Formale Fehlerbaumanalyse

Dissertation

zur Erlangung des Doktorgrades Dr. rer. nat
der Fakultät für Angewandte Informatik
der Universität Augsburg

vorgelegt von

Andreas Thums

aus Crailsheim

2004
Universität Augsburg
Fakultät für Angewandte Informatik
Lehrstuhl für Softwaretechnik und Programmiersprachen
Prof. Dr. Wolfgang Reif

Amtierender Dekan:	Prof. Dr. Wolfgang Reif
Gutachter:	Prof. Dr. Wolfgang Reif Prof. Dr. Walter Vogler
Tag der Prüfung:	18. Juni 2004

Danksagung

Ich möchte mich bei allen herzlich bedanken, die mich bei diese Arbeit tatkräftig unterstützt haben.

An erster Stelle gilt der Dank meinem Betreuer Prof. Dr. Wolfgang Reif, der mir diese Arbeit ermöglicht hat und sie mit seinen Anregungen und Ratschlägen in die richtigen Bahnen lenkte.

Der weitere Dank gilt meinen Kollegen Dr. Gerhard Schellhorn, Michael Balser, Christoph Duelli, Dominik Haneberg, Frank Ortmeier und Kurt Stenzel für die tatkräftige Unterstützung bei der Lösung fachlicher und technischer Probleme. Hervorheben möchte ich Dr. Gerhard Schellhorn und Michael Balser für die hilfreichen inhaltlichen Diskussionen zu theoretischen Teilen der Arbeit und die Unterstützung bei Implementierungsarbeiten.

Für das Korrekturlesen meiner Arbeit danke ich meiner Schwester Barbara Thums und meiner Freundin Isabel Vogler.

Schließlich möchte ich noch meiner Familie und meiner Freundin Isabel Vogler für den Rückhalt im privaten Umfeld danken, der mir die notwendige Motivation für diese Arbeit gab.

Augsburg, März 2004.

Kurzfassung

Diese Arbeit integriert die Sicherheitsanalysetechnik *Fehlerbaumanalyse* (FTA) aus dem Bereich der Ingenieurwissenschaften in *formale Methoden* aus dem Bereich der Softwaretechnik. Wir nennen diesen integrierten Ansatz *formale FTA*. Die klassische FTA untersucht die Ausfallsicherheit von Systemen, während die formalen Methoden auf die funktionale Korrektheit fokussieren. Die formale FTA verknüpft die zwei, in ihrem jeweiligen Anwendungsgebiet bewährten, Techniken zu einer durchgängigen Analysemethode für Spezifikationen sicherheitskritischer Anwendungen. Das Anwendungsgebiet der formalen FTA sind softwarebasierte, eingebettete Systeme.

Die Integration der FTA in formale Methoden besteht in der Definition einer formalen Semantik. Um die dynamischen Aspekte softwarebasierter eingebetteter Systeme abbilden zu können, wird die FTA-Semantik in Intervalltemporallogik (ITL) definiert. Aus der FTA-Semantik lassen sich Bedingungen ableiten, deren Nachweis garantiert, dass die FTA vollständig durchgeführt wurde, also keine Ausfallursachen bei der Analyse übersehen wurden.

Um diese FTA-Bedingungen über einer Spezifikation softwarebasierter eingebetteter Systeme nachweisen zu können, wird eine Spezifikationssprache benötigt, die dynamische Aspekte dieser Systeme abbilden kann und ebenso den Nachweis der FTA-Bedingungen in ITL ermöglicht. Dazu werden Statecharts in ITL integriert, ein Beweiskalkül entwickelt und prototypisch implementiert. Das interaktive Verifikationswerkzeug KIV dient dabei als Entwicklungsplattform. Es können nun FTA-Bedingungen über Statechart-Spezifikationen nachgewiesen werden. Die Besonderheit des Statechart-Kalküls ist, dass er algebraisch spezifizierte Datentypen in Statechart-Spezifikationen erlaubt und damit sehr allgemeine und insbesondere nicht zustandsendliche Datentypen in Statecharts verwendet werden können.

Die Arbeit wird durch eine Anwendungsmethodik für die formale FTA abgerundet, die durch ein Werkzeug unterstützt wird. Es wurde die Möglichkeit geschaffen, Fehlerbäume über Statechart-Spezifikationen in KIV zu erstellen und die Beweisbedingungen für die formale FTA zu erzeugen. Die Beweisbedingungen können dann mit dem entstandenen Statechart-Kalkül nachgewiesen werden. Damit ist eine vollständige methodische und werkzeugseitige Unterstützung für die formale FTA entstanden.

