden darf. Für Strecken mit geringem Verkehrsaufkommen, ist Zugleitbetrieb zulässig, wenn für eine technische Sicherung gesorgt wird. Die Art dieser Sicherung ist nicht vorgeschrieben. Ein Protokoll wie das hier entwickelte, bei dem die Kommunikation als sichere Datenübertragung zwischen Computern implementiert wird, in Zusammenhang mit einer technischen Wegfahrsperre im Triebfahrzeug, die ein Fahren des Zuges ohne Fahrerlaubnis verhindert, könnte als technische Sicherung im Sinne dieser Schrift ausreichend sein. Die lückenlose Verfeinerungskette bis zur Implementation garantiert dabei, dass die Funktionsweise des Algorithmus auf dem verteilten Computernetz (Anlagen bei Zugleiter, und auf jedem Triebfahrzeug) fehlerfrei funktioniert: sicheren und verklemmungsfreien Betrieb gewährleistet.

Ein solches System kann kostengünstig sein, da keine Modifikationen der Gleisanlage erforderlich sind, beispielsweise keine aufwendigen Achsenzähler oder Balisen mit dafür notwendiger Infrastruktur.

12.4 Tool Support

Die Ontological Hazard Analysis ist noch eine neue Methode, und diese Arbeit stellt bisher die einzige Anwendung dar, in der sie bis hin zur Source- und Object-Code-Ebene vollständig durchgeführt wurde, und damit erstmals die Machbarkeit dieses Verfahrens unter Beweis gestellt wurde.

Weil quasi alles neu erfunden werden musste, gibt es noch keine Software, die diese Prozesse unterstützt. Hilfreich könnte sich vor allem Software erweisen, die die Buchhaltungsaufgaben übernimmt, das heißt die Ontologie mit allen Objekten und Relationen verwaltet, und auch Buch führt über die Bedeutungspostulate, Sicherheits- (Safety und Security) Axiome und Verfeinerungsbeweise.

Zur Erfassung, Strukturierung und Formalisierung der Anforderungen und Spezifikationen ist ein Parsing- Datenbanksystem vorstellbar, dass natürlichsprachlich formulierte Pflichten- und Lastenheften segmentiert, und damit eine Erfassung und Überprüfung sowie eine Formalisierung dieser Anforderungen erleichtert. Das Datenbanksystem kann auch beim Vergleich mit früher entwickelten Systemen helfen, durch Vergleich mit damaligen Anforderungen, Vollständigkeit zu erzielen.

12.5 Ausblick

Das vorgestellte Verfahren der Ontological Hazard Analysis ist nicht auf den Einsatz im Bahnbereich beschränkt; neben den weiteren durchgeführten Anwendungsbeispielen [SSL09] ist allgemein eine Anwendung in der Requirements-Analyse und Implementation von sicherheitskritischen Anwendungen gut vorstellbar. Neben weiteren Bahnbetriebsprotokollen und Stellwerken sind die Luftfahrt- und Automobilindustrie sowie deren Zulieferer für die Entwicklung zuverlässiger und Sicherer E/E/PE-Systeme zunehmend auf effiziente, dabei aber rigorose formale

Entwicklungsprozesse angewiesen.

Anforderungen und Spezifikationen sind traditionell einer der Schwachpunkte bei der Entwicklung von embedded Software, insbesondere für sicherheits- oder missionskritische Systeme. Eine rigorose Methode wie die hier vorgestellte Ontological Hazard Analysis zusammen mit einem Correct-By-Construction-Entwicklungsmodell wie SPARK kann bei vielen dieser Probleme effektiv bei der Lösung unterstützen.

Literaturverzeichnis

[Bar03] John Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison Wesley, 2003.

[BSD09] NetBSD web site. Online: `http://www.netbsd.org/`, March 2009.

[Car52] Rudolph Carnap. Meaning postulates. *Philosophical Studies*, 3:65–73, 1952.

[Dij75] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975. Online: `http://www.cs.utexas.edu/users/EWD/ewd04xx/EWD472.PDF`.

[DJ95] Gregory Duval and Jacques Julliand. Modeling and verification of the RUBIS μ-kernel with SPIN. In *Proceedings of the First SPIN Workshop*, 1995. Online: `http://spinroot.com/spin/Workshops/ws95/duval.pdf`.

[FVN] Verband Deutscher Verkehrsunternehmen. *Fahrdienstvorschrift für Nichtbundeseigene Eisenbahnen (FV-NE)*. Ausgabe 1984, Fassung 2004.

[GLL+00] S. Gnesi, G. Lenzini, D. Latella, C. Abbaneo, A. Amendola, and P. Marmo. An automatic SPIN validation of a safety critical railway control system. In *Proceedings of the 2000 International Conference on Dependable Systems and Networks*, pages 119–224, 2000. Online: `http://fmt.isti.cnr.it/WEBPAPER/ftcs00.ps`.

[Gre96] J. C. Gregoire. State space compression in spin with GETSs. In *Proceedings of the Second Spin Workshop*, New Brunswick, New Jersey, August 1996. Rutgers University, American Mathematical Society.

[HAZ01] International Electrotechnical Comission. *International Standard 61882, Hazard and Operability studies, Application Guide*, first edition, May 2001.

[Hoa78] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, 1978. Online: `http://www.cs.virginia.edu/crab/hoare1978csp.pdf`.

[Hol99] Gerard J. Holzmann. The engineering of a model checker: the Gnu i-protocol case study revisited. Technical report, Bell Laboratories, Lucent Technologies, 1999. Online: `http://spinroot.com/spin/Doc/spin99.pdf`.

[Hol03] Gerard J. Holzmann. *The SPIN Model Checker*. Addison Wesley, 2003.

[HP95] Gerard J. Holzmann and Doron Peled. An improvement in formal verification. In *Proceedings of the 7th IFIP WG6.1 International Conference on Formal Description Techniques VII*, pages 197–211, London, UK, UK, 1995. Chapman & Hall, Ltd.

[HS00] Gerard J. Holzmann and Margaret H. Smith. Automating software feature verification. *Bell Labs Technical Journal*, Special Issue on Software Complexity, April–June 2000. Online: `http://spinroot.com/gerard/pdf/bltj2000.pdf`.

[Lad95] Peter Ladkin. Analysis of a technical description of the Airbus A320 braking system. *High Integrity Systems*, 1(4):331–349, 1995.

[Lad05] Peter B. Ladkin. Ontological analysis. *SafetySystems*, 14(3), May 2005. Online: `http://www.rvs.uni-bielefeld.de/` → `Publications` → `Publications of Record`.

[Lam83] Leslie Lamport. What good is temporal logic? *Information Processing*, pages 657–668, 1983.

[Lam94] Leslie Lamport. How to write a long formula. Technical Report 119, DEC Systems Research Center, 1994. Online: `http://research.microsoft.com/en-us/um/people/lamport/pubs/lamport-howtowrite.pdf`.

[Lam95] Leslie Lamport. TLA in pictures. *IEEE Transactions on Software Engineering*, SE-21:768–775, September 1995. Online: `http://research.microsoft.com/en-us/um/people/lamport/pubs/lamport-pictures.pdf`.

[LL95] Peter B. Ladkin and Stefan Leue. Interpreting message flow graphs. *Formal Aspects of Computing*, 7(5):473–509, 1995.

[LO82] Leslie Lamport and Susan Owicki. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems*, 4(3):455–495, July 1982.

[PMRL04] Shivendra S. Panwar, Shiwen Mao, Jeong-dong Ryoo, and Yihan Li. *TCP/IP Essentials: A Lab-Based Approach*. Cambridge University Press, New York, NY, USA, 2004.

[SS83] D. Skeen and M. Stonebraker. A formal model of crash recovery in a distributed system. *IEEE Transactions on Software Engineering*, 9(3):219–228, 1983.

[SSL09] J. Stuphorn, B. Sieker, and P.B. Ladkin. Dependable risk analysis for systems with E/E/PE components: Two case studies. *Proceedings of the Safety-critical Systems Symposium 2009*, 2009.

[Stu05] Jörn Stuphorn. Iterative decomposition of a communication-bus system using ontological analysis. Diplomarbeit, Universität Bielefeld, July 2005. RVS-Dip-05-02 Online: `http://www.rvs.uni-bielefeld.de/` → `Publications` → `Theses`.

[VDV04] Verband Deutscher Verkehrsunternehmen. *Schriften 752: Empfehlungen zur Auswahl geeigneter Betriebsverfahren fr eingleisige Eisenbahnstrecken*, 3 2004.

Index

Anhang A.1: SPARK-Sourcecode von Phil Thornley

```
    1: ------------------------------------------------------------------------------
    2: -- Control package specification
    3: -- Version 1.0, April 2006
    4: --
    5: --  Copyright (C) 2006 Phil Thornley - phil@sparksure.com
    6: --
    7: --  Produced in cooperation with Bernd Sieker and Peter Ladkin,
    8: --                               University of Bielefeld
    9: --
   10: --  This program is free software; you can redistribute it and/or modify
   11: --  it under the terms of the GNU General Public License as published by
   12: --  the Free Software Foundation; either version 2 of the License, or
   13: --  (at your option) any later version.
   14: ------------------------------------------------------------------------------
   15:
   16: with Trains;
   17: with Track;
   18: with Messages.FA_M;
   19: --# inherit Trains,
   20: --#         Track,
   21: --#         Messages.FA_M,
   22: --#         Messages.FE_M,
   23: --#         Messages.AFE_M,
   24: --#         Messages.AM_M,
   25: --#         Configuration;
   26: package Control
   27: --# own C_State : C_State_T;
   28: is
   29:
   30:    --# type C_State_T is abstract;
   31:
   32:    type C_States is (C_S0, C_S2, C_S4, C_S5, C_S12);
   33:
   34:    function State_Of(T : Trains.Train_ID) return C_States;
   35:    --# global C_State;
   36:
   37:    function Position(T : Trains.Train_ID) return Track.Station_ID;
   38:    --# global C_State;
   39:
   40:    function Next(T : Trains.Train_ID) return Track.Station_T;
   41:    --# global C_State;
   42:
   43:    procedure Initialize;
   44:    --# global    out C_State;
   45:    --# derives C_State from ;
   46:
   47:
   48:    --# function To_S2 (T            : Trains.Train_ID;
   49:    --#                 S_Imp, S_Exp : C_State_T      ) return Boolean;
   50:    --#
   51:    --# function To_S5 (T            : Trains.Train_ID;
   52:    --#                 S_Imp, S_Exp : C_State_T      ) return Boolean;
   53:    --#
   54:    --# function To_S4 (T            : Trains.Train_ID;
   55:    --#                 S_Imp, S_Exp : C_State_T      ) return Boolean;
   56:    --#
   57:    --# function To_S12 (T            : Trains.Train_ID;
   58:    --#                  S_Imp, S_Exp : C_State_T      ) return Boolean;
   59:    --#
   60:    --# function No_Change (S_Imp, S_Exp : C_State_T) return Boolean;
   61:
   62:    --# function Train_Is_Waiting (T : Trains.Train_ID; S : C_State_T) return Boolean;
   63:    --#
   64:    --# function Next_Is_Set      (T : Trains.Train_ID; S : C_State_T) return Boolean;
   65:    --#
   66:    --# function No_Train_At         (St : Track.Station_ID; S : C_State_T) return Boolean;
   67:    --#
   68:    --# function No_Train_Cleared_To (St : Track.Station_ID; S : C_State_T) return Boolean;
   69:
   70:
   71:    procedure Process_FA (Message : in     Messages.FA_M.FA_Message_T;
   72:                          Train   : in     Trains.Train_ID);
   73:    --# global in out C_State;
   74:    --# derives C_State from
   75:    --#            *,
   76:    --#            Message,
   77:    --#            Train;
   78:    --# post (State_Of(Train, C_State~)  = C_S0 -> To_S2(Train, C_State~, C_State))
   79:    --#      and
   80:    --#      (State_Of(Train, C_State~) /= C_S0 -> No_Change(C_State~, C_State));
   81:
   82:    procedure Process_AM (Train   : in     Trains.Train_ID);
   83:    --# global in out C_State;
   84:    --# derives C_State from
   85:    --#            *,
   86:    --#            Train;
```

```
   87:    --# pre Next_Is_Set (Train, C_State);
   88:    --# post (State_Of(Train, C_State~)  = C_S4 -> To_S12(Train, C_State~, C_State))
   89:    --#      and
   90:    --#      (State_Of(Train, C_State~) /= C_S4 -> No_Change(C_State~, C_State));
   91:
   92:    function KH (Train : Trains.Train_ID) return Boolean;
   93:    --# global C_State;
   94:    --# pre Next_Is_Set (Train, C_State);
   95:    --# return (No_Train_At         (Next(Train, C_State), C_State) and
   96:    --#         No_Train_Cleared_To (Next(Train, C_State), C_State));
   97:
   98:
   99:    procedure Check_Waiting_Train (Train : in     Trains.Train_ID);
  100:    --# global in out C_State;
  101:    --#           out Messages.Out_Queues;
  102:    --# derives C_State from
  103:    --#            *,
  104:    --#            Train
  105:    --# &       Messages.Out_Queues from
  106:    --#            C_State,
  107:    --#            Train;
  108:    --# pre Train_Is_Waiting (Train, C_State) -> Next_Is_Set (Train, C_State);
  109:    --# post ((State_Of(Train, C_State~) = C_S2 and KH(Train, C_State~))
  110:    --#                                       -> To_S4(Train, C_State~, C_State))
  111:    --#      and
  112:    --#      ((State_Of(Train, C_State~) = C_S2 and not KH(Train,C_State~))
  113:    --#                                       -> To_S5(Train, C_State~, C_State))
  114:    --#      and
  115:    --#      ((State_Of(Train, C_State~) = C_S5 and KH(Train, C_State~))
  116:    --#                                       -> To_S4(Train, C_State~, C_State))
  117:    --#      and
  118:    --#      ((State_Of(Train, C_State~) = C_S5 and not KH(Train, C_State~))
  119:    --#                                       -> No_Change(C_State~, C_State))
  120:    --#      and
  121:    --#      ((State_Of(Train, C_State~) /= C_S2 and
  122:    --#        State_Of(Train, C_State~) /= C_S5) -> No_Change(C_State~, C_State));
  123:
  124:    procedure Prepare_Next_Move (Train : in     Trains.Train_ID);
  125:    --# global in out C_State;
  126:    --# derives C_State from
  127:    --#            *,
  128:    --#            Train;
  129:
  130:    procedure Pass_Time;
  131:    --# derives ;
  132:
  133: end Control;
```

```
    1: ------------------------------------------------------------------------------
    2: -- Control package body
    3: -- Version 1.0, April 2006
    4: --
    5: --  Copyright (C) 2006 Phil Thornley - phil@sparksure.com
    6: --
    7: --  Produced in cooperation with Bernd Sieker and Peter Ladkin,
    8: --                               University of Bielefeld
    9: --
   10: --  This program is free software; you can redistribute it and/or modify
   11: --  it under the terms of the GNU General Public License as published by
   12: --  the Free Software Foundation; either version 2 of the License, or
   13: --  (at your option) any later version.
   14: ------------------------------------------------------------------------------
   15:
   16: with Messages.FE_M;
   17: with Messages.AFE_M;
   18: with Messages.AM_M;
   19: with Track;        use type Track.Station_ID;
   20: with Configuration;
   21: package body Control
   22: --# own C_State is
   23: --#        Current_State;
   24: is
   25:
   26:    No_Station : constant := Track.Station_T'First;
   27:
   28:    type Train_Rec is
   29:       record
   30:          C_State : C_States;
   31:          At_St   : Track.Station_ID;
   32:          Next_St : Track.Station_T;
   33:       end record;
   34:
   35:    type States_T is array(Trains.Train_ID) of Train_Rec;
   36:    Current_State : States_T;
   37:
   38:    procedure Initialize
   39:    --# global    out Current_State;
   40:    --# derives Current_State from ;
   41:    is
   42:    begin
   43:       for I in Trains.Train_ID loop
   44:          Current_State(I) := Train_Rec'(C_State => C_S0, -- Expected Flow error 23 here.
   45:                                         At_St   => Configuration.Starting_Stations(I),
   46:                                         Next_St => No_Station);
   47:       end loop;
   48:    end Initialize;  -- Expected Warning 602 here.
   49:
   50:    procedure Process_FA (Message : in     Messages.FA_M.FA_Message_T;
   51:                          Train   : in     Trains.Train_ID)
   52:    --# global in out Current_State;
   53:    --# derives Current_State from
   54:    --#            *,
   55:    --#            Train,
   56:    --#            Message;
   57:    --# post (Current_State~(Train).C_State = C_S0 ->
   58:    --#          (Current_State(Train).C_State = C_S2       and
   59:    --#           Current_State(Train).Next_St > No_Station and
   60:    --#          (for all T in Trains.Train_ID =>
   61:    --#            (T /= Train -> Current_State~(T) = Current_State(T)))))
   62:    --#      and
   63:    --#      (Current_State~(Train).C_State /= C_S0 -> Current_State~ = Current_State);
   64:    is
   65:    begin
   66:       if Current_State(Train).C_State = C_S0 then
   67:          Current_State(Train).C_State := C_S2;
   68:          Current_State(Train).Next_St := Messages.FA_M.Station_Of(Message);
   69:       end if;
   70:    end Process_FA;
   71:
   72:    procedure Process_AM (Train   : in     Trains.Train_ID)
   73:    --# global in out Current_State;
   74:    --# derives Current_State from
   75:    --#            *,
   76:    --#            Train;
   77:    --# pre Current_State(Train).Next_St > No_Station;
   78:    --# post (Current_State~(Train).C_State = C_S4 ->
   79:    --#          (Current_State(Train) = Train_Rec'(C_S12,
   80:    --#                                             Current_State~(Train).Next_St,
   81:    --#                                             No_Station)
   82:    --#          and
   83:    --#          (for all T in Trains.Train_ID =>
   84:    --#            (T /= Train -> Current_State~(T) = Current_State(T)))))
   85:    --#      and
   86:    --#      (Current_State~(Train).C_State /= C_S4 -> Current_State~ = Current_State);
```

```
   is
   begin
      if Current_State(Train).C_State = C_S4 then
         Current_State(Train) := Train_Rec'(C_S12,
                                            Current_State(Train).Next_St,
                                            No_Station);
      end if;
   end Process_AM;

   function KH (Train : Trains.Train_ID) return Boolean
   --# global Current_State;
   --# pre Current_State(Train).Next_St > No_Station;
   --# return ((for all T in Trains.Train_ID =>
   --#          (Current_State(T).At_St /= Current_State(Train).Next_St)) and
   --#         (for all T in Trains.Train_ID =>
   --#           (Current_State(T).C_State = C_S4 ->
   --#             Current_State(T).Next_St /= Current_State(Train).Next_St)));
   is
      Nxt    : Track.Station_ID;
      Result : Boolean := True;
   begin
      Nxt := Current_State(Train).Next_St;
      for I in Trains.Train_ID loop
         if Current_State(I).At_St = Nxt then
            Result := False;
            exit;
         end if;
         if Current_State(I).C_State = C_S4 and
            Current_State(I).Next_St = Nxt
         then
            Result := False;
            exit;
         end if;
      --# assert Result and
      --#        Nxt = Current_State(Train).Next_St and
      --#        (for all T in Trains.Train_ID range 1 .. I =>
      --#          (Current_State(T).At_St /= Current_State(Train).Next_St)) and
      --#        (for all T in Trains.Train_ID range 1 .. I =>
      --#          (Current_State(T).C_State = C_S4 ->
      --#            Current_State(T).Next_St /= Current_State(Train).Next_St));
      end loop;
      return Result;
   end KH;

   procedure Check_Waiting_Train (Train : in     Trains.Train_ID)
   --# global in out Current_State;
   --#           out Messages.Out_Queues;
   --# derives Current_State from
   --#            *,
   --#            Train
   --# &       Messages.Out_Queues from
   --#            Current_State,
   --#            Train;
   --# pre (Current_State(Train).C_State = C_S2 or
   --#         Current_State(Train).C_State = C_S5) ->
   --#                Current_State(Train).Next_St > No_Station;
   --# post ((Current_State~(Train).C_State = C_S2 and
   --#        KH(Train, Current_State~)) ->
   --#          (Current_State(Train) = Train_Rec'(C_S4,
   --#                                             Current_State~(Train).At_St,
   --#                                             Current_State~(Train).Next_St) and
   --#          (for all T in Trains.Train_ID =>
   --#            (T /= Train -> Current_State~(T) = Current_State(T)))))
   --#      and
   --#      ((Current_State~(Train).C_State = C_S2 and
   --#        not KH(Train, Current_State~)) ->
   --#          (Current_State(Train) = Train_Rec'(C_S5,
   --#                                             Current_State~(Train).At_St,
   --#                                             Current_State~(Train).Next_St) and
   --#          (for all T in Trains.Train_ID =>
   --#            (T /= Train -> Current_State~(T) = Current_State(T)))))
   --#      and
   --#      ((Current_State~(Train).C_State = C_S5 and
   --#        KH(Train, Current_State~)) ->
   --#          (Current_State(Train) = Train_Rec'(C_S4,
   --#                                             Current_State~(Train).At_St,
   --#                                             Current_State~(Train).Next_St) and
   --#          (for all T in Trains.Train_ID =>
   --#            (T /= Train -> Current_State~(T) = Current_State(T)))))
   --#      and
   --#      ((Current_State~(Train).C_State = C_S5 and
   --#        not KH(Train, Current_State~)) -> Current_State~ = Current_State)
   --#      and
   --#      ((Current_State~(Train).C_State /= C_S2 and
   --#        Current_State~(Train).C_State /= C_S5) -> Current_State~ = Current_State);

```

```
   is

      procedure Send_FE (T : in     Trains.Train_ID;
                         S : in     Track.Station_ID)
      --# global Messages.Out_Queues;
      --# derives Messages.Out_Queues from
      --#            T,
      --#            S;
      is
         FE_Message : Messages.FE_M.FE_Message_T;
      begin
         Messages.FE_M.Initialize (FE_Message,
                                   T,
                                   S);
         Messages.FE_M.Send (FE_Message);
      end Send_FE;

      procedure Send_AFE
      --# global Messages.Out_Queues;
      --# derives Messages.Out_Queues from ;
      is
         AFE_Message : Messages.AFE_M.AFE_Message_T;
      begin
         Messages.AFE_M.Initialize (AFE_Message);
         Messages.AFE_M.Send (AFE_Message);
      end Send_AFE;

   begin
      if Current_State(Train).C_State = C_S2 then
         if KH(Train) then
            Send_FE (Train,
                     Current_State(Train).Next_St);
            Current_State(Train).C_State := C_S4;
         else
            Send_AFE;
            Current_State(Train).C_State := C_S5;
         end if;
      elsif Current_State(Train).C_State = C_S5 then
         if KH(Train) then
            Send_FE (Train,
                     Current_State(Train).Next_St);
            Current_State(Train).C_State := C_S4;
         end if;
      end if;
   end Check_Waiting_Train;

   procedure Prepare_Next_Move (Train : in     Trains.Train_ID)
   --# global in out Current_State;
   --# derives Current_State from
   --#            *,
   --#            Train;
   is
   begin
      if Current_State(Train).C_State = C_S12 then
         Current_State(Train).C_State := C_S0;
      end if;
   end Prepare_Next_Move;

   procedure Pass_Time
   is
   --# hide Pass_Time;
   begin
      delay 0.0;
   end Pass_Time;

   function State_Of(T : Trains.Train_ID) return C_States
   --# global in     Current_State;
   is
   begin
      return Current_State(T).C_State;
   end State_Of;

   function Position(T : Trains.Train_ID) return Track.Station_ID
   --# global in     Current_State;
   is
   begin
      return Current_State(T).At_St;
   end Position;

   function Next(T : Trains.Train_ID) return Track.Station_T
   --# global in     Current_State;
   is
   begin
      return Current_State(T).Next_St;
   end Next;

```

```
259: end Control;
```

```
    1: ------------------------------------------------------------------------------
    2: -- Controller main program
    3: -- Version 1.0, April 2006
    4: --
    5: --  Copyright (C) 2006 Phil Thornley - phil@sparksure.com
    6: --
    7: --  Produced in cooperation with Bernd Sieker and Peter Ladkin,
    8: --                               University of Bielefeld
    9: --
   10: --  This program is free software; you can redistribute it and/or modify
   11: --  it under the terms of the GNU General Public License as published by
   12: --  the Free Software Foundation; either version 2 of the License, or
   13: --  (at your option) any later version.
   14: ------------------------------------------------------------------------------
   15:
   16: with Trains;
   17: with Control;
   18: with Messages.FA_M;
   19: with Messages.AM_M;
   20: --# inherit Trains,
   21: --#         Control,
   22: --#         Messages.FA_M,
   23: --#         Messages.AM_M;
   24: --# main_program;
   25: procedure Controller
   26: --# global    out Control.C_State;
   27: --#        in     Messages.In_Queues;
   28: --#           out Messages.Out_Queues;
   29: --# derives Control.C_State,
   30: --#         Messages.Out_Queues from
   31: --#            Messages.In_Queues;
   32: is
   33:    FA_Message : Messages.FA_M.FA_Message_T;
   34:    AM_Message : Messages.AM_M.AM_Message_T;
   35:
   36:    Messages_Waiting  : Boolean;
   37:    Next_Message_Type : Messages.MType_T;
   38:
   39: begin
   40:
   41:    Control.Initialize;
   42:
   43:    loop
   44:       loop
   45:          Messages_Waiting := Messages.Message_Waiting_For(0);
   46:          exit when not Messages_Waiting;
   47:          Next_Message_Type := Messages.Next_Message_Type(0);
   48:          case Next_Message_Type is
   49:             when Messages.FA =>
   50:                Messages.FA_M.Receive (FA_Message, 0);
   51:                Control.Process_FA (FA_Message,
   52:                                    Messages.FA_M.Train_Of(FA_Message));
   53:             when Messages.AM =>
   54:                Messages.AM_M.Receive (AM_Message, 0);
   55:                Control.Process_AM (Messages.AM_M.Train_Of(AM_Message));
   56:             when Messages.AFE |
   57:                  Messages.FE  => null;
   58:          end case;
   59:       end loop;
   60:       for I in Trains.Train_ID loop
   61:          Control.Check_Waiting_Train (I);
   62:          Control.Prepare_Next_Move (I);
   63:       end loop;
   64:       Control.Pass_Time;
   65:    end loop;
   66:
   67: end Controller;
```

```
    1: ------------------------------------------------------------------------------
    2: -- Drivers package specification
    3: -- Version 1.0, April 2006
    4: --
    5: --  Copyright (C) 2006 Phil Thornley - phil@sparksure.com
    6: --
    7: --  Produced in cooperation with Bernd Sieker and Peter Ladkin,
    8: --                               University of Bielefeld
    9: --
   10: --  This program is free software; you can redistribute it and/or modify
   11: --  it under the terms of the GNU General Public License as published by
   12: --  the Free Software Foundation; either version 2 of the License, or
   13: --  (at your option) any later version.
   14: ------------------------------------------------------------------------------
   15:
   16: with Trains;
   17: with Track;
   18: with Messages.FA_M;
   19: with Messages.AFE_M;
   20: with Messages.FE_M;
   21: with Messages.AM_M;
   22: --# inherit Trains,
   23: --#         Track.Layout,
   24: --#         Messages.FA_M,
   25: --#         Messages.AFE_M,
   26: --#         Messages.FE_M,
   27: --#         Messages.AM_M,
   28: --#         Configuration;
   29: package Drivers
   30: is
   31:
   32:    type D_States is (D_S0, D_S1, D_Sx, D_S6, D_S8, D_S9, D_S10, D_S11);
   33:
   34:    type Driver_State is private;
   35:
   36:    procedure Initialize (DS :    out Driver_State;
   37:                          ID : in     Trains.Train_ID);
   38:    --# derives DS from
   39:    --#            ID;
   40:
   41:    function D_State (DS : Driver_State) return D_States;
   42:
   43:    function My_ID (DS : Driver_State) return Trains.Train_ID;
   44:
   45:    function My_Station (DS : Driver_State) return Track.Station_ID;
   46:
   47:    --# function To_S1 (DS_Imp, DS_Exp : Driver_State) return Boolean;
   48:    --#
   49:    --# function To_Sx (DS_Imp, DS_Exp : Driver_State) return Boolean;
   50:    --#
   51:    --# function Timed_Out (DS: Driver_State) return Boolean;
   52:    --#
   53:    --# function To_S6 (DS_Imp, DS_Exp : Driver_State) return Boolean;
   54:    --#
   55:    --# function To_S8 (DS_Imp, DS_Exp : Driver_State) return Boolean;
   56:    --#
   57:    --# function Completed_Move (DS: Driver_State; TS : Track.TP_T) return Boolean;
   58:    --#
   59:    --# function To_S9 (DS_Imp, DS_Exp : Driver_State) return Boolean;
   60:    --#
   61:    --# function To_S11 (DS_Imp, DS_Exp : Driver_State) return Boolean;
   62:    --#
   63:    --# function To_S0 (DS_Imp, DS_Exp : Driver_State) return Boolean;
   64:
   65:    procedure Send_FA (DS : in out Driver_State);
   66:    --# global in     Messages.In_Queues;
   67:    --#           out Messages.Out_Queues;
   68:    --# derives Messages.Out_Queues from
   69:    --#            DS,
   70:    --#            Messages.In_Queues
   71:    --# &       DS from
   72:    --#            *;
   73:    --# pre  D_State(DS) = D_S0;
   74:    --# post To_S1(DS~, DS);
   75:
   76:    procedure Process_AFE (Message : in     Messages.AFE_M.AFE_Message_T;
   77:                           DS      : in out Driver_State);
   78:    --# derives DS from
   79:    --#            *,
   80:    --#            Message;
   81:    --# post (D_State(DS~)  = D_S1 -> To_Sx(DS~, DS))    and
   82:    --#      (D_State(DS~) /= D_S1 -> DS~ = DS);
   83:
   84:    procedure Process_FE (Message : in     Messages.FE_M.FE_Message_T;
   85:                          DS      : in out Driver_State);
   86:    --# derives DS from
```

```
   --#            *,
   --#            Message;
   --# pre  D_State(DS) = D_S0 or
   --#      D_State(DS) = D_Sx;
   --# post To_S6(DS~, DS);

   procedure Start_Move (DS : in out Driver_State);
   --# derives DS from
   --#            *;
   --# pre  D_State(DS) = D_S6;
   --# post To_S8(DS~, DS);

   procedure Check_End_Move (DS : in out Driver_State);
   --# global in Track.Train_Position;
   --# derives DS from
   --#            *,
   --#            Track.Train_Position;
   --# pre  D_State(DS) = D_S8;
   --# post (    Completed_Move(DS~, Track.Train_Position) -> To_S9(DS~, DS))    and
   --#      (not Completed_Move(DS~, Track.Train_Position) -> DS~ = DS);

   procedure Send_AM (DS : in out Driver_State);
   --# global in     Messages.In_Queues;
   --#           out Messages.Out_Queues;
   --# derives DS from
   --#            *
   --# &       Messages.Out_Queues from
   --#            Messages.In_Queues,
   --#            DS;
   --# pre  D_State(DS) = D_S9;
   --# post To_S11(DS~, DS);

   procedure Prepare_Next_Block (DS : in out Driver_State);
   --# derives DS from
   --#            *;
   --# pre  D_State(DS) = D_S11;
   --# post To_S0(DS~, DS);

   procedure Pass_Time;
   --# derives ;

private

   type Move_Data is
      record
         Active  : Boolean;
         Next_St : Track.Station_ID;
      end record;

   type Driver_State is
      record
         My_Unit  : Trains.Train_ID;
         My_State : D_States;
         Move     : Move_Data;
      end record;

end Drivers;