Inhaltsverzeichnis

1 Motivation und Überblick **1**

2 Fehlerbaumanalyse **7**

2.1 Terminologie 7

2.1.1 Klassifikation von Fehlern 8

2.1.2 Sicherheit 9

2.1.3 Weitere Systemqualitäten 10

2.2 Klassische Fehlerbaumanalyse (FTA) 11

2.2.1 Die Technik 11

2.2.2 Minimale Schnittmengen 13

2.2.3 Bewertung 14

2.3 FTA in der Softwaretechnik 15

3 Beispiele **17**

3.1 Drucktank 18

3.1.1 Fehlerbaumanalyse 19

3.2 Funkbasierter Bahnübergang 21

3.2.1 Fehlerbaumanalyse 23

4 Statecharts **27**

4.1 Spezifikation mit Statecharts 27

4.1.1 Grundlagen 28

4.1.2 Ausführen von Statecharts 33

4.2 Formalisierung von Statecharts 35

4.2.1 Abstrakte Syntax 35

4.2.2 Semantik 40

4.3 Spezifikation von Fehlverhalten 47

5 Formale Fehlerbaumanalyse **51**

5.1 Motivation der FTA-Semantik 52

5.2 Intervalltemporallogik (ITL) 57

5.3 Formalisierung in ITL 59

5.3.1 Semantik der Fehlerbaumgatter 59

5.3.2 Semantik der FTA . . . 62
5.4 Das minimale Schnittmengen-Theorem . . . 64
5.5 Verwandte Arbeiten . . . 66

6 Verifikation von Fehlerbäumen 71
6.1 Das Verifikationswerkzeug KIV . . . 72
6.1.1 Algebraische Spezifikation . . . 72
6.1.2 Prädikatenlogik . . . 73
6.1.3 Dynamische Logik . . . 73
6.1.4 Intervalltemporallogik . . . 74
6.2 Statechart-Spezifikationen in KIV . . . 78
6.2.1 Syntax . . . 78
6.2.2 Timeout-Ereignisse und Schedule-Aktionen . . . 80
6.2.3 Semantische Integration in ITL . . . 81
6.3 Statechart-Kalkül . . . 82
6.3.1 Regelschema für den Statechart-Kalkül . . . 82
6.3.2 Beweis-Beispiel: Zeitschaltuhr . . . 87
6.3.3 Kalkül-Regel für Statecharts . . . 96
6.4 Korrektheit der Statechart-Kalkülregeln . . . 104
6.4.1 Die *sc unwind*-Regel . . . 105
6.4.2 Die *sc wellformed*-Regel . . . 124
6.4.3 Die *sc initialize*-Regel . . . 125
6.5 Beweistechnik und -automatisierung . . . 127
6.5.1 Beweistechnik . . . 127
6.5.2 Beweisheuristiken . . . 127
6.6 Zusammenfassung . . . 128

7 Modellprüfung von Fehlerbäumen 131
7.1 Clocked Computation Tree Logic (CCTL) . . . 132
7.1.1 E/A-Intervallautomaten . . . 132
7.1.2 Semantik von CCTL . . . 135
7.2 Vollständigkeitsprüfung in (C)CTL . . . 137
7.2.1 Formalisierung der Fehlerbaumgatter . . . 137
7.2.2 Akzeptor-Automaten . . . 139
7.3 Beispiel: Drucktank . . . 142
7.4 Verwandte Arbeiten . . . 149
7.5 Resultate . . . 150

8 Formaler Vergleich von FTA-Semantiken 151
8.1 Erweiterte ITL . . . 151
8.2 Vergleich der FTA-Semantiken . . . 156
8.2.1 FTA-Formalisierung von Hansen et al. . . . 156
8.2.2 FTA-Formalisierung von Bruns und Anderson . . . 158

8.2.3 FTA-Formalisierung in CTL . . . 158
8.2.4 FTA-Formalisierung in ITL . . . 160
8.2.5 Zusammenfassung . . . 160
8.3 Spezifikation der erweiterten ITL in KIV . . . 162
8.4 Äquivalenz der FTA-Semantiken . . . 164
8.4.1 Punktuelle Ereignisse . . . 164
8.4.2 Die Akzeptor-Automaten-Konstruktion . . . 168

9 Methodik formaler FTA **173**
9.1 Phasen einer formalen FTA . . . 173
9.1.1 Anforderungsanalyse (I) . . . 174
9.1.2 Sicherheitsanalyse (II) . . . 176
9.1.3 Integration (III) . . . 177
9.1.4 Auswertung (IV) . . . 180
9.2 Anwendungen . . . 180
9.2.1 Lose gekoppelte formale FTA . . . 180
9.2.2 Eng gekoppelte formale FTA . . . 182
9.3 Zusammenfassung . . . 183

10 Formale FTA: Funkbasierter Bahnübergang **185**
10.1 Spezifikation . . . 185
10.1.1 Spezifikation von Fehlverhalten . . . 187
10.1.2 Bremskurvenberechnung . . . 188
10.2 FTA: Kollision im funkbasierten Bahnübergang . . . 190
10.2.1 Nachweis der Vollständigkeit . . . 193
10.3 Verwandte Arbeiten . . . 195
10.4 Zusammenfassung . . . 195

11 Werkzeugunterstützung **197**
11.1 Statecharts in KIV . . . 197
11.1.1 Spezifikation des funkbasierten Bahnübergangs . . . 198
11.1.2 Beweisunterstützung . . . 199
11.2 Fehlerbäume in KIV . . . 201
11.3 Statistik und Resultate . . . 203

12 Zusammenfassung **205**

13 Ausblick **209**
13.1 Formale FTA . . . 209
13.2 Statechart-Verifikation . . . 211

Anhang **215**

A Software FTA **215**

B KIV-Fallstudie: Minimale Schnittmengen **219**
B.1 Beweis: Vollständigkeit 220
B.2 Beweis: Korrektheit 223

C Algebraische Spezifikation von Fehlerbäumen **227**

D Spezifikation des funkbasierten Bahnübergangs **243**

E Syntax und Semantik sequentieller Programme **251**

F Grammatik von Statechart-Spezifikationen **255**

Literaturverzeichnis **259**

KAPITEL 1

Motivation und Überblick

Mit Beginn der 60er Jahre wurden Techniken zur systematischen Analyse sicherheitskritischer Systeme entwickelt. Dazu gehören die Fehlerbaumanalyse (fault tree analysis, FTA [VGRH81]), die Hazard and Operability Analysis (HAZOP [FMNP95]) und die Fehlermöglichkeit und Einflussanalyse (failure mode and effect analysis, FMEA [Rei79]). Seit der Entwicklung dieser Sicherheitsanalysetechniken haben sich die Kontrollsteuerungen technischer Anlagen grundlegend geändert. Bestanden sie früher aus analogen Steuer- und Regelungsschaltungen, so bestehen sie heute aus digitalen Mikrocontrollern, die in den technischen Anlagen eingebettet sind und deren Kontrolllogik in Software implementiert ist.

Die Schwierigkeiten, die sich bei der Analyse dieser sogenannten softwarebasierten eingebetteten Systemen ergeben, lassen sich exemplarisch am Absturz der Ariane 5 im Juni 1996 illustrieren. Kurz nach dem Start kam die Ariane 5 von der vorgegebenen Flugkurve ab und explodierte. Die Untersuchung des Unfalls der Ariane 5 [Lio96] ergab, dass in der Steuerungssoftware des Trägheitsnavigationssystems die Umrechnung der Horizontalgeschwindigkeit von einer 64-Bit „Real“-Zahl in einen 16-Bit „Integer“-Wert zu einem Puffer-Überlauf führte. Der gleiche Fehler trat auch im Ersatzsystem auf und das Trägheitsnavigationssystem schaltete sich samt Ersatzsystem komplett ab. Der Ausfall des redundant ausgelegten Trägheitsnavigationssystems wurde vom Hauptsystem nicht erwartet und ein Diagnosebitmuster, das vom Trägheitsnavigationssystem noch kurz vor dem Abschalten geschickt wurde, interpretierte der Hauptrechner als Flugdaten und änderte daraufhin die Flugkurve drastisch ab. Die Rakete drohte auseinanderzubrechen und aktivierte die Selbstzerstörung.

Das Fatale an dem Puffer-Überlauf bei der Umrechnung der Horizontalgeschwindigkeit war, dass der Softwarefehler auf einem Einsatz im falschen Kontext beruhte. Die Steuerungssoftware wurde für die Ariane 4 entwickelt, die nur eine geringere Horizontalgeschwindigkeit erreichte und deshalb keinen Puffer-Überlauf erzeugen konnte. Diese Steuerung wurde aber fast unverändert in die Ariane 5 übernommen.

Ein weiteres Beispiel für tragische Fehler in softwarebasierten eingebetteten Systemen sind die Therac-25 Unfälle zwischen Juni 1985 und Januar 1987. Der Therac-25 war ein

computergesteuerter Elektronenbeschleuniger, der in der Strahlentherapie eingesetzt wurde und bei 6 Unfällen zu schweren Verletzungen bis hin zum Tod von Patienten durch zu hohe Strahlenkonzentrationen führte [LT93]. Durch fehlerhafte Software konnten spezielle Eingabesequenzen von Steuerdaten den Therac-25 in einen Zustand versetzen, in dem sehr hohe Strahlendosen erzeugt wurden. Die ebenfalls in Software realisierte Sicherheitsprüfung, die solche Zustände erkennen und verhindern sollte, wurde bei jeder 64. Benutzung fälschlicherweise ausgelassen (ein 6-Bit Zähler wurde Null) und konnte deshalb nicht verhindern, dass Patienten einer Strahlendosis ausgesetzt wurden, die teilweise dem 100-fachen der gewünschten Dosis entsprach.

Obwohl der Therac-25 mit einer FTA untersucht wurde, sind diese Fehler nicht entdeckt worden. Im Fehlerbaum wurde die Software explizit von der Analyse ausgenommen, da durch extensives Testen das Auftreten von Softwarefehlern nahezu ausgeschlossen wurde. Diese Annahme führte dazu, dass erst nach $1\frac{1}{2}$ Jahren die Software als mögliche Ursache für die Strahlenüberdosen genauer analysiert wurde.

Beide Beispiele verdeutlichen, dass die herkömmlichen Methoden zur Analyse sicherheitskritischer Systeme, wie z. B. die FTA, nicht mehr ausreichende Sicherheit garantieren, wenn die Steuerung in Software implementiert ist. In der Ariane 5 war das redundant ausgelegte Trägheitsnavigationssystem auf *zufällige* Hardwarefehler ausgerichtet. Beim Defekt des Trägheitsnavigationssystems sollte das Ersatzsystem dessen Funktionalität übernehmen. Jedoch führte ein *systematischer* Softwarefehler dazu, dass beide Einheiten, die eigentliche Steuerung und die Sicherung, gleichzeitig ausgefallen sind. Bei der Analyse des Therac-25 wurde zwar die Hardware mit FTA sicherheitstechnisch untersucht, die Software aber von der Analyse ausgeschlossen. Aus beiden Beispielen folgt, dass es bei der Analyse softwarebasierter eingebetteter Systeme nicht tragbar ist, die Hardwarekomponenten und deren Steuerungssoftware getrennt zu betrachten. Es ist eine durchgängige Sicherheitsanalyse notwendig.

An diesem Punkt knüpft diese Arbeit an. Wir entwickeln eine durchgängige Methode zur Erstellung sicherer Spezifikationen für softwarebasierte eingebettete Systeme. Dazu integrieren wir eine typische ingenieurwissenschaftliche Sicherheitsanalysemethode, die FTA, in formale Methoden aus dem Bereich der Softwaretechnik. Beide Methoden haben sich in ihrer Anwendungsdomäne längst bewährt. Die FTA analysiert Systeme bezüglich Ausfallsicherheit. Für eine Gefährdung oder einen möglichen Unfall wird untersucht, wie die Systemsicherheit von Ausfällen oder Defekten einzelner Systemkomponenten abhängt. Im Gegensatz dazu analysieren formale Methoden die funktionale Korrektheit eines Systems. Ausgehend von einem präzisen Modell des Systems, das sowohl die Hardwarekomponenten als auch die Steuerungssoftware beschreibt, werden Eigenschaften, die vom System erfüllt werden müssen, mit mathematischen Mitteln nachgewiesen. Die Integration liefert eine durchgängige Methode, die *formale Fehlerbaumanalyse*, mit der softwarebasierte eingebettete Systeme als Ganzes betrachtet und die Stärken der FTA und der formalen Methoden zusammengeführt werden. Mit der formalen FTA können wir nun softwarebasierte eingebettete Systeme, wie die Ariane 5 und der Therac-25, sicherheitstechnisch untersuchen. Da die Hardwarekomponenten und die Softwaresteuerung nicht mehr in getrennten Formalismen betrachtet werden, muss die Software nicht, wie beim Therac-25, von der Sicherheits-

analyse ausgeschlossen werden. Inkonsistenzen zwischen Hardware und Software, die bei der Ariane 5 zum Puffer-Überlauf und schließlich zum Absturz führten, werden dadurch entdeckt.

Zusätzlich zur Analyse der funktionalen Sicherheit und der Ausfallsicherheit softwarebasierter eingebetteter System bietet der hier beschriebene Ansatz der formalen FTA die Möglichkeit, mathematisch korrekt nachzuweisen, dass die FTA *vollständig* ist, d. h. dass jede mögliche Ursache für einen Systemausfall in der Analyse betrachtet wird. Dies ist mit der herkömmlichen FTA nicht möglich, da sie auf einer informellen Systembeschreibung beruht.