```

```
    1: ------------------------------------------------------------------------------
    2: -- Drivers package body
    3: -- Version 1.0, April 2006
    4: --
    5: --  Copyright (C) 2006 Phil Thornley - phil@sparksure.com
    6: --
    7: --  Produced in cooperation with Bernd Sieker and Peter Ladkin,
    8: --                               University of Bielefeld
    9: --
   10: --  This program is free software; you can redistribute it and/or modify
   11: --  it under the terms of the GNU General Public License as published by
   12: --  the Free Software Foundation; either version 2 of the License, or
   13: --  (at your option) any later version.
   14: ------------------------------------------------------------------------------
   15:
   16: with Configuration;
   17: with Track.Layout;
   18: package body Drivers
   19: is
   20:
   21:    Timeout_Delay : constant := 10;
   22:
   23:    --# inherit Trains,
   24:    --#         Track;
   25:    package Report
   26:    is
   27:
   28:       procedure Action (U : Trains.Train_ID; St : Track.Station_ID; Text : String);
   29:       --# derives null from
   30:       --#            U,
   31:       --#            St,
   32:       --#            Text;
   33:
   34:    end Report;
   35:
   36:    package body Report is separate;
   37:
   38:
   39:    function Get_Next_Station (Now_At : Track.Station_ID) return Track.Station_ID
   40:    is
   41:    begin
   42:       return Track.Layout.Next_St(Now_At);
   43:    end Get_Next_Station;
   44:
   45:    procedure Initialize (DS :    out Driver_State;
   46:                          ID : in     Trains.Train_ID)
   47:    is
   48:    begin
   49:       DS := Driver_State'
   50:             (My_Unit  => ID,
   51:              My_State => D_S11,
   52:              Move     => Move_Data'(Active  => False,
   53:                                     Next_St => Configuration.Starting_Stations(ID)));
   54:    end Initialize;
   55:
   56:    function D_State (DS : Driver_State) return D_States
   57:    is
   58:    begin
   59:       return DS.My_State;
   60:    end D_State;
   61:
   62:    function My_ID (DS : Driver_State) return Trains.Train_ID
   63:    is
   64:    begin
   65:       return DS.My_Unit;
   66:    end My_ID;
   67:
   68:    function My_Station (DS : Driver_State) return Track.Station_ID
   69:    is
   70:    begin
   71:       return DS.Move.Next_St;
   72:    end My_Station;
   73:
   74:    procedure Send_FA (DS : in out Driver_State)
   75:    is
   76:       FA_Message : Messages.FA_M.FA_Message_T;
   77:    begin
   78:       Report.Action (DS.My_Unit, DS.Move.Next_St, " Send FA");
   79:       Messages.FA_M.Initialize (FA_Message,
   80:                                 DS.My_Unit,
   81:                                 DS.Move.Next_St);
   82:       Messages.FA_M.Send (FA_Message);
   83:       DS.My_State := D_S1;
   84:    end Send_FA;
   85:
   86:    procedure Process_AFE (Message : in     Messages.AFE_M.AFE_Message_T;
```

```
   87:                          DS      : in out Driver_State)
   88:    is
   89:    begin
   90:       if DS.My_State = D_S1 then
   91:          DS.My_State := D_Sx;
   92:       end if;
   93:    end Process_AFE; -- Expect Flow errors 30 and 50 here - Message imported but not used.
   94:
   95:    procedure Process_FE (Message : in     Messages.FE_M.FE_Message_T;
   96:                          DS      : in out Driver_State)
   97:    is
   98:    begin
   99:       Report.Action (DS.My_Unit, DS.Move.Next_St, " Receive FE");
  100:       DS.My_State := D_S6;
  101:    end Process_FE; -- Expect Flow errors 30 and 50 here - Message imported but not used.
  102:
  103:    procedure Start_Move (DS : in out Driver_State)
  104:    is
  105:    begin
  106:       Report.Action (DS.My_Unit, DS.Move.Next_St, " Start move");
  107:       DS.My_State := D_S8;
  108:       DS.Move.Active := True;
  109:    end Start_Move;
  110:
  111:    procedure Check_End_Move (DS : in out Driver_State)
  112:    is
  113:       Arrived : Boolean;
  114:    begin
  115:       Track.At_Station (Arrived);
  116:       if Arrived then
  117:          DS.My_State    := D_S9;
  118:          DS.Move.Active := False;
  119:       end if;
  120:    end Check_End_Move;
  121:
  122:    procedure Send_AM (DS : in out Driver_State)
  123:    is
  124:       AM_Message : Messages.AM_M.AM_Message_T;
  125:    begin
  126:       Report.Action (DS.My_Unit, DS.Move.Next_St, " Send AM");
  127:       Messages.AM_M.Initialize (AM_Message,
  128:                                 DS.My_Unit,
  129:                                 DS.Move.Next_St);
  130:       Messages.AM_M.Send (AM_Message);
  131:       DS.My_State := D_S11;
  132:    end Send_AM;
  133:
  134:    procedure Prepare_Next_Block (DS : in out Driver_State)
  135:    is
  136:    begin
  137:       DS.My_State := D_S0;
  138:       DS.Move.Next_St := Get_Next_Station (DS.Move.Next_St);
  139:       Report.Action (DS.My_Unit, DS.Move.Next_St, " Prepared");
  140:    end Prepare_Next_Block;
  141:
  142:    procedure Pass_Time
  143:    is
  144:    --# hide Pass_Time;
  145:    begin
  146:       delay 0.0;
  147:    end Pass_Time;
  148:
  149: end Drivers;
  150:
```

```
    1: ------------------------------------------------------------------------------
    2: -- A_Driver main program
    3: -- Version 1.0, April 2006
    4: --
    5: --  Copyright (C) 2006 Phil Thornley - phil@sparksure.com
    6: --
    7: --  Produced in cooperation with Bernd Sieker and Peter Ladkin,
    8: --                               University of Bielefeld
    9: --
   10: --  This program is free software; you can redistribute it and/or modify
   11: --  it under the terms of the GNU General Public License as published by
   12: --  the Free Software Foundation; either version 2 of the License, or
   13: --  (at your option) any later version.
   14: ------------------------------------------------------------------------------
   15:
   16: with Ada.Text_IO;  -- Expect Warning 1 - Ada is not visible.
   17:
   18: with Drivers;           use type Drivers.D_States;
   19: with Trains;
   20: with Messages;
   21: with Messages.AFE_M;
   22: with Messages.FE_M;
   23: --# inherit Drivers,
   24: --#         Trains,
   25: --#         Track,
   26: --#         Messages,
   27: --#         Messages.AFE_M,
   28: --#         Messages.FE_M;
   29: --# main_program;
   30: procedure A_Driver(My_Train_ID : Trains.Train_ID)
   31: --# global in     Messages.In_Queues;
   32: --#           out Messages.Out_Queues;
   33: --#        in     Track.Train_Position;
   34: --# derives Messages.Out_Queues from
   35: --#            Messages.In_Queues,
   36: --#            Track.Train_Position,
   37: --#            My_Train_ID;
   38: is
   39:
   40:    My_State : Drivers.Driver_State;
   41:
   42:    My_Unit_ID : Messages.Unit_T;
   43:
   44:    Message_Waiting   : Boolean;
   45:    Next_Message_Type : Messages.MType_T;
   46:
   47:    FE_Message  : Messages.FE_M.FE_Message_T;
   48:    AFE_Message : Messages.AFE_M.AFE_Message_T;
   49:
   50:    procedure Report_AFE (U : Messages.Unit_T)
   51:    --# derives Null from U;
   52:    is
   53:    --# hide Report_AFE;
   54:    begin
   55:       Ada.Text_IO.Put_Line (Messages.Unit_T'Image(My_Unit_ID) & " Receive AFE");
   56:    end Report_AFE;
   57:
   58: begin
   59:    My_Unit_ID := Messages.Unit_T(My_Train_ID);
   60:    Drivers.Initialize (My_State, Trains.Train_ID(My_Unit_ID));
   61:    loop
   62:       Message_Waiting := Messages.Message_Waiting_For(My_Unit_ID);
   63:       if Message_Waiting then
   64:          Next_Message_Type := Messages.Next_Message_Type(My_Unit_ID);
   65:          case Next_Message_Type is
   66:             when Messages.FE =>
   67:                Messages.FE_M.Receive (FE_Message,
   68:                                       My_Unit_ID);
   69:                if Drivers.D_State (My_State) = Drivers.D_S1 or
   70:                  Drivers.D_State (My_State) = Drivers.D_Sx
   71:                then
   72:                   Drivers.Process_FE (FE_Message,
   73:                                       My_State);
   74:                end if;
   75:             when Messages.AFE =>
   76:                Report_AFE(My_Unit_ID);
   77:                Messages.AFE_M.Receive (AFE_Message,
   78:                                        My_Unit_ID);
   79:                Drivers.Process_AFE (AFE_Message,
   80:                                     My_State);
   81:             when Messages.FA |
   82:                  Messages.AM =>
   83:                null;
   84:          end case;
   85:       end if;
   86:       case Drivers.D_State(My_State) is
```

```
   87:          when Drivers.D_S0 =>
   88:             Drivers.Send_FA (My_State);
   89:          when Drivers.D_S1 |
   90:               Drivers.D_Sx =>
   91:             null;
   92:          when Drivers.D_S6 =>
   93:             Drivers.Start_Move (My_State);
   94:          when Drivers.D_S8 =>
   95:             Drivers.Check_End_Move (My_State);
   96:          when Drivers.D_S9 =>
   97:             Drivers.Send_AM (My_State);
   98:          when Drivers.D_S10 =>
   99:             null;
  100:          when Drivers.D_S11 =>
  101:             Drivers.Prepare_Next_Block (My_State);
  102:       end case;
  103:       Drivers.Pass_Time;
  104:    end loop;
  105:
  106: end A_Driver;
  107:
```

```
 1: with Trains;
 2: with Track;
 3: --# inherit Trains,
 4: --#         Track;
 5: package Configuration
 6: is
 7:
 8:    type St_St_T is array (Trains.Train_ID) of Track.Station_ID;
 9:
10:    Starting_Stations : constant St_St_T := St_St_T'(1, 8, 15, 22);
11:
12: end Configuration;
```

```
 1: with Ada.Text_IO;
 2: separate (Drivers)
 3: package body Report is
 4:
 5:    function Spaces (S : in     Trains.Train_ID) return String
 6:    is
 7:       Space : String (1 .. Integer(S) * 10) := (others => ' ');
 8:    begin
 9:       return Space;
10:    end Spaces;
11:
12:    procedure Action (U : Trains.Train_ID; St : Track.Station_ID; Text : String)
13:    is
14:    begin
15:       Ada.Text_IO.Put_Line (Spaces(U) & Trains.Train_ID'Image(U) & Text &
16:         Track.Station_ID'Image(St));
17:    end Action;
18:
19: end Report;
```

Anhang A.2: Beweise der Korrektheit der SPARK-Implementation von Phil Thornley

```
rule_family control_rules:
   X requires   [X:any].


control_rules(1): state_of(A, B) may_be_replaced_by fld_c_state(element(fld_current_state(B), [A]))
                  if [1<=A, A<=4].

control_rules(2): no_change(A, B) may_be_replaced_by
                    fld_current_state(A) = fld_current_state(B).

control_rules(3): to_s2(A, B, C) may_be_replaced_by
                    fld_c_state(element(fld_current_state(C), [A])) = c_s2 and
                    fld_next_st(element(fld_current_state(C), [A])) > 0    and
                    for_all(t : trains__train_id, 1<=t and t<=4 and t <> A ->
                            element(fld_current_state(B), [t]) = element(fld_current_state(C), [t]))
                  if [1<=A, A<=4].

control_rules(4): to_s4(A, B, C) may_be_replaced_by
                    element(fld_current_state(C), [A]) =
                       mk__train_rec(c_state := c_s4,
                                     at_st   := fld_at_st(element(fld_current_state(B), [A])),
                                     next_st := fld_next_st(element(fld_current_state(B), [A])))
                    and
                    for_all(t : trains__train_id, 1<=t and t<=4 and t <> A ->
                            element(fld_current_state(B), [t]) = element(fld_current_state(C), [t]))
                  if [1<=A, A<=4].
control_rules(5): to_s5(A, B, C) may_be_replaced_by
                    element(fld_current_state(C), [A]) =
                       mk__train_rec(c_state := c_s5,
                                     at_st   := fld_at_st(element(fld_current_state(B), [A])),
                                     next_st := fld_next_st(element(fld_current_state(B), [A])))
                    and
                    for_all(t : trains__train_id, 1<=t and t<=4 and t <> A ->
                            element(fld_current_state(B), [t]) = element(fld_current_state(C), [t]))
                  if [1<=A, A<=4].

control_rules(6): to_s12(A, B, C) may_be_replaced_by
                    element(fld_current_state(C), [A]) =
                       mk__train_rec(c_state := c_s12,
                                     at_st   := fld_next_st(element(fld_current_state(B), [A])),
                                     next_st := 0)
                         and
                    for_all(t : trains__train_id, 1<=t and t<=4 and t <> A ->
                            element(fld_current_state(B), [t]) = element(fld_current_state(C), [t]))
                  if [1<=A, A<=4].

control_rules(7): train_is_waiting(A, B) may_be_replaced_by
                    fld_c_state(element(fld_current_state(B), [A])) = c_s2 or
                    fld_c_state(element(fld_current_state(B), [A])) = c_s5
                  if [1<=A, A<=4].

control_rules(8): next_is_set(A, B) may_be_replaced_by
                    fld_next_st(element(fld_current_state(B), [A])) > 0
                  if [1<=A, A<=4].

control_rules(9): next(A, B) may_be_replaced_by
                    fld_next_st(element(fld_current_state(B), [A]))
                  if [1<=A, A<=4].

control_rules(10): control__kh(A, B) may_be_replaced_by kh(A, fld_current_state(B))
                   if [1<=A, A<=4].

control_rules(11): no_train_at(A, B) may_be_replaced_by
                     for_all(t : trains__train_id, 1<=t and t<=4 ->
                             fld_at_st(element(fld_current_state(B), [t])) <> A)
                   if [1<=A, A<=25].

control_rules(12): no_train_cleared_to(A, B) may_be_replaced_by
                     for_all(t : trains__train_id, 1<=t and t<=4 ->
                             (fld_c_state(element(fld_current_state(B), [t])) = c_s4 ->
                               fld_next_st(element(fld_current_state(B), [t])) <> A))
                   if [1<=A, A<=25].

```

```
rule_family driver_rules:
   X requires [X:any].

driver_rules(1): d_state(A) may_be_replaced_by fld_my_state(A).

driver_rules(2): my_id(A) may_be_replaced_by fld_my_unit(A).

driver_rules(3): my_station(A) may_be_replaced_by fld_next_st(fld_move(A)).

driver_rules(4): to_s1(A, B) may_be_replaced_by
                              B = upf_my_state(A, d_s1).

driver_rules(5): to_sx(A, B) may_be_replaced_by
                              B = upf_my_state(A, d_sx).

driver_rules(6): to_s6(A, B) may_be_replaced_by
                              B = upf_my_state(A, d_s6).

driver_rules(7): to_s8(A, B) may_be_replaced_by
                   B = upf_move(upf_my_state(A, d_s8),
                                upf_active(fld_move(A), true)).

driver_rules(8): to_s9(A, B) may_be_replaced_by
                    B = upf_move(upf_my_state(A, d_s9),
                                 upf_active(fld_move(A), false)).

driver_rules(9): completed_move(A, B) may_be_replaced_by track__is_at_station(B).

driver_rules(10): to_s0(A, B) may_be_replaced_by
                    B = upf_move(upf_my_state(A, d_s0),
                                 upf_next_st(fld_move(A),
                                             get_next_station(fld_next_st(fld_move(A))))).

driver_rules(11): to_s11(A, B) may_be_replaced_by
                                 B = upf_my_state(A, d_s11).

```

```
    1: ------------------------------------------------------------------------------
    2:                           Semantic Analysis Summary
    3:              SPARK Proof Obligation Summariser Release 4.4 / 09.05
    4:                  Praxis High Integrity Systems, Bath, England
    5: ------------------------------------------------------------------------------
    6:
    7: Summary of:
    8:
    9: Verification Condition files (.vcg)
   10: Simplified Verification Condition files (.siv)
   11: Proof Logs (.plg)
   12:
   13: in the directory:
   14: G:\Phil\SPARK\Train_Dispatch\Code
   15:
   16: Summary produced: 15-APR-2006 12:11:04.46
   17:
   18: File g:\phil\spark\train_dispatch\code\controller\control\check_waiting_train.vcg
   19: procedure Control.Check_Waiting_Train
   20:
   21: VCs generated 15-APR-2006 12:10:48
   22:
   23: VCs simplified 15-APR-2006 12:10:50
   24:
   25: VCs proved 15-APR-2006 12:11:00
   26:
   27: VCs for procedure_check_waiting_train :
   28: ------------------------------------------------------------------------
   29:      |       |                    |  -----Proved In-----  |       |       |
   30:  #   | From  | To                 | vcg | siv | plg | prv | False | TO DO |
   31: ------------------------------------------------------------------------
   32:  1   | start | rtc check @ 201    |     | YES |     |     |       |       |
   33:  2   | start | pre check @ 202    |     | YES |     |     |       |       |
   34:  3   | start | rtc check @ 203    |     | YES |     |     |       |       |
   35:  4   | start | rtc check @ 205    |     | YES |     |     |       |       |
   36:  5   | start | rtc check @ 208    |     | YES |     |     |       |       |
   37:  6   | start | rtc check @ 210    |     | YES |     |     |       |       |
   38:  7   | start | pre check @ 211    |     | YES |     |     |       |       |
   39:  8   | start | rtc check @ 212    |     | YES |     |     |       |       |
   40:  9   | start | rtc check @ 214    |     | YES |     |     |       |       |
   41:  10  | start |    assert @ finish |     |     | YES |     |       |       |
   42:  11  | start |    assert @ finish |     |     | YES |     |       |       |
   43:  12  | start |    assert @ finish |     |     | YES |     |       |       |
   44:  13  | start |    assert @ finish |     | YES |     |     |       |       |
   45:  14  | start |    assert @ finish |     | YES |     |     |       |       |
   46:  15  |       | refinement         |     | YES |     |     |       |       |
   47:  16  |       | refinement         |     |     | YES |     |       |       |
   48: ------------------------------------------------------------------------
   49:
   50:
   51: File g:\phil\spark\train_dispatch\code\controller\control\check_waiting_train\send_afe.vcg
   52: procedure Control.Check_Waiting_Train.Send_AFE
   53:
   54: VCs generated 15-APR-2006 12:10:48
   55:
   56: VCs simplified 15-APR-2006 12:10:50
   57:
   58: VCs for procedure_send_afe :
```

```
   59: --------------------------------------------------------------------------
   60:       |       |                    |  -----Proved In-----  |       |       |
   61:  #    | From  | To                 | vcg | siv | plg | prv | False | TO DO |
   62: --------------------------------------------------------------------------
   63:  1    | start |    assert @ finish | YES |     |     |     |       |       |
   64: --------------------------------------------------------------------------
   65:
   66:
   67: File g:\phil\spark\train_dispatch\code\controller\control\check_waiting_train\send_fe.vcg
   68: procedure Control.Check_Waiting_Train.Send_FE
   69:
   70: VCs generated 15-APR-2006 12:10:48
   71:
   72: VCs simplified 15-APR-2006 12:10:50
   73:
   74: VCs for procedure_send_fe :
   75: --------------------------------------------------------------------------
   76:       |       |                    |  -----Proved In-----  |       |       |
   77:  #    | From  | To                 | vcg | siv | plg | prv | False | TO DO |
   78: --------------------------------------------------------------------------
   79:  1    | start | rtc check @ 184    |     | YES |     |     |       |       |
   80:  2    | start |    assert @ finish | YES |     |     |     |       |       |
   81: --------------------------------------------------------------------------
   82:
   83:
   84: File g:\phil\spark\train_dispatch\code\controller\control\initialize.vcg
   85: procedure Control.Initialize
   86:
   87: VCs generated 15-APR-2006 12:10:48
   88:
   89: VCs simplified 15-APR-2006 12:10:52
   90:
   91: VCs for procedure_initialize :
   92: --------------------------------------------------------------------------
   93:       |       |                    |  -----Proved In-----  |       |       |
   94:  #    | From  | To                 | vcg | siv | plg | prv | False | TO DO |
   95: --------------------------------------------------------------------------
   96:  1    | start |    assert @ 43     |     | YES |     |     |       |       |
   97:  2    | 43    |    assert @ 43     |     | YES |     |     |       |       |
   98:  3    | 43    | rtc check @ 44     |     | YES |     |     |       |       |
   99:  4    | 43    |    assert @ finish | YES |     |     |     |       |       |
  100:  5    |       | refinement         | YES |     |     |     |       |       |
  101:  6    |       | refinement         | YES |     |     |     |       |       |
  102: --------------------------------------------------------------------------
  103:
  104:
  105: File g:\phil\spark\train_dispatch\code\controller\control\kh.vcg
  106: function Control.KH
  107:
  108: VCs generated 15-APR-2006 12:10:48
  109:
  110: VCs simplified 15-APR-2006 12:10:52
  111:
  112: VCs proved 15-APR-2006 12:11:01
  113:
  114: VCs for function_kh :
  115: --------------------------------------------------------------------------
  116:       |       |                    |  -----Proved In-----  |       |       |
```

#	From	To	vcg	siv	plg	prv	False	TO DO
1	start	rtc check @ 108		YES				
2	start	rtc check @ 110		YES				
3	120	rtc check @ 110		YES				
4	start	rtc check @ 114		YES				
5	120	rtc check @ 114		YES				
6	start	assert @ 120			YES			
7	120	assert @ 120			YES			
8	start	assert @ finish			YES			
9	start	assert @ finish			YES			
10	120	assert @ finish		YES				
11	120	assert @ finish			YES			
12	120	assert @ finish			YES			
13		refinement		YES				
14		refinement			YES			