Für die Integration der FTA in formale Methoden entwickeln wir in dieser Arbeit eine präzise Semantik für Fehlerbäume. Aus der Semantik leiten wir Vollständigkeitsbedingungen ab, deren Nachweis garantiert, dass, außer der im Fehlerbaum beschriebenen Ursachen, keine weiteren zu Gefährdungen durch das System führen können. Für die Verifikation der Vollständigkeitsbedingungen schaffen wir die notwendige Werkzeugunterstützung zur Modellierung eingebetteter Systeme und zur Beweisführung. Schließlich schlagen wir eine Anwendungsmethodik für die formale FTA vor, die einen effektiven und effizienten Einsatz ermöglicht.

Der Ausgangspunkt dieser Arbeit ist die klassische FTA, die wir in Kapitel 2 einführen. Das Ergebnis einer FTA für eine gegebene Gefährdung sind Mengen von Ereignissen (Ausfälle oder Defekte), die, wenn sie zusammen auftreten, zu der untersuchten Gefährdung führen. Diese Mengen von Ereignissen werden *Schnittmengen* genannt. *Minimale Schnittmengen* (engl. minimal cut sets) sind kleinstmögliche Mengen von Ereignissen, die zusammen zur untersuchten Gefährdung führen. Sie bilden den Ausgangspunkt für eine qualitative und quantitative Analyse zur Einschätzung der Wahrscheinlichkeit der Systemgefährdung und Ermittlung der Schwachstellen des Systems. In der Literatur existieren bereits einige Ansätze, die herkömmliche FTA für die Analyse von Software anzupassen bzw. zu erweitern. Diese Ansätze sind für softwarebasierte eingebettete Systeme interessant und werden zum Abschluss des Kapitels diskutiert. Wir gehen in unserer Arbeit mit der Integration von FTA und formalen Methoden jedoch über diese Ansätze hinaus.

In Kapitel 3 präsentieren wir die Anwendung der klassischen FTA an zwei Beispielen, die in späteren Kapiteln auch zur Erläuterung und Evaluierung der formalen FTA dienen. Das erste Beispiel, der *Drucktank*, stammt aus dem *Fault Tree Handbook* [VGRH81], dem Standardwerk zur Fehlerbaumanalyse. Das zweite Beispiel ist ein funkgesteuerter Bahnübergang. Diese Fallstudie ist der sogenannten *FunkFahrBetrieb* (FFB, [FFB96]), ein Projekt der Deutschen Bahn AG, und wird im DFG[1]-Schwerpunktprogramm „Integration von Techniken der Softwarespezifikation für ingenieurwissenschaftliche Anwendungen“ [Sof98] behandelt. Der funkbasierte Bahnübergang ist von uns bereits sicherheitstechnisch analysiert worden [RST00a], und wir präsentieren hier die wesentlichen Ergebnisse der FTA.

Nachdem wir die klassische FTA eingeführt und deren Anwendung an zwei Beispielen demonstriert haben, stellen wir in Kapitel 4 Statecharts vor. In den Ingenieurwissenschaf-

[1] Deutsche Forschungsgemeinschaft.

ten haben sich Statecharts [Har87] zur Modellierung eingebetteter Systeme durchgesetzt. Statecharts sind erweiterte endliche Zustandsautomaten, die wir zur Beschreibung dynamischer Systeme verwenden. Zustandsautomaten, insbesondere Statecharts, dienen uns in dieser Arbeit zur präzisen Spezifikation der Systemmodelle für die formale FTA. Aus diesem Grund präsentieren wir neben der informellen Beschreibung von Statecharts eine formale Semantik. Mit Statecharts modellieren wir das gesamte System, also nicht nur die Softwaresteuerung, sondern auch die Hardwarekomponenten. Hardwarekomponenten können ausfallen oder Defekte aufweisen, die wir in der formalen FTA betrachten. Die möglichen Ausfälle von Komponenten müssen deshalb im formalen Modell berücksichtigt werden, und wir zeigen einige Möglichkeiten, wie man Ausfälle und Fehlverhalten mit Statecharts modelliert. Im Allgemeinen können zur Beschreibung des Systemmodells für die formale FTA aber beliebige Spezifikationstechniken, deren Semantik auf Folgen von Zuständen basiert, eingesetzt werden. Die vorgestellte Modellierung von Fehlern ist leicht auf solche Formalismen übertragbar.

Mit der klassischen FTA und den Statecharts als Spezifikationsmethode haben wir die Grundlage für die formale FTA geschaffen. Am Beispiel des funkbasierten Bahnübergangs zeigen wir in Kapitel 5 Schwächen der herkömmlichen FTA, die sich bei der Anwendung auf softwarebasierte eingebettete Systeme ergeben [RST00b]. Auf der Grundlage von FTA-Formalisierungen aus der Literatur [HRS94, BA93] diskutieren wir notwendige Anforderungen einer Formalisierung der FTA. Die FTA-Formalisierungen aus der Literatur erfüllen nicht alle aufgestellten Anforderungen. Deshalb erweitern wir die klassische FTA und geben eine formale Semantik hierfür an [STR02]. Die Semantik ist in Intervalltemporallogik (ITL, [Mos85, CMZ02]) formuliert, um die dynamischen Aspekte eingebetteter Systeme wiedergeben zu können. Aus der Semantik werden *Vollständigkeitsbedingungen* für die FTA abgeleitet. Erfüllt das Systemmodell diese Bedingungen, garantiert das *minimale Schnittmengen*-Theorem, dass der Fehlerbaum bezüglich dem Modell vollständig ist, d. h. es gibt keine weiteren Ursachen für die Gefährdung. Hierzu präsentieren wir das minimale Schnittmengen-Theorem und führen den formalen Beweis, dass die von uns entwickelte Semantik der formalen FTA auch tatsächlich das Schnittmengen-Theorem erfüllt.

Für den Nachweis der Vollständigkeitsbedingungen einer FTA benötigen wir einen Beweiskalkül, der den Nachweis von ITL-Bedingungen über Statechart-Spezifikationen erlaubt. Für die werkzeugseitige Unterstützung wählen wir das interaktive Spezifikations- und Verifikationswerkzeugs KIV [BRS+00]. KIV unterstützt algebraische Spezifikationen zur Beschreibung funktionalen Verhaltens sowie ITL zum Nachweis temporaler Eigenschaften. Darauf aufbauend wollen wir in Kapitel 6 eine Unterstützung von Statechart-Spezifikationen [BT02] schaffen. Basierend auf der Semantikdefinition von Statecharts aus Kapitel 4 entwickeln wir eine Spezifikationsmöglichkeit für Statecharts über algebraischen Spezifikationen und eine Erweiterung des ITL-Kalküls um Regeln für Statecharts. Mit dem erweiterten ITL-Kalkül können wir dann ITL-Eigenschaften über Statechart-Spezifikationen nachweisen. Mit den Erweiterungen können in KIV dynamische Systeme mit Statecharts spezifiziert, eine formale FTA durchgeführt und die Vollständigkeitsbedingungen nachgewiesen werden.

Als Alternative zur interaktiven Verifikation besteht die Möglichkeit, Eigenschaften

zustandsendlicher Systeme mit Modellprüfung automatisch nachzuweisen. Die Modellierung der Systeme für die Modellprüfung erfolgt meist mit Zustandsautomaten. Unserer Kenntnis nach gibt es keinen frei verfügbaren ITL-Modellprüfer, mit dem die Vollständigkeitsbedingungen der FTA nachgewiesen werden können. Viele verfügbare Modellprüfer benutzen zur Beschreibung der Eigenschaften die *Computation Tree Logic* (CTL, [CE81]). In Kapitel 7 definieren wir FTA-Bedingungen für formale Fehlerbäume in CTL [TS03]. Die CTL-Bedingungen für die formale FTA entsprechen für eingeschränkte Fehlerbäume der ITL-Semantik. Die Einschränkungen diskutieren wir ebenfalls in Kapitel 7. Mit den CTL-Bedingungen weisen wir die Vollständigkeit der FTA für das Drucktank-Beispiel nach und demonstrieren damit die Anwendbarkeit der formalen FTA mit CTL-Modellprüfern.

Bei der intensiven Analyse alternativer FTA-Semantiken haben sich einige Schwächen aber auch Gemeinsamkeiten der Semantiken aus der Literatur herausgestellt [TSR02]. In Kapitel 8 präsentieren wir detailliert Unterschiede verschiedener Semantiken. Wir betrachten dabei zwei FTA-Semantiken aus der Literatur [HRS94, BA93] und die von uns vorgestellten Semantiken in ITL und CTL. Es zeigt sich, dass nur die ITL-Semantik die dynamischen Eigenschaften eingebetteter Systeme korrekt widerspiegelt. Unter gewissen Einschränkungen sind die vier Semantiken aber äquivalent. Wir beschreiben die notwendigen Einschränkungen, präsentieren eine Formalisierung der Semantiken in KIV und einen formalen Beweis der Äquivalenz.

Bisher haben wir die FTA so erweitert, dass sie die Anforderungen eingebetteter Systeme erfüllt. Für die Integration mit formalen Methoden haben wir eine formale Semantik der FTA entwickelt und die notwendige, interaktive und automatische Beweisunterstützung für die Durchführung der formalen FTA geschaffen. Für den Einsatz in realen Projekten geben wir in Kapitel 9 eine Anwendungsmethodik für die formale FTA an [TS02, ORS+03]. Die Methodik hat sich aus verschiedenen Fallstudien heraus entwickelt und rundet die theoretische Arbeit zur formalen FTA ab.

Zur praktischen Anwendung der formalen FTA haben wir in KIV ein Werkzeug zur Erstellung formaler Fehlerbäume integriert. Mit dieser Werkzeugunterstützung untersuchen wir in Kapitel 10 die Fallstudie des funkbasierten Bahnübergangs sicherheitstechnisch. Wir spezifizieren die Fallstudie mit Statecharts in KIV und erstellen einen Fehlerbaum für eine typische Gefährdung in diesem Szenario. Für die formale FTA formalisieren wir die Ereignisse des Fehlerbaums bezüglich dem formalen Statechart-Modell, erzeugen die sich daraus ergebenden Vollständigkeitsbedingungen und weisen sie nach.

Im Kapitel 11 stellen wir dann die Werkzeugunterstützung, die während dieser Arbeit entstanden ist, vor. Dies ist die erwähnte Unterstützung für die Erstellung formaler Fehlerbäume und die Spezifikations- und Beweisunterstützung für Statecharts. Sie erlaubt, Fehlerbäume zu erstellen und erzeugt die Vollständigkeitsbedingungen für die Gatter im Fehlerbaum als Beweisverpflichtungen über einem formalen Statechart-Modell. Diese Beweisverpflichtungen prüfen wir mit der Beweisunterstützung für Statecharts. Das Beispiel des funkbasierten Bahnübergangs aus dem vorhergehenden Kapitel zeigt, dass eine formale FTA für Statechart-Spezifikationen mit interaktivem Nachweis der Beweispflichten mit der geschaffenen Werkzeugunterstützung möglich ist.

Schließlich fassen wir in Kapitel 12 die Ergebnisse der Arbeit zusammen und geben in

Kapitel 13 einen Ausblick auf interessante Anknüpfungspunkte für weitere Forschungen.

KAPITEL 2

Fehlerbaumanalyse

Die Fehlerbaumanalyse (fault tree analysis, FTA) ist eine Sicherheitsanalysetechnik, die 1961 in den Bell Telephone Laboratories entwickelt wurde. Die FTA untersucht Anlagen oder technische Systeme daraufhin, ob sie Gefährdungen verursachen können. Dazu werden systematisch alle Ursachen für eine mögliche Gefährdung analysiert und das Gefährdungspotential aus den Ursachen abgeleitet. Seit ihrer Einführung hat die FTA sowohl in der Luft- und Raumfahrttechnik als auch in der Elektrotechnik und Nuklearindustrie weite Verbreitung gefunden und wird inzwischen auch für softwarebasierte Systeme eingesetzt.