File g:\phil\spark\train_dispatch\code\controller\control\next.vcg
function Control.Next

VCs generated 15-APR-2006 12:10:48

VCs simplified 15-APR-2006 12:10:54

VCs for function_next :

			-----Proved In-----					
#	From	To	vcg	siv	plg	prv	False	TO DO
1	start	rtc check @ 256		YES				
2	start	assert @ finish		YES				
3		refinement	YES					
4		refinement	YES					

File g:\phil\spark\train_dispatch\code\controller\control\position.vcg
function Control.Position

VCs generated 15-APR-2006 12:10:48

VCs simplified 15-APR-2006 12:10:54

VCs for function_position :

			-----Proved In-----					
#	From	To	vcg	siv	plg	prv	False	TO DO
1	start	rtc check @ 249		YES				
2	start	assert @ finish		YES				
3		refinement	YES					
4		refinement	YES					

File g:\phil\spark\train_dispatch\code\controller\control\prepare_next_move.vcg

```
procedure Control.Prepare_Next_Move

VCs generated 15-APR-2006 12:10:48

VCs simplified 15-APR-2006 12:10:54

VCs for procedure_prepare_next_move :
------------------------------------------------------------------------------
      |       |                    |  -----Proved In-----  |       |       |
 #    | From  | To                 | vcg | siv | plg | prv | False | TO DO |
------------------------------------------------------------------------------
 1    | start | rtc check @ 226    |     | YES |     |     |       |       |
 2    | start | rtc check @ 227    |     | YES |     |     |       |       |
 3    | start |    assert @ finish | YES |     |     |     |       |       |
 4    | start |    assert @ finish | YES |     |     |     |       |       |
 5    |       | refinement         | YES |     |     |     |       |       |
 6    |       | refinement         | YES |     |     |     |       |       |
------------------------------------------------------------------------------


File g:\phil\spark\train_dispatch\code\controller\control\process_am.vcg
procedure Control.Process_AM

VCs generated 15-APR-2006 12:10:48

VCs simplified 15-APR-2006 12:10:54

VCs proved 15-APR-2006 12:11:03

VCs for procedure_process_am :
------------------------------------------------------------------------------
      |       |                    |  -----Proved In-----  |       |       |
 #    | From  | To                 | vcg | siv | plg | prv | False | TO DO |
------------------------------------------------------------------------------
 1    | start | rtc check @ 89     |     | YES |     |     |       |       |
 2    | start | rtc check @ 90     |     | YES |     |     |       |       |
 3    | start |    assert @ finish |     | YES |     |     |       |       |
 4    | start |    assert @ finish |     | YES |     |     |       |       |
 5    |       | refinement         |     | YES |     |     |       |       |
 6    |       | refinement         |     |     | YES |     |       |       |
------------------------------------------------------------------------------


File g:\phil\spark\train_dispatch\code\controller\control\process_fa.vcg
procedure Control.Process_FA

VCs generated 15-APR-2006 12:10:48

VCs simplified 15-APR-2006 12:10:55

VCs proved 15-APR-2006 12:11:03

VCs for procedure_process_fa :
------------------------------------------------------------------------------
      |       |                    |  -----Proved In-----  |       |       |
 #    | From  | To                 | vcg | siv | plg | prv | False | TO DO |
------------------------------------------------------------------------------
 1    | start | rtc check @ 66     |     | YES |     |     |       |       |
```

```
  2    | start | rtc check @ 67      |     | YES |     |     |       |       |
  3    | start | rtc check @ 68      |     | YES |     |     |       |       |
  4    | start | rtc check @ 68      |     | YES |     |     |       |       |
  5    | start |    assert @ finish  |     | YES |     |     |       |       |
  6    | start |    assert @ finish  |     | YES |     |     |       |       |
  7    |       | refinement          | YES |     |     |     |       |       |
  8    |       | refinement          |     |     | YES |     |       |       |
 ------------------------------------------------------------------------------


 File g:\phil\spark\train_dispatch\code\controller\control\state_of.vcg
 function Control.State_Of

 VCs generated 15-APR-2006 12:10:48

 VCs simplified 15-APR-2006 12:10:56

 VCs for function_state_of :
 ------------------------------------------------------------------------------
      |       |                     |  -----Proved In-----  |       |       |
  #   | From  | To                  | vcg | siv | plg | prv | False | TO DO |
 ------------------------------------------------------------------------------
  1    | start | rtc check @ 242     |     | YES |     |     |       |       |
  2    | start |    assert @ finish  |     | YES |     |     |       |       |
  3    |       | refinement          | YES |     |     |     |       |       |
  4    |       | refinement          | YES |     |     |     |       |       |
 ------------------------------------------------------------------------------


 File g:\phil\spark\train_dispatch\code\controller\controller.vcg
 procedure controller

 VCs generated 15-APR-2006 12:10:48

 VCs for function_state_of :
 ------------------------------------------------------------------------------
      |       |                     |  -----Proved In-----  |       |       |
  #   | From  | To                  | vcg | siv | plg | prv | False | TO DO |
 ------------------------------------------------------------------------------
  1    | start | rtc check @ 242     |     | YES |     |     |       |       |
  2    | start |    assert @ finish  |     | YES |     |     |       |       |
  3    |       | refinement          | YES |     |     |     |       |       |
  4    |       | refinement          | YES |     |     |     |       |       |
 ------------------------------------------------------------------------------


 File g:\phil\spark\train_dispatch\code\driver\a_driver.vcg
 procedure A_Driver

 VCs generated 15-APR-2006 12:10:49

 VCs for function_state_of :
 ------------------------------------------------------------------------------
      |       |                     |  -----Proved In-----  |       |       |
  #   | From  | To                  | vcg | siv | plg | prv | False | TO DO |
 ------------------------------------------------------------------------------
  1    | start | rtc check @ 242     |     | YES |     |     |       |       |
  2    | start |    assert @ finish  |     | YES |     |     |       |       |
```

```
  3    |       | refinement          | YES |     |     |     |       |       |
  4    |       | refinement          | YES |     |     |     |       |       |
-------------------------------------------------------------------------------


File g:\phil\spark\train_dispatch\code\driver\drivers\check_end_move.vcg
procedure drivers.Check_End_Move

VCs generated 15-APR-2006 12:10:48

VCs simplified 15-APR-2006 12:10:56

VCs proved 15-APR-2006 12:11:03

VCs for procedure_check_end_move :
-------------------------------------------------------------------------------
      |       |                     |  -----Proved In-----  |       |       |
 #    | From  | To                  | vcg | siv | plg | prv | False | TO DO |
-------------------------------------------------------------------------------
 1    | start | rtc check @ 117     |     | YES |     |     |       |       |
 2    | start |    assert @ finish  |     |     | YES |     |       |       |
 3    | start |    assert @ finish  |     | YES |     |     |       |       |
-------------------------------------------------------------------------------


File g:\phil\spark\train_dispatch\code\driver\drivers\d_state.vcg
function drivers.D_State

VCs generated 15-APR-2006 12:10:48

VCs simplified 15-APR-2006 12:10:56

VCs for function_d_state :
-------------------------------------------------------------------------------
      |       |                     |  -----Proved In-----  |       |       |
 #    | From  | To                  | vcg | siv | plg | prv | False | TO DO |
-------------------------------------------------------------------------------
 1    | start |    assert @ finish  |     | YES |     |     |       |       |
-------------------------------------------------------------------------------


File g:\phil\spark\train_dispatch\code\driver\drivers\get_next_station.vcg
function drivers.Get_Next_Station

VCs generated 15-APR-2006 12:10:48

VCs simplified 15-APR-2006 12:10:57

VCs for function_get_next_station :
-------------------------------------------------------------------------------
      |       |                     |  -----Proved In-----  |       |       |
 #    | From  | To                  | vcg | siv | plg | prv | False | TO DO |
-------------------------------------------------------------------------------
 1    | start | rtc check @ 42      |     | YES |     |     |       |       |
 2    | start |    assert @ finish  |     | YES |     |     |       |       |
-------------------------------------------------------------------------------


```

```
File g:\phil\spark\train_dispatch\code\driver\drivers\initialize.vcg
procedure drivers.Initialize

VCs generated 15-APR-2006 12:10:48

VCs simplified 15-APR-2006 12:10:57

VCs for procedure_initialize :
------------------------------------------------------------------------------
      |       |                     |  -----Proved In-----  |       |       |
 #    | From  | To                  | vcg | siv | plg | prv | False | TO DO |
------------------------------------------------------------------------------
 1    | start | rtc check @ 49      |     | YES |     |     |       |       |
 2    | start |    assert @ finish  | YES |     |     |     |       |       |
------------------------------------------------------------------------------


File g:\phil\spark\train_dispatch\code\driver\drivers\my_id.vcg
function drivers.My_ID

VCs generated 15-APR-2006 12:10:48

VCs simplified 15-APR-2006 12:10:57

VCs for function_my_id :
------------------------------------------------------------------------------
      |       |                     |  -----Proved In-----  |       |       |
 #    | From  | To                  | vcg | siv | plg | prv | False | TO DO |
------------------------------------------------------------------------------
 1    | start |    assert @ finish  |     | YES |     |     |       |       |
------------------------------------------------------------------------------


File g:\phil\spark\train_dispatch\code\driver\drivers\my_station.vcg
function drivers.My_Station

VCs generated 15-APR-2006 12:10:48

VCs simplified 15-APR-2006 12:10:57

VCs for function_my_station :
------------------------------------------------------------------------------
      |       |                     |  -----Proved In-----  |       |       |
 #    | From  | To                  | vcg | siv | plg | prv | False | TO DO |
------------------------------------------------------------------------------
 1    | start |    assert @ finish  |     | YES |     |     |       |       |
------------------------------------------------------------------------------


File g:\phil\spark\train_dispatch\code\driver\drivers\prepare_next_block.vcg
procedure drivers.Prepare_Next_Block

VCs generated 15-APR-2006 12:10:49

VCs simplified 15-APR-2006 12:10:58

VCs for procedure_prepare_next_block :
------------------------------------------------------------------------------
```

```
      |       |                    |  -----Proved In-----  |       |       |
 #    | From  | To                 | vcg | siv | plg | prv | False | TO DO |
----------------------------------------------------------------------------
 1    | start | rtc check @ 137    |     | YES |     |     |       |       |
 2    | start | rtc check @ 138    |     | YES |     |     |       |       |
 3    | start | rtc check @ 138    |     | YES |     |     |       |       |
 4    | start | rtc check @ 139    |     | YES |     |     |       |       |
 5    | start |    assert @ finish |     | YES |     |     |       |       |
----------------------------------------------------------------------------


File g:\phil\spark\train_dispatch\code\driver\drivers\process_afe.vcg
procedure drivers.Process_AFE

VCs generated 15-APR-2006 12:10:48

VCs simplified 15-APR-2006 12:10:58

VCs proved 15-APR-2006 12:11:04

VCs for procedure_process_afe :
----------------------------------------------------------------------------
      |       |                    |  -----Proved In-----  |       |       |
 #    | From  | To                 | vcg | siv | plg | prv | False | TO DO |
----------------------------------------------------------------------------
 1    | start | rtc check @ 91     |     | YES |     |     |       |       |
 2    | start |    assert @ finish |     |     | YES |     |       |       |
 3    | start |    assert @ finish |     |     | YES |     |       |       |
----------------------------------------------------------------------------


File g:\phil\spark\train_dispatch\code\driver\drivers\process_fe.vcg
procedure drivers.Process_FE

VCs generated 15-APR-2006 12:10:48

VCs simplified 15-APR-2006 12:10:58

VCs for procedure_process_fe :
----------------------------------------------------------------------------
      |       |                    |  -----Proved In-----  |       |       |
 #    | From  | To                 | vcg | siv | plg | prv | False | TO DO |
----------------------------------------------------------------------------
 1    | start | rtc check @ 99     |     | YES |     |     |       |       |
 2    | start | rtc check @ 100    |     | YES |     |     |       |       |
 3    | start |    assert @ finish |     | YES |     |     |       |       |
----------------------------------------------------------------------------


File g:\phil\spark\train_dispatch\code\driver\drivers\send_am.vcg
procedure drivers.Send_AM

VCs generated 15-APR-2006 12:10:49

VCs simplified 15-APR-2006 12:10:58

VCs for procedure_send_am :
----------------------------------------------------------------------------
```

```
      |       |                    |  -----Proved In-----  |       |       |
 #    | From  | To                 | vcg | siv | plg | prv | False | TO DO |
----------------------------------------------------------------------------
 1    | start | rtc check @ 126    |     | YES |     |     |       |       |
 2    | start | rtc check @ 127    |     | YES |     |     |       |       |
 3    | start | rtc check @ 131    |     | YES |     |     |       |       |
 4    | start |    assert @ finish |     | YES |     |     |       |       |
----------------------------------------------------------------------------


File g:\phil\spark\train_dispatch\code\driver\drivers\send_fa.vcg
procedure drivers.Send_FA

VCs generated 15-APR-2006 12:10:48

VCs simplified 15-APR-2006 12:10:59

VCs for procedure_send_fa :
----------------------------------------------------------------------------
      |       |                    |  -----Proved In-----  |       |       |
 #    | From  | To                 | vcg | siv | plg | prv | False | TO DO |
----------------------------------------------------------------------------
 1    | start | rtc check @ 78     |     | YES |     |     |       |       |
 2    | start | rtc check @ 79     |     | YES |     |     |       |       |
 3    | start | rtc check @ 83     |     | YES |     |     |       |       |
 4    | start |    assert @ finish |     | YES |     |     |       |       |
----------------------------------------------------------------------------


File g:\phil\spark\train_dispatch\code\driver\drivers\start_move.vcg
procedure drivers.Start_Move

VCs generated 15-APR-2006 12:10:48

VCs simplified 15-APR-2006 12:10:59

VCs for procedure_start_move :
----------------------------------------------------------------------------
      |       |                    |  -----Proved In-----  |       |       |
 #    | From  | To                 | vcg | siv | plg | prv | False | TO DO |
----------------------------------------------------------------------------
 1    | start | rtc check @ 106    |     | YES |     |     |       |       |
 2    | start | rtc check @ 107    |     | YES |     |     |       |       |
 3    | start |    assert @ finish |     | YES |     |     |       |       |
----------------------------------------------------------------------------


================================================================================
Summary:

Total subprograms fully proved by examiner:               1
Total subprograms fully proved by simplifier:            18
Total subprograms fully proved by checker:                6
Total subprograms fully proved by review:                 0
Total subprograms with at least one false VC:             0
Total subprograms with at least one undischarged VC:      0
                                                        -----
Total subprograms for which VCs have been generated:     25
```

```
523:
524:
525: Total VCs by type:
526:                            -----------Proved By------------
527:                       Total  Examiner   Simp  Checker  Review  False  Undiscgd
528: Assert or Post:         42         6     24       12       0      0         0
529: Precondition check:      2         0      2        0       0      0         0
530: Check statement:         0         0      0        0       0      0         0
531: Runtime check:          45         0     45        0       0      0         0
532: Refinement VCs:         22        15      3        4       0      0         0
533: Inheritance VCs:         0         0      0        0       0      0         0
534: ================================================================================
535: Totals:                111        21     74       16       0      0         0
536: % Totals:                          18%    66%      14%      0%     0%        0%
537: ====================== End of Semantic Analysis Summary ========================
```