In der Sicherheitsanalyse werden Begriffe benutzt, die umgangssprachlich nicht so streng definiert sind. Außerdem unterscheiden sich teilweise Begriffsdefinitionen in den Ingenieurwissenschaften von denen im Software-Engineering. Wir benutzen den nächsten Abschnitt dazu, die Terminologie der Sicherheitsingenieure einzuführen und sie auf die Softwareanalyse zu übertragen. Anschließend beschreiben wir ausführlich die FTA für technische Systeme. Technische Anlagen haben sich seit der Entwicklung der FTA in den 60er Jahren jedoch grundlegend geändert. Analoge Steuerungen sind mehr und mehr durch digitale Mikrocontroller mit Softwaresteuerungen ersetzt worden. Damit die softwarebasierten Kontrollsteuerungen den Sicherheitsanforderungen analoger Schaltungen genügen, muss auch die Software sicherheitstechnisch analysiert werden. Deshalb diskutieren wir zum Abschluss diese Kapitels einige Sicherheitsanalysetechniken für softwarebasierte Systeme, die auf der FTA basieren.

2.1 Terminologie

In diesem Abschnitt definieren wir die für unseren Argumentationszusammenhang wichtigsten Begriffe aus der Sicherheitsanalyse. Dazu gehören *Fehler*, *Gefährdung*, *Sicherheit*, *fehlerhaftes*, *punktuelles* und *temporales Ereignis*. Die Fachliteratur über Sicherheitsanalyse stammt meist aus dem englischen Sprachraum. Deshalb nutzen wir diesen Abschnitt auch dazu, deutsche Begriffe zu definieren und sie den entsprechenden englischen Fachbegriffen gegenüberzustellen. Dabei halten wir uns überwiegend an die VDI/VDE Richtlinie

3542 [VDI95] und der DIN 400 41 [DIN90]. Die Begriffsdefinitionen übernehmen wir aus *Safeware – System, Safety, and Computers* von Leveson [Lev95].

2.1.1 Klassifikation von Fehlern

Ursachen für Sicherheitsmängel sind immer Fehler, sei es in der Planung, Entwicklung, Produktion oder im Einsatz. Für die Sicherheitsanalyse ist es sinnvoll, den Begriff „Fehler“ zu verfeinern. Im Englischen gibt es im Bereich der Sicherheitsanalyse die drei Begriffe *failure*, *error* und *fault*, die im Allgemeinen mit Fehler übersetzt werden. Wir definieren im Folgenden eigenständige Begriffe dafür.

> **Ausfall (failure):** Ein Ausfall ist ein Nichtausführen oder die Unfähigkeit eines Systems bzw. einer Komponente, ihre Funktionalität für eine gegebene Zeitspanne unter gegebenen Einflüssen auszuführen.

Ursachen für den Ausfall eines Gerätes können entweder sein, dass das Gerät so entwickelt wurde, dass es seine vorgegebene Aufgabe nicht erfüllen kann, oder dass es durch einen Defekt nicht mehr dazu in der Lage ist.

> **Fehlerzustand (error):** Ein Fehlerzustand ist ein Designfehler oder eine Abweichung vom gewünschten Zustand.

Der Unterschied zwischen Ausfall und Fehlerzustand ist, dass der Ausfall zu einem bestimmten Zeitpunkt auftritt, er ist ein Ereignis. Ein Fehlerzustand ist jedoch statisch vorhanden. Software kann nicht ausfallen, jedoch kann ein Rechner ausfallen, wenn die Hardware einen Defekt aufweist oder eine Software abgearbeitet wird, die einen Fehlerzustand erreicht.

> **Störung (fault):** Eine Störung ist ein Ereignis, das ein System in einen ungewünschten Zustand versetzt.

Im Gegensatz zu einem Ausfall ist eine Störung damit ein übergeordnetes Fehlerereignis. Wenn ein Ventil öffnet, weil es fälschlicherweise einen Befehl dazu bekommen hat, weist die Komponente eine Störung, jedoch keinen Ausfall auf, denn sie hat wie gefordert reagiert. Im Allgemeinen führt jeder Ausfall zu einer Störung, jedoch liegt nicht jeder Störung ein Ausfall zugrunde.

Häufig wird die Störung noch weiter in *primäre*, *sekundäre* und *Anweisungsstörung* untergliedert. Eine Komponente hat eine primäre Störung, wenn eine Störung eintritt, obwohl die Komponente in dem für sie vorgesehenen Modus betrieben wird. Ursachen können fehlende Wartung oder Fehler in der Herstellung sein. Fällt eine Komponente durch Überbeanspruchung aus, d. h. ihre Belastung übersteigt den Wert, für den sie konstruiert wurde, liegt eine sekundäre Störung vor. Diese Art von Störung tritt meist zufällig während der Laufzeit auf. Eine Anweisungsstörung liegt vor, wenn die Komponente wie erwartet reagiert, jedoch das Steuersignal zum falschen Zeitpunkt eintrifft. Die Ursachen liegen dann in der Steuerlogik.

Die Begriffe können analog auf die Softwareanalyse übertragen werden. Eine primäre Störung liegt vor, wenn ein Programm aufgrund eines (Design-) Fehlers eine falsche Ausgabe berechnet, eine sekundäre Störung, wenn nicht erwartete Eingaben auftreten und eine Anweisungsstörung, wenn ein Programm wie erwartet auf Eingaben reagiert, aber die Eingaben zum falschen Zeitpunkt oder in der falschen Reihenfolge ankommen.

> **fehlerhaftes Ereignis (fault event):** Ein fehlerhaftes Ereignis ist ein Ausfall, ein Fehlerzustand oder eine Störung.

Ein fehlerhaftes Ereignis fasst alle Fehlerarten zusammen und beschreibt unerwünschte Bedingungen, Zustände und Zustandswechsel. Damit ist der Begriff „Ereignis“ weiter gefasst als in der Automatentheorie. Dort beschreibt ein Ereignis einen Zustandswechsel, der zu einem bestimmten Zeit*punkt* eintritt und keine Dauer hat. In der Sicherheitsanalyse kann ein Ereignis auch Zeit*räume* beschreiben. Ein Ereignis nach der Definition aus der Automatentheorie bezeichnen wir im Folgenden als punktuelles Ereignis.

> **punktuelles Ereignis (timeless event):** Ein punktuelles Ereignis tritt zu einem Zeitpunkt ein.

Möchten wir punktuelle Ereignisse und Ereignisse nach der allgemeineren Definition aus der Sicherheitsanalyse unterscheiden, bezeichnen wir die allgemeineren Ereignisse als temporale Ereignisse.

> **temporales Ereignis (temporal event):** Ein temporales Ereignis beschreibt ein Ereignis, das für einen Zeitraum besteht, also eine Dauer hat.

2.1.2 Sicherheit

Die Auswirkungen von Fehlern sind sehr unterschiedlich, je nachdem, ob sie vom System noch abgefangen werden oder nicht, und je nachdem, welche äußeren, nicht beeinflussbaren Umstände vorhanden sind.

> **Unfall (accident):** Ein Unfall ist ein unerwünschtes, nicht geplantes Ereignis, das einen bestimmten Verlust nach sich zieht.

Ein Unfall unterscheidet sich von einer Störung dadurch, dass er einen gewissen Verlust nach sich zieht. Dieser Verlust kann eine Beschädigung oder ein Verlust einer Anlage, Verletzungen an Menschen oder Verlust von Menschenleben sein. Deshalb werden Unfälle in kleinere, ernste und katastrophale Unfälle eingeteilt.

> **Vorfall (incident):** Ein Vorfall ist ein Ereignis, das keinen Verlust nach sich zieht, aber unter anderen Umständen das Potential dazu hätte.

Ein Vorfall wäre, wenn giftige Substanzen in die Luft gelangen, aber keine Menschen in der Nähe sind, die gefährdet werden. Dieser Vorfall hat das Potential zu einem Unfall. Er hätte in einem bewohnten Gebiet auftreten können.

> **Gefährdung (hazard):** Eine Gefährdung ist ein Zustand oder eine Menge von Bedingungen eines Systems, welche zusammen mit anderen, nicht kontrollierbaren Bedingungen der Umgebung zu einem Unfall führen können.

Damit ist eine Gefährdung unmittelbar mit der Systemumgebung verknüpft. Da die Umgebung nicht kontrolliert werden kann, werden neben Gefährdungen, die zu einem Unfall führen, auch diejenigen Zustände betrachtet, die zu einem Vor- bzw. Zwischenfall führen.

Software an sich stellt keine Gefährdung dar. Sie kann keinen verletzen und nichts beschädigen. Deshalb ist für die Gefährdungsanalyse von softwarebasierten Systemen wichtig, das System, in dem die Software eingesetzt wird, mit zu betrachten. Denn dieses kann durch Softwarefehler in einen gefährlichen Zustand versetzt werden. Jedoch muss die Systemgrenze so gewählt werden, dass Ereignisse, die innerhalb dieser Grenze liegen, auch kontrolliert bzw. beeinflusst werden können. Ereignisse, die nicht beeinflusst werden können, gehören zur Umgebung.

Eine Gefährdung wird durch ihren Grad (schlimmster Unfall, den sie verursachen kann) und ihrer Eintrittswahrscheinlichkeit charakterisiert. Die Eintrittswahrscheinlichkeit kann sowohl qualitativ als auch quantitativ angegeben werden. Beide Faktoren gehen in die Definition des Risikos ein.

> **Risiko (risk):** Das Risiko ist der Grad einer Gefährdung kombiniert mit der Wahrscheinlichkeit, dass die Gefährdung zu einem Unfall führt, und der Dauer des gefährlichen Zustands.

Um das Risiko einer Gefährdung beurteilen zu können, ist es notwendig, die Umgebung sinnvoll zu definieren. Von der Umgebung hängt die Wahrscheinlichkeit ab, die zu einem Unfall führt. Das Risiko, das durch eine geladenen Pistole entsteht, unterscheidet sich z. B. gewaltig davon, ob die Pistole in einer großen Menschenmenge getragen wird oder in einem verlassenen Waldstück.

Nachdem wir die Risiken und Gefährdungen betrachtet haben, können wir schließlich Sicherheit definieren.

> **Sicherheit (safety):** Sicherheit ist die Abwesenheit von Unfällen und Vorfällen.

2.1.3 Weitere Systemqualitäten

Im deutschen Sprachgebrauch wird Sicherheit auch im Zusammenhang mit dem unerlaubten Zugang zu Informationen und unerlaubtem Ausführen von Aktivitäten benutzt. Als Stichworte seien hier nur die sichere Authentisierung, z. B. Zugriffskontrolle bei Kreditkarten durch PIN, und Verschlüsselung zum Schutz von Daten vor unberechtigtem Zugriff genannt. Im Englischen gibt es dafür den eigenständigen Begriff *security*.

Eine weitere Qualität eines Systems ist die *Zuverlässigkeit* (reliability). Zuverlässigkeit misst, wie gut ein System seine vorgegebene Aufgabe erfüllt. Es ist jedoch zu beachten, dass eine Zunahme der Zuverlässigkeit nicht notwendigerweise eine Zunahme der Sicherheit

nach sich zieht. Es kann sogar der entgegengesetzte Effekt eintreten. Wenn die Zuverlässigkeit einer Pistole erhöht wird, kann sie möglicherweise leichter feuern und wird deshalb unsicherer im Gebrauch.

Neben der Zuverlässigkeit gibt es noch die *Verfügbarkeit* (availability) als Systemqualität. Die Verfügbarkeit hängt von der Anzahl der Ausfälle eines Systems innerhalb einer gewissen Zeitspanne ab. Bei Schutzeinrichtungen erhöht eine hohe Verfügbarkeit die Sicherheit. Jedoch ist ein Flugzeug dann am Besten vor einem Absturz geschützt, wenn es nicht abhebt, selbst wenn dies auf einen nicht verfügbaren Antrieb zurückzuführen ist.

Die Gesamtqualität eines Systems, bestehend aus Sicherheit (safety und security), Zuverlässigkeit und Verfügbarkeit, wird mit *Verlässlichkeit* (dependability) beschrieben. Auch die *RAMS*-Eigenschaften (reliability, availability, maintainability, safety) beurteilen die Gesamtqualität eines Systems. Sie betrachten auch die *Wartbarkeit* (maintainability) eines Systems. In der obigen Diskussion sieht man aber, dass die Zunahme einzelner Qualitätsfaktoren nicht zwangsläufig zur Zunahme der Gesamtqualität eines Systems führt. Dieser Punkt wird oft vernachlässigt.