Anhang B.1: PROMELA-Source-Code

```
/*
   Model of modified Zugleitbetrieb for SPIN
*/


/* These can be modified for individual runs of the checker. */

#define TRAINS 3
#define STATIONS 4

/* array to store route of trains. A route is a list of stations.
   A route will never be longer than a complete circle
   minus one station, so a size of the number of stations
   if sufficient. */

typedef bytestationtype {
   byte station[STATIONS];
}


/* Array of bools of the size of the number of stations.
   Used in multi-dimensional arrays to store flags per train
   and per station. */

typedef boolstationtype {
   bool station[STATIONS];
}


/* array of bool of size of the number of stations,
   used as a type in various multi-dimensional arrays. */

typedef boolbetweentype {
   bool next[STATIONS];
}


/* two-dimensional bool array for a matrix of stations */

typedef bool2betweentype {
   boolbetweentype previous[STATIONS];
}


/* two-dimensional array of a series of stations for each train,
   defining that train's (intended) route */

bytestationtype Route[TRAINS];


/* two-dimensional array to indicate whether a train is in a station
   and whether for a train the Ankunftmeldung in a specific station
   has been transmitted and received. */

boolstationtype inZ[TRAINS] = false;


/* three-dimensional arrays to indicate that for a track segment between
   stations the corresponding message has been sent and received
   (FA, FE, AFE), the train actually is between the stations (zw),
   no obstructions are known (KH),
   the segment is occupied under central responsibility (ZV) */

bool2betweentype zw[TRAINS] = false,
                 KH[TRAINS] = false,
```

```
                ZV[TRAINS] = false;

/* arrays to indicate for each train:
   the next station on its trip,
   the currently occupied or recently-left station,
   the index pointing to the current station in its Route array */

byte Next[TRAINS],
     Current[TRAINS],
     Routeindex[TRAINS];

byte Routelast[TRAINS];


/* symbolic types for message types */

/* A message can be of type
   FA - Fahranfrage
   FE - Fahrerlaubnis
   AFE - Ablehnung Fahrerlaubnis
   AM - Ankunftmeldung */

mtype = {tFA, tFE, tAFE, tAM};


/* A Zuglaufmeldung is a triple, consisting of
   message type, train ID, and train destination.
   This is modelled after the original ZLM in
   the FV-NE standard. */

typedef Zuglaufmeldung {
    mtype type;
    byte train;
    byte destination;
}


/* type to set whether or not a station is occupied
   in either direction */

typedef directiontype {
   byte direction[2]
}


/* Communication channels:
   two for each train: bidirectional, synchronous (buffer length 0)
   c2t: controller to train
   t2c: train to controller
*/

chan c2t[TRAINS] = [0] of { Zuglaufmeldung };
chan t2c[TRAINS] = [0] of { Zuglaufmeldung };


/* A general purpose counter for enumerating trains */
byte counter;


init {

/* Create non-deterministic routes for all trains.
   Maximum one train per direction per station
   In Verification runs this will create all possible
   train combinations.
```

```
   In Simulation runs this will create a (pseudo-) random setup.
*/

   directiontype occupied[STATIONS];
   byte Routelength[TRAINS];
   byte train_counter = 0;
   byte routecounter;
   byte start;
   byte station;
   bit direction;

   do

   /* abort the loop if all trains have been handled */
   :: (train_counter == TRAINS) -> break;
   :: else ->

      /* start with an initial route length of 1 */
      Routelength[train_counter] = 1;

      /* non-determinasticially set direction for this train */
      if
      :: direction = 1;
      :: direction = 0;
      fi;

      /* initially set the start position to position 0 */
      start = 0;

      /* in a non-deterministic loop either increase or decrease
         the start position.
         Exit the loop non-deterministically if there is not already
         another train at this station in the same direction */
      do
      :: (start < STATIONS - 2 ) -> start++;
      :: (start > 0 ) -> start--;
      :: (!occupied[start].direction[direction]) -> break;
      od;
      occupied[start].direction[direction] = true;

      /* in a similar manner, non-deterministically set the length
         of this train's route. */
      do
      :: ( Routelength[train_counter] > 1 ) -> break;
      :: ( Routelength[train_counter] > 2 ) -> Routelength[train_counter]--;
      :: ( Routelength[train_counter] < STATIONS ) ->
         Routelength[train_counter]++;
      od;

      /* fill the route array for this train */
      station = start;
      Route[train_counter].station[0] = station;

      /* iterate over the length of the route */
      routecounter = 1;
      d_step {
         do

         /* if the iteration counter is equal to the route length, exit the loop */
         :: (routecounter == Routelength[train_counter]) -> break;
         :: else ->

            /* otherwise increment the route counter */

            /* depending on the train's direction, either increment or decrement
```

```
               the station variable (modulo number of stations) to subsequently
               fill the route array */
            if
            :: (direction == 1) -> station = (station + 1) % STATIONS;
            :: (direction == 0) -> station = (station + STATIONS - 1) % STATIONS;
            fi;
            Route[train_counter].station[routecounter] = station;
            routecounter++;
         od;
         /* set a special variable to indicate the final destination of the train */
         Routelast[train_counter] = Route[train_counter].station[Routelength[train_counter]
- 1];
         /* Routelast[train_counter] = station; */
         /* during validation (simulation) runs, print the train route data */
         routecounter = 0;
         do
         :: routecounter == Routelength[train_counter] -> break;
         :: else ->
            routecounter++;
         od;

      }

      /* increment the train counter, and enter the next iteration to make the route
      for the next train */
      train_counter++;
   od;


   /*********************
     Start the processes
    *********************/

   /* for every train start a driver and a controller process */

   train_counter = 0;
   atomic {
      do

      /* if the counter has reached the total number of trains, exit the loop */
      :: (train_counter == TRAINS) -> break;

      /* otherwise ... */
      :: else ->

         /* Set all Routeindex counters, Current and Next variables to appropriate values
            to avoid race conditions */

         Routeindex[train_counter] = 0;
         Current[train_counter] = Route[train_counter].station[0];
         Next[train_counter] = Route[train_counter].station[1];

         /* start the controller process for this train, with this train's route start and e
nd */
         printf("Starting Controller process for Train %d, going from %d to %d.\n",
                 train_counter, Route[train_counter].station[0], Routelast[train_counter]);
         run controller(train_counter, Route[train_counter].station[0], Routelast[train_coun
ter]);

         /* start the driver process for this train, with this train's route start and end *
/
         printf("Starting Driver process for Train %d, going from %d to %d.\n",
                 train_counter, Route[train_counter].station[0], Routelast[train_counter]);
         run driver(train_counter, Route[train_counter].station[0], Routelast[train_counter]
);
```

```
         /* increment the counter and enter the next iteration */
         train_counter++;
      od;
   } /* end atomic */

}


proctype driver(byte train, start, destination)
{

   /* initialisation */

   Zuglaufmeldung ZLM;

   /* start the main loop for the journey of one train from its
      start to its destination station. */

   do

   :: true ->


S0:
      /* Label S0 corresponds to state S0 in the driver's state machine
         of the Level 3 MFGs
         - Variables are set to indicate the train's next station
         - Flag is set to indicate that train is in its current station.

         In one deterministic step set the compononents of the message triple
         to the message type (FA), the train ID to the current train,
         and the destination to this train's current "Next" station.

         Send the message through the appropriate channel.

         This is the specification of the transition from S0 to S1,
         which follows unconditionally after sending the message.

         Note that since we use synchronous communications, this
         driver process will block until the controller has received
         the message.
      */

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

         /* send the compsed Fahranfrage-Message to the controller */
         t2c[train]!ZLM;
      } /* end atomic */


S1:
      /* Label S1 corresponds to state S1 in the driver's state machine
         of the Level 3 MFGs
         - FA is true whenever the FA-message has been sent and received. Since this
           process is unblocked only when the message has been received, this is now true.
         - Upon reception of a message from the controller, there will be a transition eithe
r
           . into S6, if the message type was FE
           . into Sx, if the message type was AFE
```

```
      */


      /* wait for message (blocks when using asynchronous communications) */

      c2t[train]?ZLM;

      if

      :: ZLM.type == tFE ->

         /* if message type was FE, then goto Label S6 */

         goto S6;

      :: ZLM.type == tAFE ->

         /* if the message type is AFE, then go to Label Sx.
            Since Sx just goes back to S1, this effectively does nothing,
            but it is here to adhere to the MFG structure from the Ontological
            Analysis.
          */

         goto Sx;

      fi;


Sx:
      goto S1;


S6:
      /* Label S6 corresponds to state S6 in the driver's state machine
         of the Level 3 MFGs
         atomically, the following flags are set; this fulfills the predicates
         necessary for the state machine to be in state S6.
         - The flag inZ for this train and its current station is set
           to false
         - the flag indicating that the train is now between its so-called "current"
           station and the next station on its route, is set to true.

         This represents the actual jouney of the train between the stations.

         The state machine then tansits into state S8.
      */

      skip;
      d_step {
         inZ[train].station[Current[train]] = false;
         zw[train].previous[Current[train]].next[Next[train]] = true;
      } /* end d_step */


S8:
      /* Label S8 corresponds to state S8 in the driver's state machine
         of the Level 3 MFGs
         atomically, the following flags are set; this fulfills the predicates
         necessary for the state machine to be in state S6.
         - The flag inZ for this train and its "next" station is set
           to true
         - the flag indicating that the train is now between its so-called "current"
           station and the "next" station on its route, is set to false.

         This represents the arrival of the train in the new station.
```

```
      The state machine then tansits into state S9.
   */

   d_step {
      zw[train].previous[Current[train]].next[Next[train]] = false;
      inZ[train].station[Next[train]] = true;
   } /* end d_step */


S9:
   /* Label S9 corresponds to state S9 in the driver's state machine
      of the Level 3 MFGs
      In one deterministic step, compose the message to contain
      - type: AM
      - train ID: this train
      - destination: so-called "Next" station on this train's route.
      Then send the message to the controller.

      Note that this expression will block until the message is received
      by the controller.

      After the message is sent, the state machine transits into
      state S11.
    */

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

      /* Send the composed Ankunftmeldung-Message to controller */
      t2c[train]!ZLM;
   } /* end atomic */


   /* The state machine of one driver has reached its final state
      S11 is the accepting state in the MFG, indicating that a single
      leg of the journey from one station to the next has been completed.
      Here we model a journey spanning several stations, so the variables
      have to be re-set to reflect that the journey to the next
      station will begin. */

S11:

   if
   :: Current[train] == destination ->

      /* if the train has reached its destination,
         clear the inZ flag to indicate that the train has been parked
         and is no longer blocking that station,
         end the main do loop, end the process.
       */

      inZ[train].station[Current[train]] = false;
      break;

   :: else ->

      /* As long as the current station of the train is not its final
         destination, do the following ...
```

```
            do in one deterministic step:
            - Set the next station variable for this train to the next
              station on the train's route stored in the Route array
            - Set the inZ flag for this train and its current station to TRUE.
         */

         d_step {
            Next[train] = Route[train].station[Routeindex[train] + 1];
            inZ[train].station[Current[train]] = true;
         } /* end d_step */
      fi;

   od;

} /* end proctype "driver" */


inline Check_KH(tr, from_station, to_station)
{

   /* an inline "function" to determine if there is no obstruction
      ("kein Hindernis", KH) for the journey of a train from one
      station to the next. This is both a somewhat reduced version of
      the check that has to be performed in reality, which might include
      a check of the state of the tracks, people on the tracks, etc.),
      and a concretion of the abstract concept of KH in the MFG.

      In this implementation, KH for a train to go from its current
      to its next station is false if:
      for any other train T2:
        \/ T2 occupies segment from current to next station
        \/ T2 occupies segment from next to current station
        \/  /\ T2 is in this train's next station
            /\ T2's next station is _not_ this train's current station

      The third point is based on the assumption that all stations have
      spring switches. that guarantee that trains entering a station in
      different directions always end up on different tracks. So a station
      can only be entered if only one other is already in it, and that other
      train is travelling in the opposite direction. Trains do not reverse
      directions in this model.

      This inline checks for a train tr, if there are obstructions
      known that would prevent it from occupying the segment from
      from_station to to_station under central responsibility.

      this loop is executed deterministically, meaning that
      always the first executable guard is passed.

    */

   d_step {

      /* First, set the KH flag for this tr journey to true */
      KH[tr].previous[from_station].next[to_station] = true;

      /* initialise the counter to enumerate all other trains to 0 */
      counter = 0;

      /* begin the main loop over all other trains */
      do

      :: (counter == TRAINS) ->
         /* if the counter reaches the total number of trains, terminate the loop:
```

```
            all trains have been checked
          */
         break;

      :: (counter == tr) ->
         /* if the counter is the same as this train's number, skip the checks
          */
         counter++;

      :: else ->
         /* otherwise, check ...
          */

         if
         :: ( ZV[counter].previous[from_station].next[to_station] ) ->

            /* The other train with ID "counter" is occupying the segment
               from "from_station" to "to_station" under central resonsibility
             */
            KH[tr].previous[from_station].next[to_station] = false;
            break;

         :: ( ZV[counter].previous[to_station].next[from_station] ) ->

            /* The other train with ID "counter" is occupying the segment
               from "to_station" to "from_station" (i. e. the same segment,
               but in reverse direction)
             */
            KH[tr].previous[from_station].next[to_station] = false;
            break;

         :: ( ( inZ[counter].station[to_station] ) && ( Next[counter] != from_station ) ) ->

            /* the other train with ID "counter" is in the "to_station"
               _and_ is travelling in the same direction as this train.
               (i. e. the other train's destination is _not_ "from_station")
             */
            KH[tr].previous[from_station].next[to_station] = false;
            break;

         :: else ->
            /* otherwise do nothing
             */
            skip;

         fi;

         /* increase the loop counter, enter the next iteration */
         counter++;

      od;
   } /* end d_step */
} /* end inline KH_Check */


/* Process specification for the dispatcher/controller;
   an individual process is created for each train */

proctype controller(byte train, start, destination) {

   /* initialisation */

   Zuglaufmeldung ZLM;
   byte previous, next;
```

```
   Current[train] = start;

   /* begin main loop through state machine */

   do

   :: Current[train] == destination ->
      /* the train journey of this train is complete, the train has been
         removed from the virtual track (ZV flag has been cleared).
         The job is done, exit from the main loop.
       */
      break;


   :: else ->

     /* if the train has not yet reached its destination, do the following ... */

S0:

     /* Label S0 corresponds to the state S0 in the controller state machine
        in the MFG
        upon reception of the appropriate message (FA) it transits to state
        (Label) S2. If any other message is received, remain in state S0.

        Note that since synchronous communications are used, the process blocks
        in the recepction expression ("?") if no message is received.
     */

      /* wait for reception of message from train */
      t2c[train]?ZLM;
      if
         /* upon message type: FA ... */
      :: ZLM.type == tFA ->

         /* - make the transition to state S2
          */
         goto S2;

      :: else ->
         /* No other message can be received here */
         assert(false);
      fi;


S2:
      /* State S2 corresponds to the state S2 in the controller's state machine
         in the MFG.
         KH is determined using the inline "function", depending on the
         outcome:
         - if KH for this train and the segment between the current position
           and the "Next" station is true: compose the FE message and send
           it to the driver process; transit to Label S4
         - if KH is false, compose the AFE message and send it to the driver;
           transit to Label (state) S5
      */

      /* This needs to be done atomically, to avoid a race condition where another
         process can occupy a segment under central control.
      */
      atomic {
         /* Check for any obstruction (see inline "Function" Check_KH */
         Check_KH(train, Current[train], Next[train]);
         ZV[train].previous[Current[train]].next[Next[train]] =
            KH[train].previous[Current[train]].next[Next[train]];
```

```
      } /* end atomic */


      if
      :: ( ZV[train].previous[Current[train]].next[Next[train]] ) ->

         /* the flag indicating that this train occupies the
            segment between the current position and the train's
            route's next station is set, so
            compose the FE message and send it to the driver;
            then transition to state S4 */

         d_step {
            ZLM.type = tFE;
            ZLM.train = train;
            ZLM.destination = Next[train];
         }

         /* Send the composed Fahrerlaubnis-Message to Train */

         c2t[train]!ZLM;
         goto S4;

      :: else ->
         /* if an obstruction is known, and the ZV flag has not been set,
            compose and send the AFE message;
            then transition to state S5 */
         d_step {
            ZLM.type = tAFE;
            ZLM.train = train;
            ZLM.destination = Next[train];
         }

         /* Send the composed Ablehnung der Fahrerlaubnis-Message to train */
         c2t[train]!ZLM;
         goto S5;
      fi;


S4:
      /* Label S4 corresponds to the state S4 in the controller state machine
         in the MFG.
         FE has been sent (and received)
         Wait for train to finish the segment, and send the AM.
         Upon recepction of the Ankunftmeldung (message type AM) from
         the driver, clear the flag indicating occupation of the
         segment under central responsibility. Then transit to state
         S12.
       */

      /* wait for message "AM" from train
         Note that since synchronous communications are used, this expression
         blocks until a message is received.
       */
      atomic {
         previous = Current[train];
         next = Next[train];
      }

      atomic {
         t2c[train]?ZLM;

         if
         /* message type is AM ... */
         :: ( ZLM.type == tAM ) ->
```

```
            /* clear the "central responsibility" flag;
               transition to state S12 */
            ZV[train].previous[previous].next[next] = false;

            Routeindex[train]++;
            Current[train]=Route[train].station[Routeindex[train]];
            goto S12;
         :: else ->
         /* No other type of message can be received here */
             assert(false);
         fi;
      }


S5:

      /* According to regulations, after the denial of clearance message
         has been given due to obstructions, clearance shall be given as soon
         as the obstruction disappears.
         So KH is checked in a loop until it is true, then the FE
         message is composed and sent, the machine transits to state S4.
         This corresponds to the state S5 in the controller's state machine
         in the MFG.
      */

      /* loop, checking KH, and sending FE if it becomes true */

      atomic {
         /* This atomic sequence will never block ... */

         /* Check for absence of obstructions */
         Check_KH(train, Current[train], Next[train]);

         /* set the flag indicating occupation of the segment under
            central responsibility to the truth value of the KH check */
         ZV[train].previous[Current[train]].next[Next[train]] =
            KH[train].previous[Current[train]].next[Next[train]]
      } /* end atomic */

      if
      :: ( ZV[train].previous[Current[train]].next[Next[train]] ) ->

         /* If segment is "reserved" under central responsibility, send
            the FE message, then transition to state S4 */
         d_step {
            ZLM.type = tFE;
            ZLM.train = train;
            ZLM.destination = Next[train];
         };
         c2t[train]!ZLM;
         goto S4;

      :: else ->
         /* otherwise: stay in S5 */
         goto S5;
      fi;


S12:
      /* The segment has been traversed successfully, state S12 in the MFG is
         accepting state of the state machine.

         This must not be confused with the valid end state of the promela
         process. This is only reached after the train has reached its
         _final_ destination.
```

```
      */
   skip;
   od; /* end loop over inddividual segments of complete train journey */
}
```

```
/*
   Model of modified Zugleitbetrieb for SPIN
t/


/* These can be modified for individual runs of the checker. */

#define TRAINS 4
#define STATIONS 5

/* Array of bools of the size of the number of stations.
   Used in multi-dimensional arrays to store flags per train
   and per station. */

typedef boolstationtype {
   bool station[STATIONS];
}


/* array of bool of size of the number of stations,
   used as a type in various multi-dimensional arrays. */

typedef boolbetweentype {
   bool next[STATIONS];
}


/* two-dimensional bool array for a matrix of stations */

typedef bool2betweentype {
   boolbetweentype previous[STATIONS];
}


/* Arrays defining for each train its start station
   and the direction it is traveling */

byte start_station[TRAINS];
byte last_destination[TRAINS];
short direction[TRAINS];


/* speaking names for train directions */

#define forward (1)
#define backward (-1)

/* two-dimensional array to indicate whether a train is in a station
   and whether for a train the Ankunftmeldung in a specific station
```

```
   has been transmitted and received. */

boolstationtype inZ[TRAINS] = false;


/* three-dimensional arrays to indicate that for a track segment between
   stations the corresponding message has been sent and received
   (FA, FE, AFE), the train actually is between the stations (zw),
   no obstructions are known (KH),
   the segment is occupied under central responsibility (ZV) */

bool2betweentype zw[TRAINS] = false,
                 KH[TRAINS] = false,
                 ZV[TRAINS] = false;

/* arrays to indicate for each train:
   the next station on its trip,
   the currently occupied or recently-left station,
   the index pointing to the current station in its Route array */

byte Next[TRAINS],
     Current[TRAINS];


/* symbolic types for message types */

/* A message can be of type
   FA - Fahranfrage
   FE - Fahrerlaubnis
   AFE - Ablehnung Fahrerlaubnis
   AM - Ankunftmeldung */

mtype = {tFA, tFE, tAFE, tAM};


/* A Zuglaufmeldung is a triple, consisting of
   message type, train ID, and train destination.
   This is modelled after the original ZLM in
   the FV-NE standard. */

typedef Zuglaufmeldung {
    mtype type;
    byte train;
    byte destination;
}


/* Communication channels:
   two for each train: bidirectional, synchronous (buffer length 0)
```

```
   c2t: controller to train
   t2c: train to controller
*/

chan c2t[TRAINS] = [0] of { Zuglaufmeldung };
chan t2c[TRAINS] = [0] of { Zuglaufmeldung };


/* A general purpose counter for enumerating trains */
byte counter;


init {

   byte train_counter;

   /* Always set up Train 0 in Station 2 (middle of the route)
      Set up the other trains in one of several explicit situations
    */

   start_station[0] = 2;

   /* To be shown elsewhere: That these situations are all possible
      situations for four trains.

      Trains always run to the end of the track (station #4 in forward
      direction or station 0 in backward direction)
      Also set up possible permutations of train running directions;
    */

#include "Situations/Current-Situation.prom"


   /*********************
     Start the processes
    *********************/

   /* for every train start a driver and a controller process */

   train_counter = 0;
   atomic {
      do

      /* if the counter has reached the total number of trains, exit the loop */
      :: (train_counter == TRAINS) -> break;

      /* otherwise ... */
      :: else ->

```

```
148:          /* Set all destinations, Current and Next variables to appropriate values
149:             to avoid race conditions */
150:
151:          if
152:          :: direction[train_counter] == forward -> last_destination[train_counter] = STATIONS - 1;
153:          :: direction[train_counter] == backward -> last_destination[train_counter] = 0;
154:          fi;
155:          Current[train_counter] = start_station[train_counter];
156:          Next[train_counter] = start_station[train_counter] + direction[train_counter];
157:
158:          /* start the controller process for this train, with this train's route start and end */
159:          printf("Starting Controller process for Train %d, going from %d to %d.\n",
160:                  train_counter, start_station[train_counter], last_destination[train_counter]);
161:          run controller(train_counter, start_station[train_counter], last_destination[train_counter]);
162:
163:          /* start the driver process for this train, with this train's route start and end */
164:          printf("Starting Driver process for Train %d, going from %d to %d.\n",
165:                  train_counter, start_station[train_counter], last_destination[train_counter]);
166:          run driver(train_counter, start_station[train_counter], last_destination[train_counter]);
167:
168:          /* increment the counter and enter the next iteration */
169:          train_counter++;
170:       od;
171:    } /* end atomic */
172:
173: }
174:
175:
176: proctype driver(byte train, start, destination)
177: {
178:
179:    /* initialisation */
180:
181:    Zuglaufmeldung ZLM;
182:
183:    /* start the main loop for the journey of one train from its
184:       start to its destination station. */
185:
186:    do
187:
188:    :: true ->
189:
190:
191: S0:
192:       /* Label S0 corresponds to state S0 in the driver's state machine
193:          of the Level 3 MFGs
194:          - Variables are set to indicate the train's next station
195:          - Flag is set to indicate that train is in its current station.
196:
```

```
197:          In one deterministic step set the compononents of the message triple
198:          to the message type (FA), the train ID to the current train,
199:          and the destination to this train's current "Next" station.
200:
201:          Send the message through the appropriate channel.
202:
203:          This is the specification of the transition from S0 to S1,
204:          which follows unconditionally after sending the message.
205:
206:          Note that since we use synchronous communications, this
207:          driver process will block until the controller has received
208:          the message.
209:       */
210:
211:       atomic {
212:          d_step {
213:             ZLM.type = tFA;
214:             ZLM.train = train;
215:             ZLM.destination = Next[train];
216:          } /* end d_step */
217:
218:          /* send the compsed Fahranfrage-Message to the controller */
219:          t2c[train]!ZLM;
220:       } /* end atomic */
221:
222:
223: S1:
224:       /* Label S1 corresponds to state S1 in the driver's state machine
225:          of the Level 3 MFGs
226:          - FA is true whenever the FA-message has been sent and received. Since this
227:            process is unblocked only when the message has been received, this is now true.
228:          - Upon reception of a message from the controller, there will be a transition either
229:            . into S6, if the message type was FE
230:            . into Sx, if the message type was AFE
231:       */
232:
233:
234:       /* wait for message (blocks when using asynchronous communications) */
235:
236:       c2t[train]?ZLM;
237:
238:       if
239:
240:       :: ZLM.type == tFE ->
241:
242:          /* if message type was FE, then goto Label S6 */
243:
244:          goto S6;
245:
```

```
246:       :: ZLM.type == tAFE ->
247:
248:          /* if the message type is AFE, then go to Label Sx.
249:             Since Sx just goes back to S1, this effectively does nothing,
250:             but it is here to adhere to the MFG structure from the Ontological
251:             Analysis.
252:           */
253:
254:          goto Sx;
255:
256:       fi;
257:
258:
259: Sx:
260:       goto S1;
261:
262:
263: S6:
264:       /* Label S6 corresponds to state S6 in the driver's state machine
265:          of the Level 3 MFGs
266:          atomically, the following flags are set; this fulfills the predicates
267:          necessary for the state machine to be in state S6.
268:          - The flag inZ for this train and its current station is set
269:            to false
270:          - the flag indicating that the train is now between its so-called "current"
271:            station and the next station on its route, is set to true.
272:
273:          This represents the actual jouney of the train between the stations.
274:
275:          The state machine then tansits into state S8.
276:       */
277:
278:       skip;
279:       d_step {
280:          inZ[train].station[Current[train]] = false;
281:          zw[train].previous[Current[train]].next[Next[train]] = true;
282:       } /* end d_step */
283:
284:
285: S8:
286:       /* Label S8 corresponds to state S8 in the driver's state machine
287:          of the Level 3 MFGs
288:          atomically, the following flags are set; this fulfills the predicates
289:          necessary for the state machine to be in state S6.
290:          - The flag inZ for this train and its "next" station is set
291:            to true
292:          - the flag indicating that the train is now between its so-called "current"
293:            station and the "next" station on its route, is set to false.
294:
```

```
295:          This represents the arrival of the train in the new station.
296:
297:          The state machine then tansits into state S9.
298:       */
299:
300:       d_step {
301:          zw[train].previous[Current[train]].next[Next[train]] = false;
302:          inZ[train].station[Next[train]] = true;
303:       } /* end d_step */
304:
305:
306: S9:
307:       /* Label S9 corresponds to state S9 in the driver's state machine
308:          of the Level 3 MFGs
309:          In one deterministic step, compose the message to contain
310:          - type: AM
311:          - train ID: this train
312:          - destination: so-called "Next" station on this train's route.
313:          Then send the message to the controller.
314:
315:          Note that this expression will block until the message is received
316:          by the controller.
317:
318:          After the message is sent, the state machine transits into
319:          state S11.
320:        */
321:
322:       atomic
323:       {
324:          d_step
325:          {
326:             ZLM.type = tAM;
327:             ZLM.train = train;
328:             ZLM.destination = Next[train];
329:          } /* end d_step */
330:
331:          /* Send the composed Ankunftmeldung-Message to controller */
332:          t2c[train]!ZLM;
333:       } /* end atomic */
334:
335:
336:       /* The state machine of one driver has reached its final state
337:          S11 is the accepting state in the MFG, indicating that a single
338:          leg of the journey from one station to the next has been completed.
339:          Here we model a journey spanning several stations, so the variables
340:          have to be re-set to reflect that the journey to the next
341:          station will begin. */
342:
343: S11:
```

```
344:
345:       if
346:       :: Current[train] == destination ->
347:
348:          /* if the train has reached its destination,
349:             clear the inZ flag to indicate that the train has been parked
350:             and is no longer blocking that station,
351:             end the main do loop, end the process.
352:           */
353:
354:          inZ[train].station[Current[train]] = false;
355:          break;
356:
357:       :: else ->
358:
359:          /* As long as the current station of the train is not its final
360:             destination, do the following ...
361:
362:             do in one deterministic step:
363:             - Set the next station variable for this train to the next
364:               station on the train's route stored in the Route array
365:             - Set the inZ flag for this train and its current station to TRUE.
366:          */
367:
368:          d_step {
369:             Next[train] = Current[train] + direction[train];
370:             inZ[train].station[Current[train]] = true;
371:          } /* end d_step */
372:       fi;
373:
374:    od;
375:
376: } /* end proctype "driver" */
377:
378:
379: inline Check_KH(tr, from_station, to_station)
380: {
381:
382:    /* an inline "function" to determine if there is no obstruction
383:       ("kein Hindernis", KH) for the journey of a train from one
384:       station to the next. This is both a somewhat reduced version of
385:       the check that has to be performed in reality, which might include
386:       a check of the state of the tracks, people on the tracks, etc.),
387:       and a concretion of the abstract concept of KH in the MFG.
388:
389:       In this implementation, KH for a train to go from its current
390:       to its next station is false if:
391:       for any other train T2:
392:         \/ T2 occupies segment from current to next station
```

```
393:         \/ T2 occupies segment from next to current station
394:         \/  /\ T2 is in this train's next station
395:             /\ T2's next station is _not_ this train's current station
396:
397:       The third point is based on the assumption that all stations have
398:       spring switches. that guarantee that trains entering a station in
399:       different directions always end up on different tracks. So a station
400:       can only be entered if only one other is already in it, and that other
401:       train is travelling in the opposite direction. Trains do not reverse
402:       directions in this model.
403:
404:       This inline checks for a train tr, if there are obstructions
405:       known that would prevent it from occupying the segment from
406:       from_station to to_station under central responsibility.
407:
408:       this loop is executed deterministically, meaning that
409:       always the first executable guard is passed.
410:
411:     */
412:
413:    d_step {
414:
415:       /* First, set the KH flag for this tr journey to true */
416:       KH[tr].previous[from_station].next[to_station] = true;
417:
418:       /* initialise the counter to enumerate all other trains to 0 */
419:       counter = 0;
420:
421:       /* begin the main loop over all other trains */
422:       do
423:
424:       :: (counter == TRAINS) ->
425:          /* if the counter reaches the total number of trains, terminate the loop:
426:             all trains have been checked
427:           */
428:          break;
429:
430:       :: (counter == tr) ->
431:          /* if the counter is the same as this train's number, skip the checks
432:           */
433:          counter++;
434:
435:       :: else ->
436:          /* otherwise, check ...
437:           */
438:
439:          if
440:          :: ( ZV[counter].previous[from_station].next[to_station] ) ->
441:
```

```
442:             /* The other train with ID "counter" is occupying the segment
443:                from "from_station" to "to_station" under central resonsibility
444:              */
445:             KH[tr].previous[from_station].next[to_station] = false;
446:             break;
447:
448:          :: ( ZV[counter].previous[to_station].next[from_station] ) ->
449:
450:             /* The other train with ID "counter" is occupying the segment
451:                from "to_station" to "from_station" (i. e. the same segment,
452:                but in reverse direction)
453:              */
454:             KH[tr].previous[from_station].next[to_station] = false;
455:             break;
456:
457:          :: ( ( inZ[counter].station[to_station] ) && ( Next[counter] != from_station ) ) ->
458:
459:             /* the other train with ID "counter" is in the "to_station"
460:                _and_ is travelling in the same direction as this train.
461:                (i. e. the other train's destination is _not_ "from_station")
462:              */
463:             KH[tr].previous[from_station].next[to_station] = false;
464:             break;
465:
466:          :: else ->
467:             /* otherwise do nothing
468:              */
469:             skip;
470:
471:          fi;
472:
473:          /* increase the loop counter, enter the next iteration */
474:          counter++;
475:
476:       od;
477:    } /* end d_step */
478: } /* end inline KH_Check */
479:
480:
481: /* Process specification for the dispatcher/controller;
482:    an individual process is created for each train */
483:
484: proctype controller(byte train, start, destination) {
485:
486:    /* initialisation */
487:
488:    Zuglaufmeldung ZLM;
489:    byte previous, next;
490:
```

```
491:    Current[train] = start;
492:
493:    /* begin main loop through state machine */
494:
495:    do
496:
497:    :: Current[train] == destination ->
498:       /* the train journey of this train is complete, the train has been
499:          removed from the virtual track (ZV flag has been cleared).
500:          The job is done, exit from the main loop.
501:        */
502:       break;
503:
504:
505:    :: else ->
506:
507:      /* if the train has not yet reached its destination, do the following ... */
508:
509: S0:
510:
511:      /* Label S0 corresponds to the state S0 in the controller state machine
512:         in the MFG
513:         upon reception of the appropriate message (FA) it transits to state
514:         (Label) S2. If any other message is received, remain in state S0.
515:
516:         Note that since synchronous communications are used, the process blocks
517:         in the recepction expression ("?") if no message is received.
518:      */
519:
520:       /* wait for reception of message from train */
521:       t2c[train]?ZLM;
522:       if
523:          /* upon message type: FA ... */
524:       :: ZLM.type == tFA ->
525:
526:          /* - make the transition to state S2
527:           */
528:          goto S2;
529:
530:       :: else ->
531:          /* No other message can be received here */
532:          assert(false);
533:       fi;
534:
535:
536: S2:
537:       /* State S2 corresponds to the state S2 in the controller's state machine
538:          in the MFG.
539:          KH is determined using the inline "function", depending on the
```

```
540:          outcome:
541:          - if KH for this train and the segment between the current position
542:            and the "Next" station is true: compose the FE message and send
543:            it to the driver process; transit to Label S4
544:          - if KH is false, compose the AFE message and send it to the driver;
545:            transit to Label (state) S5
546:       */
547:
548:       /* This needs to be done atomically, to avoid a race condition where another
549:          process can occupy a segment under central control.
550:       */
551:       atomic {
552:          /* Check for any obstruction (see inline "Function" Check_KH */
553:          Check_KH(train, Current[train], Next[train]);
554:          ZV[train].previous[Current[train]].next[Next[train]] =
555:             KH[train].previous[Current[train]].next[Next[train]];
556:       } /* end atomic */
557:
558:
559:       if
560:       :: ( ZV[train].previous[Current[train]].next[Next[train]] ) ->
561:
562:          /* the flag indicating that this train occupies the
563:             segment between the current position and the train's
564:             route's next station is set, so
565:             compose the FE message and send it to the driver;
566:             then transition to state S4 */
567:
568:          d_step {
569:             ZLM.type = tFE;
570:             ZLM.train = train;
571:             ZLM.destination = Next[train];
572:          }
573:
574:          /* Send the composed Fahrerlaubnis-Message to Train */
575:
576:          c2t[train]!ZLM;
577:          goto S4;
578:
579:       :: else ->
580:          /* if an obstruction is known, and the ZV flag has not been set,
581:             compose and send the AFE message;
582:             then transition to state S5 */
583:          d_step {
584:             ZLM.type = tAFE;
585:             ZLM.train = train;
586:             ZLM.destination = Next[train];
587:          }
588:
```

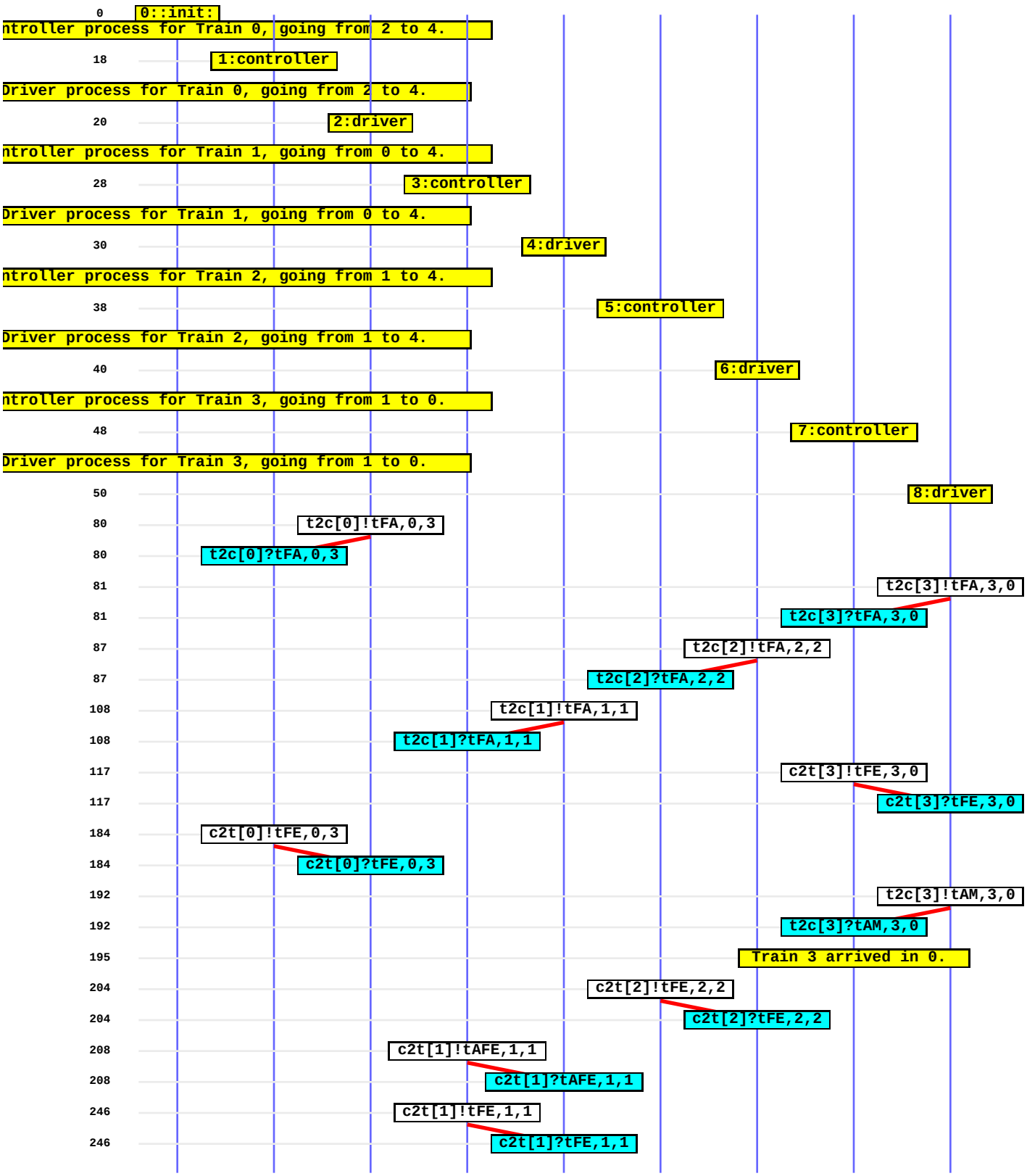
```
589:          /* Send the composed Ablehnung der Fahrerlaubnis-Message to train */
590:          c2t[train]!ZLM;
591:          goto S5;
592:       fi;
593:
594:
595: S4:
596:       /* Label S4 corresponds to the state S4 in the controller state machine
597:          in the MFG.
598:          FE has been sent (and received)
599:          Wait for train to finish the segment, and send the AM.
600:          Upon recepction of the Ankunftmeldung (message type AM) from
601:          the driver, clear the flag indicating occupation of the
602:          segment under central responsibility. Then transit to state
603:          S12.
604:        */
605:
606:       /* wait for message "AM" from train
607:          Note that since synchronous communications are used, this expression
608:          blocks until a message is received.
609:        */
610:       atomic {
611:          previous = Current[train];
612:          next = Next[train];
613:       }
614:
615:       atomic {
616:          t2c[train]?ZLM;
617:
618:          if
619:          /* message type is AM ... */
620:          :: ( ZLM.type == tAM ) ->
621:             /* clear the "central responsibility" flag;
622:                transition to state S12 */
623:             ZV[train].previous[previous].next[next] = false;
624:             printf("Train %d arrived in %d.\n", train, next);
625:
626:             Current[train] = Current[train] + direction[train];
627:             goto S12;
628:          :: else ->
629:          /* No other type of message can be received here */
630:              assert(false);
631:          fi;
632:       }
633:
634:
635: S5:
636:
637:       /* According to regulations, after the denial of clearance message
```

```
638:          has been given due to obstructions, clearance shall be given as soon
639:          as the obstruction disappears.
640:          So KH is checked in a loop until it is true, then the FE
641:          message is composed and sent, the machine transits to state S4.
642:          This corresponds to the state S5 in the controller's state machine
643:          in the MFG.
644:       */
645:
646:       /* loop, checking KH, and sending FE if it becomes true */
647:
648:       atomic {
649:          /* This atomic sequence will never block ... */
650:
651:          /* Check for absence of obstructions */
652:          Check_KH(train, Current[train], Next[train]);
653:
654:          /* set the flag indicating occupation of the segment under
655:             central responsibility to the truth value of the KH check */
656:          ZV[train].previous[Current[train]].next[Next[train]] =
657:             KH[train].previous[Current[train]].next[Next[train]]
658:       } /* end atomic */
659:
660:       if
661:       :: ( ZV[train].previous[Current[train]].next[Next[train]] ) ->
662:
663:          /* If segment is "reserved" under central responsibility, send
664:             the FE message, then transition to state S4 */
665:          d_step {
666:             ZLM.type = tFE;
667:             ZLM.train = train;
668:             ZLM.destination = Next[train];
669:          };
670:          c2t[train]!ZLM;
671:          goto S4;
672:
673:       :: else ->
674:          /* otherwise: stay in S5 */
675:          goto S5;
676:       fi;
677:
678:
679: S12:
680:       /* The segment has been traversed successfully, state S12 in the MFG is
681:          accepting state of the state machine.
682:
683:          This must not be confused with the valid end state of the promela
684:          process. This is only reached after the train has reached its
685:          _final_ destination.
686:       */
```

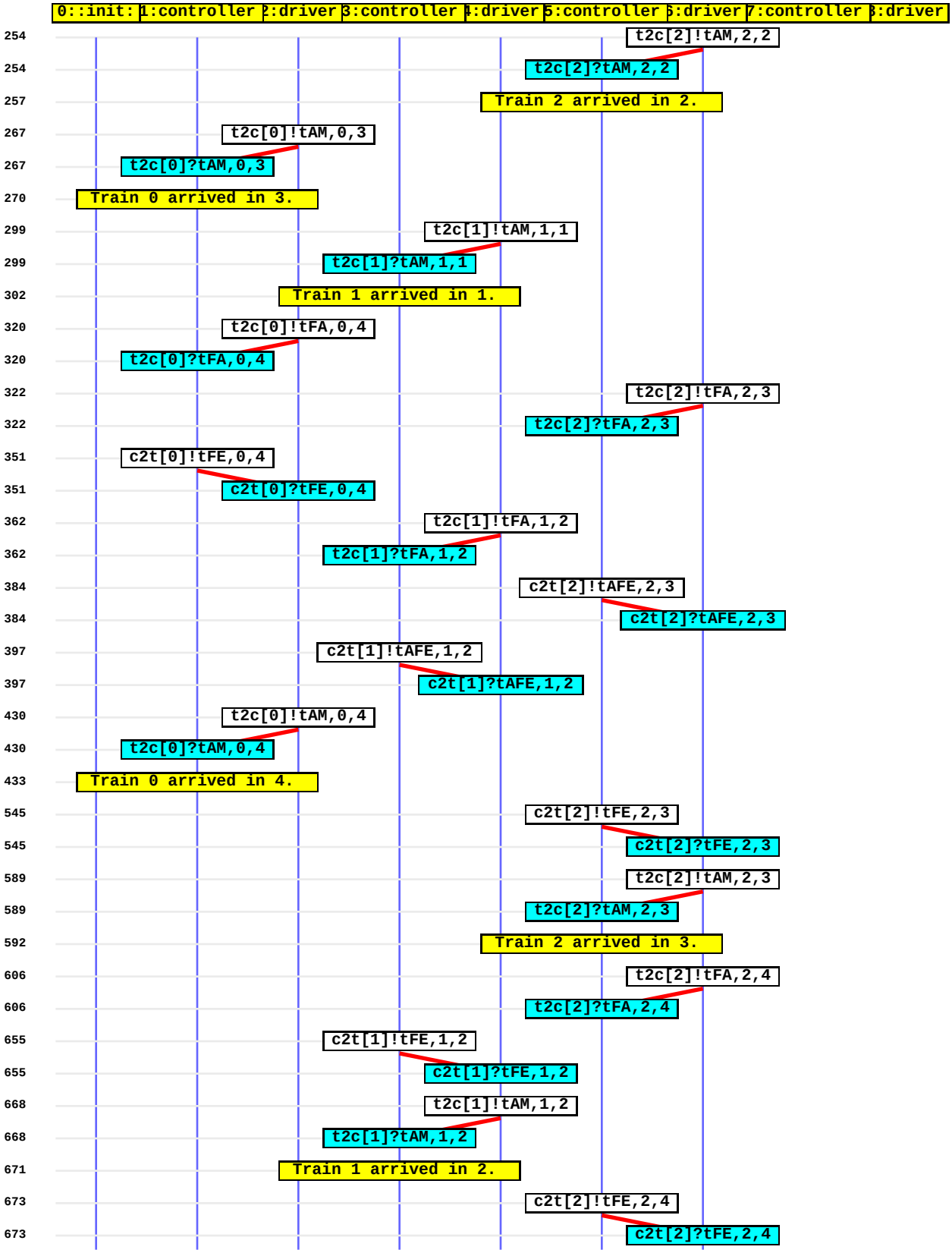
```
687:
688:    skip;
689:
690:    od; /* end loop over inddividual segments of complete train journey */
691:
692: }
```

Anhang B.2: Message-Sequence-Chart-Ausgabe eines SPIN-Simulationslaufs

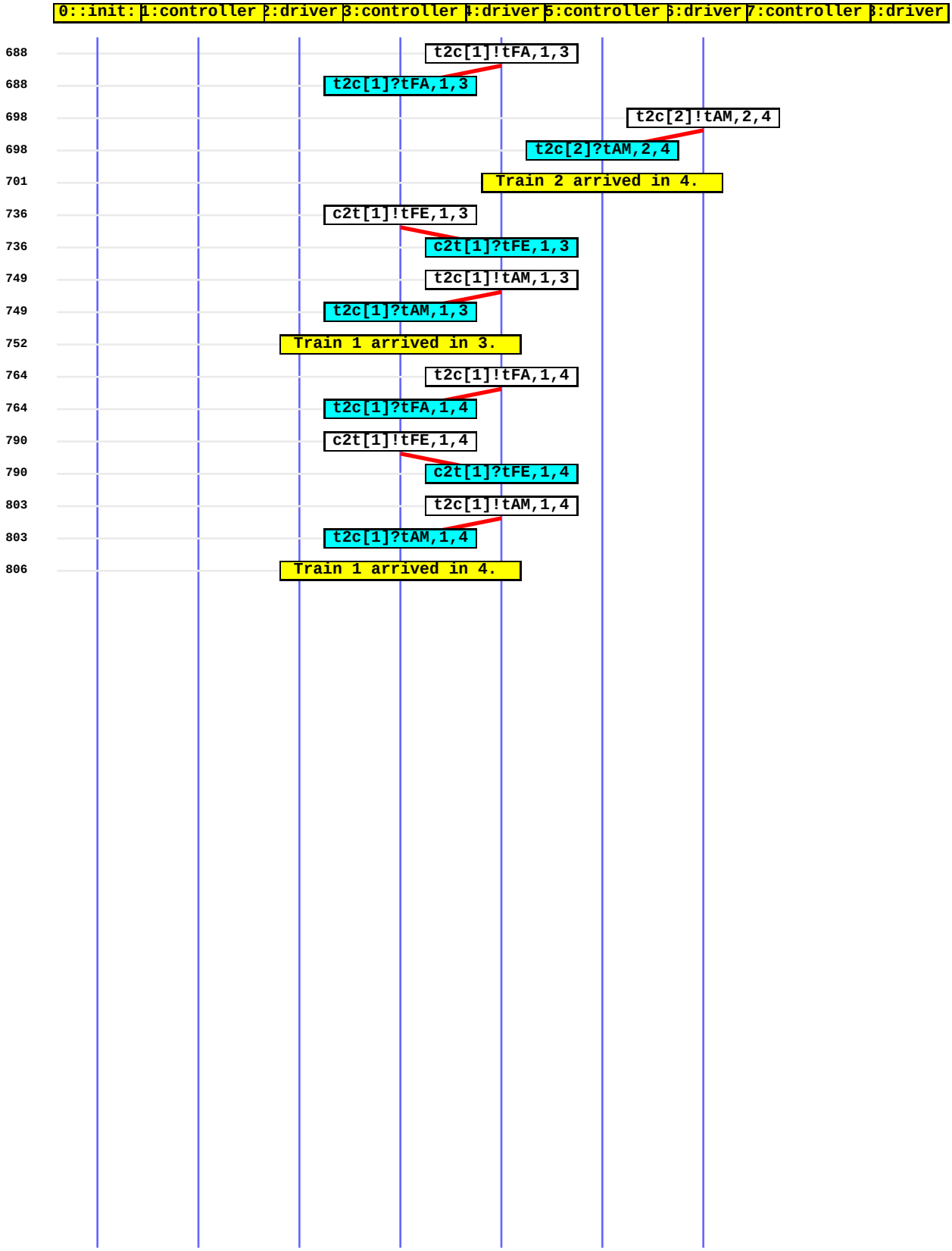
Spin Version 5.1.6 -- 9 May 2008 -- Four-Train-Situations.prom -- MSC -- 1



Spin Version 5.1.6 -- 9 May 2008 -- Four-Train-Situations.prom -- MSC -- 2



Spin Version 5.1.6 -- 9 May 2008 -- Four-Train-Situations.prom -- MSC -- 3



Anhang B.2: SPIN-Ausgabe der PROMELA-Statemachines

```
    1: proctype init
    2:         state   1 -(tr  33)-> state   2  [id   0 tp   2] [----G] line 119 => start_station[0] = 2
    3:         state   2 -(tr  34)-> state   3  [id   1 tp   2] [----G] line 5 => start_station[1] = 1
    4:         state   3 -(tr  35)-> state   4  [id   2 tp   2] [----G] line 6 => start_station[2] = 2
    5:         state   4 -(tr  36)-> state   7  [id   3 tp   2] [----G] line 7 => start_station[3] = 3
    6:         state   7 -(tr  37)-> state   9  [id   4 tp   2] [----G] line 9 => direction[1] = 1
    7:         state   7 -(tr  38)-> state   9  [id   5 tp   2] [----G] line 9 => direction[1] = -(1)
    8:         state   9 -(tr  39)-> state  10  [id   8 tp   2] [----G] line 14 => direction[0] = 1
    9:         state  10 -(tr  40)-> state  13  [id   9 tp   2] [----G] line 15 => direction[2] = -(1)
   10:         state  13 -(tr  41)-> state  15  [id  10 tp   2] [----G] line 17 => direction[3] = 1
   11:         state  13 -(tr  42)-> state  15  [id  11 tp   2] [----G] line 17 => direction[3] = -(1)
   12:         state  15 -(tr  43)-> state  35  [id  14 tp   2] [----L] line 138 => train_counter = 0
   13:         state  35 -(tr  44)-> state  34  [id  15 tp   2] [A---G] line 139 => ((train_counter==4))
   14:         state  35 -(tr   2)-> state  23  [id  17 tp   2] [A---G] line 139 => else
   15:         state  34 -(tr   1)-> state  36  [id  33 tp   2] [----G] line 140 => break
   16:         state  36 -(tr  50)-> state   0  [id  35 tp 3500] [--e-L] line 173 => -end-     [(257,9)]
   17:         state  23 -(tr  45)-> state  28  [id  18 tp   2] [A---G] line 151 => ((direction[train_counter]==1))
   18:         state  23 -(tr  46)-> state  28  [id  20 tp   2] [A---G] line 151 => ((direction[train_counter]==-(1)))
   19:         state  28 -(tr  48)-> state  30  [id  27 tp   2] [A---G] line 161 => (run controller(train_counter,start_station[train_counter],last_destination[tra
in_counter]))
   20:         state  30 -(tr  49)-> state  32  [id  29 tp   2] [A---G] line 166 => (run driver(train_counter,start_station[train_counter],last_destination[train_c
ounter]))
   21:         state  32 -(tr  44)-> state  34  [id  15 tp   2] [A---G] line 140 => ((train_counter==4))
   22:         state  32 -(tr   2)-> state  23  [id  17 tp   2] [A---G] line 140 => else
   23: state 23 line 151 is a loopstate
   24: proctype driver
   25:         state  38 -(tr   1)-> state   7  [id  36 tp   2] [----L] line 186 => (1)
   26:         state   7 -(tr  21)-> state   6  [id  40 tp   2] [A---G] line 211 => D_STEP
   27:         state   6 -(tr  22)-> state   8  [id  41 tp   4] [----G] line 219 => t2c[train]!ZLM.type,ZLM.train,ZLM.destination      [(2,2)]
   28:         state   8 -(tr  23)-> state  13  [id  43 tp 503] [----G] line 236 => c2t[train]?ZLM.type,ZLM.train,ZLM.destination      [(1,3)]
   29:         state  13 -(tr  24)-> state  16  [id  44 tp   2] [----L] line 238 => ((ZLM.type==tFE))
   30:         state  13 -(tr  25)-> state   8  [id  46 tp   2] [----L] line 238 => ((ZLM.type==tAFE))
   31:         state  16 -(tr   1)-> state  19  [id  51 tp   2] [----L] line 278 => (1)
   32:         state  19 -(tr  26)-> state  22  [id  54 tp   2] [D---G] line 279 => D_STEP
   33:         state  22 -(tr  27)-> state  28  [id  57 tp   2] [D---G] line 300 => D_STEP
   34:         state  28 -(tr  28)-> state  27  [id  61 tp   2] [A---G] line 322 => D_STEP
   35:         state  27 -(tr  22)-> state  36  [id  62 tp   4] [----G] line 332 => t2c[train]!ZLM.type,ZLM.train,ZLM.destination      [(2,2)]
   36:         state  36 -(tr  29)-> state  30  [id  64 tp   2] [----G] line 345 => ((Current[train]==destination))
   37:         state  36 -(tr   2)-> state  35  [id  67 tp   2] [----G] line 345 => else
   38:         state  30 -(tr  30)-> state  41  [id  65 tp   2] [----G] line 354 => inZ[train].station[Current[train]] = 0
   39:         state  41 -(tr  32)-> state   0  [id  76 tp 3500] [--e-L] line 376 => -end-     [(257,9)]
   40:         state  35 -(tr  31)-> state  38  [id  70 tp   2] [D---G] line 368 => D_STEP
   41: state 38 line 186 is a loopstate
   42: state 8 line 236 is a loopstate
   43: proctype controller
   44:         state   1 -(tr   3)-> state 110  [id  77 tp   2] [----G] line 491 => Current[train] = start
   45:         state 110 -(tr   4)-> state 113  [id  78 tp   2] [----G] line 495 => ((Current[train]==destination))
   46:         state 110 -(tr   2)-> state   5  [id  80 tp   2] [----G] line 495 => else
   47:         state 113 -(tr  20)-> state   0  [id 189 tp 3500] [--e-L] line 692 => -end-     [(257,9)]
   48:         state   5 -(tr   5)-> state  10  [id  81 tp 504] [----G] line 521 => t2c[train]?ZLM.type,ZLM.train,ZLM.destination      [(2,3)]
   49:         state  10 -(tr   6)-> state  39  [id  82 tp   2] [----L] line 522 => ((ZLM.type==tFA))
   50:         state  10 -(tr   2)-> state   9  [id  84 tp   2] [----L] line 522 => else
   51:         state  39 -(tr   8)-> state  38  [id 112 tp   2] [A---G] line 551 => D_STEP
   52:         state  38 -(tr   9)-> state  54  [id 114 tp   2] [----G] line 554 => ZV[train].previous[Current[train]].next[Next[train]] = KH[train].previous[Curre
nt[train]].next[Next[train]]
   53:         state  54 -(tr  10)-> state  44  [id 116 tp   2] [----G] line 559 => (ZV[train].previous[Current[train]].next[Next[train]])
   54:         state  54 -(tr   2)-> state  51  [id 123 tp   2] [----G] line 559 => else
   55:         state  44 -(tr  11)-> state  45  [id 120 tp   2] [D---G] line 568 => D_STEP
```

```
   56:         state  45 -(tr  12)-> state  58  [id 121 tp   3] [----G] line 576 => c2t[train]!ZLM.type,ZLM.train,ZLM.destination      [(1,2)]
   57:         state  58 -(tr  14)-> state  69  [id 132 tp   2] [A---G] line 610 => previous = Current[train]
   58:         state  69 -(tr   5)-> state  67  [id 135 tp 504] [A---G] line 615 => t2c[train]?ZLM.type,ZLM.train,ZLM.destination      [(2,3)]
   59:         state  67 -(tr  15)-> state 109  [id 136 tp 504] [A---G] line 618 => ((ZLM.type==tAM))  [(2,3)]
   60:         state  67 -(tr   2)-> state  66  [id 141 tp 504] [A---G] line 618 => else
   61:         state 109 -(tr   1)-> state 110  [id 185 tp   2] [----L] line 688 => (1)
   62:         state  66 -(tr  17)-> state  97  [id 142 tp 504] [A---G] line 630 => assert(0)  [(2,3)]
   63:         state  97 -(tr  18)-> state  96  [id 170 tp   2] [A---G] line 648 => D_STEP
   64:         state  96 -(tr   9)-> state 107  [id 172 tp   2] [----G] line 656 => ZV[train].previous[Current[train]].next[Next[train]] = KH[train].previous[Curre
nt[train]].next[Next[train]]
   65:         state 107 -(tr  10)-> state 102  [id 174 tp   2] [----G] line 660 => (ZV[train].previous[Current[train]].next[Next[train]])
   66:         state 107 -(tr   2)-> state  97  [id 181 tp   2] [----G] line 660 => else
   67:         state 102 -(tr  19)-> state 103  [id 178 tp   2] [D---G] line 665 => D_STEP
   68:         state 103 -(tr  12)-> state  58  [id 179 tp   3] [----G] line 670 => c2t[train]!ZLM.type,ZLM.train,ZLM.destination      [(1,2)]
   69:         state  51 -(tr  13)-> state  52  [id 127 tp   2] [D---G] line 583 => D_STEP
   70:         state  52 -(tr  12)-> state  97  [id 128 tp   3] [----G] line 590 => c2t[train]!ZLM.type,ZLM.train,ZLM.destination      [(1,2)]
   71:         state   9 -(tr   7)-> state  39  [id  85 tp   2] [----L] line 532 => assert(0)
   72: state 110 line 495 is a loopstate
   73: state 58 line 610 is a loopstate
   74: state 97 line 648 is a loopstate
   75:
   76: Transition Type: A=atomic; D=d_step; L=local; G=global
   77: Source-State Labels: p=progress; e=end; a=accept;
   78: Note: statement merging was used. Only the first
   79:       stmnt executed in each merge sequence is shown
   80:       (use spin -a -o3 to disable statement merging)
   81:
   82: pan: elapsed time 3.52e+07 seconds
   83: pan: rate         0 states/second
```

Anhang B.3: Input-Files und Verifier-Ausgabe von SPIN

```
/* 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;


----

(Spin Version 5.1.6 -- 9 May 2008)
        + Partial Order Reduction
        + Compression
        + Graph Encoding (-DMA=25)

Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        cycle checks            - (disabled by -DSAFETY)
        invalid end states      +

State-vector 524 byte, depth reached 578, errors: 0
MA stats: -DMA=23 is sufficient
Minimized Automaton:    279308 nodes and 1.24786e+06 edges
 36243553 states, stored
 72101724 states, matched
1.0834528e+08 transitions (= stored+matched)
 23072395 atomic steps
hash conflicts:         0 (resolved)

Stats on memory usage (in Megabytes):
18664.855       equivalent memory usage for states (stored*(State-vector + overhead))
   34.721       actual memory usage for states (compression: 0.19%)
    2.000       memory used for hash table (-w19)
    0.267       memory used for DFS stack (-m10000)
   37.228       total actual memory usage

nr of templates: [ globals chans procs ]
collapse counts: [ 42573 11 173 207 ]
unreached in proctype :init:
        (0 of 33 states)
unreached in proctype driver
        (0 of 41 states)
unreached in proctype controller
        line 532, state 9, "assert(0)"
        line 630, state 66, "assert(0)"
        (2 of 113 states)

pan: elapsed time 2.49e+03 seconds
pan: rate 14569.218 states/second
```

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

direction[1] = forward;

if
::direction[2] = forward;
::direction[2] = backward;
fi;

direction[0] = forward;
direction[3] = backward;


----

(Spin Version 5.1.6 -- 9 May 2008)
        + Partial Order Reduction
        + Compression
        + Graph Encoding (-DMA=25)

Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        cycle checks            - (disabled by -DSAFETY)
        invalid end states      +

State-vector 524 byte, depth reached 623, errors: 0
MA stats: -DMA=23 is sufficient
Minimized Automaton:    306702 nodes and 1.39078e+06 edges
 43153315 states, stored
 87512684 states, matched
1.30666e+08 transitions (= stored+matched)
 28259107 atomic steps
hash conflicts:         0 (resolved)

Stats on memory usage (in Megabytes):
22223.272       equivalent memory usage for states (stored*(State-vector + overhead))
   37.353       actual memory usage for states (compression: 0.17%)
    2.000       memory used for hash table (-w19)
    0.267       memory used for DFS stack (-m10000)
   39.865       total actual memory usage

nr of templates: [ globals chans procs ]
collapse counts: [ 44998 11 173 207 ]
unreached in proctype :init:
        (0 of 33 states)
unreached in proctype driver
        (0 of 41 states)
unreached in proctype controller
        line 532, state 9, "assert(0)"
        line 630, state 66, "assert(0)"
        (2 of 113 states)

pan: elapsed time 3.22e+03 seconds
pan: rate 13382.242 states/second
```

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

direction[1] = forward;

if
:: direction[2] = forward;
:: direction[2] = backward;
fi;

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

if
:: direction[3] = forward;
:: direction[3] = backward;
fi;


----


(Spin Version 5.1.6 -- 9 May 2008)
        + Partial Order Reduction
        + Compression
        + Graph Encoding (-DMA=25)

Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        cycle checks            - (disabled by -DSAFETY)
        invalid end states      +

State-vector 524 byte, depth reached 852, errors: 0
Minimized Automaton:    1211520 nodes and 5.83192e+06 edges
1.8413309e+08 states, stored
3.6986648e+08 states, matched
5.5399957e+08 transitions (= stored+matched)
1.190821e+08 atomic steps
hash conflicts:         0 (resolved)

Stats on memory usage (in Megabytes):
94825.619       equivalent memory usage for states (stored*(State-vector + overhead))
  156.942       actual memory usage for states (compression: 0.17%)
    2.000       memory used for hash table (-w19)
    0.267       memory used for DFS stack (-m10000)
  159.396       total actual memory usage

nr of templates: [ globals chans procs ]
collapse counts: [ 199724 11 231 277 ]
unreached in proctype :init:
        (0 of 39 states)
unreached in proctype driver
        (0 of 41 states)
unreached in proctype controller
        line 532, state 9, "assert(0)"
        line 630, state 66, "assert(0)"
        (2 of 113 states)

pan: elapsed time 1.63e+04 seconds
pan: rate 11312.047 states/second
```

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

direction[1] = forward;

if
:: direction[2] = forward;
:: direction[2] = backward;
fi;

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

direction[3] = backward;

----


(Spin Version 5.1.6 -- 9 May 2008)
        + Partial Order Reduction
        + Compression
        + Graph Encoding (-DMA=25)

Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        cycle checks            - (disabled by -DSAFETY)
        invalid end states      +

State-vector 524 byte, depth reached 887, errors: 0
Minimized Automaton:    1246098 nodes and 5.95798e+06 edges
2.0538338e+08 states, stored
4.1412228e+08 states, matched
6.1950565e+08 transitions (= stored+matched)
1.3162596e+08 atomic steps
hash conflicts:         0 (resolved)

Stats on memory usage (in Megabytes):
105769.179      equivalent memory usage for states (stored*(State-vector + overhead))
  147.353       actual memory usage for states (compression: 0.14%)
    2.000       memory used for hash table (-w19)
    0.267       memory used for DFS stack (-m10000)
  149.826       total actual memory usage

nr of templates: [ globals chans procs ]
collapse counts: [ 167769 11 230 274 ]
unreached in proctype :init:
        (0 of 36 states)
unreached in proctype driver
        (0 of 41 states)
unreached in proctype controller
        line 532, state 9, "assert(0)"
        line 630, state 66, "assert(0)"
        (2 of 113 states)

pan: elapsed time 1.73e+04 seconds
pan: rate 11840.224 states/second
```

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

direction[1] = forward;

direction[2] = forward;
direction[0] = backward;

if
:: direction[3] = forward;
:: direction[3] = backward;
fi;


----

(Spin Version 5.1.6 -- 9 May 2008)
        + Partial Order Reduction
        + Compression
        + Graph Encoding (-DMA=25)

Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        cycle checks            - (disabled by -DSAFETY)
        invalid end states      +

State-vector 524 byte, depth reached 585, errors: 0
MA stats: -DMA=23 is sufficient
Minimized Automaton:    316095 nodes and 1.54582e+06 edges
 49520362 states, stored
 99433642 states, matched
1.48954e+08 transitions (= stored+matched)
 31850210 atomic steps
hash conflicts:         0 (resolved)

Stats on memory usage (in Megabytes):
25502.201       equivalent memory usage for states (stored*(State-vector + overhead))
   41.945       actual memory usage for states (compression: 0.16%)
    2.000       memory used for hash table (-w19)
    0.267       memory used for DFS stack (-m10000)
   44.357       total actual memory usage

nr of templates: [ globals chans procs ]
collapse counts: [ 51482 11 173 207 ]
unreached in proctype :init:
        (0 of 33 states)
unreached in proctype driver
        (0 of 41 states)
unreached in proctype controller
        line 532, state 9, "assert(0)"
        line 630, state 66, "assert(0)"
        (2 of 113 states)

pan: elapsed time 3.86e+03 seconds
pan: rate 12826.617 states/second
```

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

direction[1] = forward;

direction[0] = forward;
direction[2] = backward;

direction[3] = backward;


----

(Spin Version 5.1.6 -- 9 May 2008)
        + Partial Order Reduction
        + Compression
        + Graph Encoding (-DMA=25)

Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        cycle checks            - (disabled by -DSAFETY)
        invalid end states      +

State-vector 524 byte, depth reached 720, errors: 0
MA stats: -DMA=23 is sufficient
Minimized Automaton:    356626 nodes and 1.74572e+06 edges
 56898777 states, stored
1.1463029e+08 states, matched
1.7152907e+08 transitions (= stored+matched)
 36207333 atomic steps
hash conflicts:         0 (resolved)

Stats on memory usage (in Megabytes):
29301.967       equivalent memory usage for states (stored*(State-vector + overhead))
   43.560       actual memory usage for states (compression: 0.15%)
    2.000       memory used for hash table (-w19)
    0.267       memory used for DFS stack (-m10000)
   46.017       total actual memory usage

nr of templates: [ globals chans procs ]
collapse counts: [ 47371 11 172 204 ]
unreached in proctype :init:
        (0 of 30 states)
unreached in proctype driver
        (0 of 41 states)
unreached in proctype controller
        line 532, state 9, "assert(0)"
        line 630, state 66, "assert(0)"
        (2 of 113 states)

pan: elapsed time 4.33e+03 seconds
pan: rate 13131.012 states/second
```

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

direction[1] = forward;

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

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


----

(Spin Version 5.1.6 -- 9 May 2008)
        + Partial Order Reduction
        + Compression
        + Graph Encoding (-DMA=25)

Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        cycle checks            - (disabled by -DSAFETY)
        invalid end states      +

State-vector 524 byte, depth reached 516, errors: 0
MA stats: -DMA=23 is sufficient
Minimized Automaton:    263929 nodes and 1.20974e+06 edges
 37603989 states, stored
 76605506 states, matched
1.142095e+08 transitions (= stored+matched)
 24586263 atomic steps
hash conflicts:         0 (resolved)

Stats on memory usage (in Megabytes):
19365.458       equivalent memory usage for states (stored*(State-vector + overhead))
   32.853       actual memory usage for states (compression: 0.17%)
    2.000       memory used for hash table (-w19)
    0.267       memory used for DFS stack (-m10000)
   35.275       total actual memory usage

nr of templates: [ globals chans procs ]
collapse counts: [ 40231 11 173 207 ]
unreached in proctype :init:
        (0 of 33 states)
unreached in proctype driver
        (0 of 41 states)
unreached in proctype controller
        line 532, state 9, "assert(0)"
        line 630, state 66, "assert(0)"
        (2 of 113 states)

pan: elapsed time 2.51e+03 seconds
pan: rate 14980.296 states/second
```

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

direction[1] = forward;
direction[2] = backward;

direction[0] = forward;
direction[3] = backward;


----

(Spin Version 5.1.6 -- 9 May 2008)
        + Partial Order Reduction
        + Compression
        + Graph Encoding (-DMA=25)

Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        cycle checks            - (disabled by -DSAFETY)
        invalid end states      +

State-vector 524 byte, depth reached 526, errors: 0
MA stats: -DMA=23 is sufficient
Minimized Automaton:     68144 nodes and 307828 edges
  7994131 states, stored
 16477038 states, matched
 24471169 transitions (= stored+matched)
  5339059 atomic steps
hash conflicts:         0 (resolved)

Stats on memory usage (in Megabytes):
 4116.851       equivalent memory usage for states (stored*(State-vector + overhead))
    8.928       actual memory usage for states (compression: 0.22%)
    2.000       memory used for hash table (-w19)
    0.267       memory used for DFS stack (-m10000)
   11.349       total actual memory usage

nr of templates: [ globals chans procs ]
collapse counts: [ 10784 11 116 140 ]
unreached in proctype :init:
        (0 of 30 states)
unreached in proctype driver
        (0 of 41 states)
unreached in proctype controller
        line 532, state 9, "assert(0)"
        line 630, state 66, "assert(0)"
        (2 of 113 states)

pan: elapsed time 444 seconds
pan: rate 18017.379 states/second
```

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

direction[1] = forward;
direction[2] = backward;

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

if
:: direction[3] = forward;
:: direction[3] = backward;
fi;


----

(Spin Version 5.1.6 -- 9 May 2008)
        + Partial Order Reduction
        + Compression
        + Graph Encoding (-DMA=25)

Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        cycle checks            - (disabled by -DSAFETY)
        invalid end states      +

State-vector 524 byte, depth reached 550, errors: 0
MA stats: -DMA=23 is sufficient
Minimized Automaton:    243123 nodes and 1.17092e+06 edges
 33521384 states, stored
 68469880 states, matched
1.0199126e+08 transitions (= stored+matched)
 22183639 atomic steps
hash conflicts:         0 (resolved)

Stats on memory usage (in Megabytes):
17262.981       equivalent memory usage for states (stored*(State-vector + overhead))
   32.677       actual memory usage for states (compression: 0.19%)
    2.000       memory used for hash table (-w19)
    0.267       memory used for DFS stack (-m10000)
   35.080       total actual memory usage

nr of templates: [ globals chans procs ]
collapse counts: [ 42118 11 174 210 ]
unreached in proctype :init:
        (0 of 36 states)
unreached in proctype driver
        (0 of 41 states)
unreached in proctype controller
        line 532, state 9, "assert(0)"
        line 630, state 66, "assert(0)"
        (2 of 113 states)

pan: elapsed time 2.08e+03 seconds
pan: rate 16103.276 states/second
```

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

if
:: direction[1] = forward;
:: direction[1] = backward;
fi;

direction[0] = forward;
direction[2] = backward;

if
:: direction[3] = forward;
:: direction[3] = backward;
fi;


----

(Spin Version 5.1.6 -- 9 May 2008)
        + Partial Order Reduction
        + Compression
        + Graph Encoding (-DMA=25)

Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        cycle checks            - (disabled by -DSAFETY)
        invalid end states      +

State-vector 524 byte, depth reached 605, errors: 0
MA stats: -DMA=23 is sufficient
Minimized Automaton:    290749 nodes and 1.40647e+06 edges
 42392199 states, stored
 85983236 states, matched
1.2837544e+08 transitions (= stored+matched)
 27900098 atomic steps
hash conflicts:         0 (resolved)

Stats on memory usage (in Megabytes):
21831.310       equivalent memory usage for states (stored*(State-vector + overhead))
   38.719       actual memory usage for states (compression: 0.18%)
    2.000       memory used for hash table (-w19)
    0.267       memory used for DFS stack (-m10000)
   41.134       total actual memory usage

nr of templates: [ globals chans procs ]
collapse counts: [ 50259 11 174 210 ]
unreached in proctype :init:
        (0 of 36 states)
unreached in proctype driver
        (0 of 41 states)
unreached in proctype controller
        line 532, state 9, "assert(0)"
        line 630, state 66, "assert(0)"
        (2 of 113 states)

pan: elapsed time 2.96e+03 seconds
pan: rate 14335.152 states/second
```