2.2 Klassische Fehlerbaumanalyse (FTA)

Es ist ein wesentlicher Aspekt einer Sicherheitsanalyse, das System bezüglich Gefährdungen und deren Ursachen zu untersuchen. Dazu kann die Fehlerbaumanalyse (*fault tree analysis, FTA* [VGRH81, Lei95]) eingesetzt werden. Die FTA ist eine top-down Analysemethode, die Gefahren systematisch auf ihre Ursachen herunterbricht und das Ergebnis in einer Baumstruktur darstellt. Damit hilft die FTA weniger, Gefährdungen zu entdecken, als diese systematisch zu analysieren.

2.2.1 Die Technik

Mit der Fehlerbaumanalyse werden technische Systeme, wie z. B. Kernreaktoren oder Chemieanlagen, daraufhin untersucht, ob sie bestimmte Gefährdungen (Austritt radioaktiver Substanzen, giftiger Chemikalien usw.) verursachen können.

Als erster Schritt wird bei einer Fehlerbaumanalyse die Systemgrenze genau festgelegt. Dazu gehört die Definition der Gefährdung, der Ereignisse, die bei der Analyse berücksichtigt werden, der physikalischen Systemgrenzen und des initialen Systemzustands. Es macht beispielsweise einen großen Unterschied, ob ein Bremssystem für Autos im Stadtverkehr oder auf der Autobahn untersucht wird und die Geschwindigkeit $30\frac{\mathrm{km}}{\mathrm{h}}$ oder $130\frac{\mathrm{km}}{\mathrm{h}}$ beträgt. Sind die Systemgrenzen definiert, wird das Eintreten der Gefährdung (Top-Ereignis im Fehlerbaum), schrittweise auf seine Ursachen zurückgeführt. Dazu wird eine graphische Notation verwendet, die in der IEC 1025 Norm [IEC90] standardisiert wurde (siehe Abbildung 2.1).

Die Fehlerbaumanalyse verknüpft Ursachen für ein Ereignis (Ausgangsereignis) mit entsprechenden Gattern. Je nachdem, ob eine der Ursachen oder alle Ursachen notwendig sind, um das Ausgangsereignis auszulösen, wird das Ereignis durch ein *Oder-Gatter* bzw.

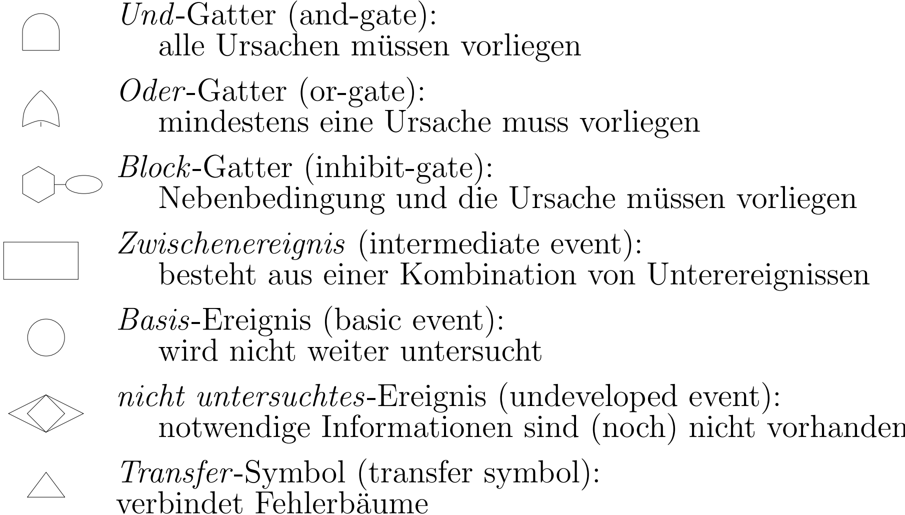


Abbildung 2.1: Symbole für die Fehlerbaumanalyse nach IEC 1025

ein *Und-Gatter* mit den Ursachen (Unterereignissen) verknüpft. Ein *Block-Gatter* führt zwischen einem Ereignis und der entsprechenden Ursache eine Nebenbedingung ein. Die Nebenbedingung muss zusätzlich zur Ursache vorhanden sein, damit die Wirkung eintritt. Die Bedingung beschreibt Ereignisse, die keine Fehler oder Defekte sind und im „Normalbetrieb“ auftreten. Um einen großen Fehlerbaum anschaulich zu präsentieren, können ganze Unterbäume durch ein *Transfer*-Symbol markiert und separat analysiert werden.

Die im Fehlerbaum definierten Ursachen sind *Zwischenereignisse*, die weiter untersucht werden, bis ein gewünschter Detaillierungsgrad erreicht wird. Ursachen, die nicht weiter untersucht werden, sind Blätter im Fehlerbaum. Blätter sind entweder *Basisereignisse* des Systems oder Ereignisse, die für die Analyse (noch) nicht detailliert genug beschrieben wurden (*nicht untersuchte Ereignisse*). Nicht untersuchte Ereignisse können z. B. Ausfälle von Anlagenteilen betreffen, die im bisherigen Systemdesign noch nicht vollständig beschrieben sind.

Eine wesentliche Eigenschaft der Ereignisse in einem Fehlerbaum ist, dass sie *unerwünscht* sind. Sie beschreiben Fehlerzustände, Störungen oder Ausfälle. Diese Eigenschaft wird im Fehlerbaum-Handbuch [VGRH81] Seite IV-3 explizit gefordert: „an intermediate event is a fault event that occurs ...“.

Ein Beispiel für einen Fehlerbaum ist in Abbildung 2.2 zu sehen. Die Gefährdung *Motor überlastet* wird auf die Ursachen *Sicherung defekt* und *Strom liegt an* zurückgeführt. Deshalb sind die Ursachen über ein *Und*-Gatter mit dem Top-Ereignis verbunden. Das Ereignis *Strom liegt an* wird weiter untersucht. Entweder wird es durch das Ereignis *Relais öffnet nicht* oder durch *Kontrollschaltung liefert 'ein'* verursacht. Diese beiden Ereignisse sind mit einem *Oder*-Gatter verknüpft. Da die Kontrollschaltung nicht weiter bekannt ist, wird dieser Fehler als nicht untersuchtes Ereignis gekennzeichnet. Die beiden anderen Blätter im Fehlerbaum sind Basisereignisse. Weitere Beispiele zur FTA finden sich in Kapitel 3.

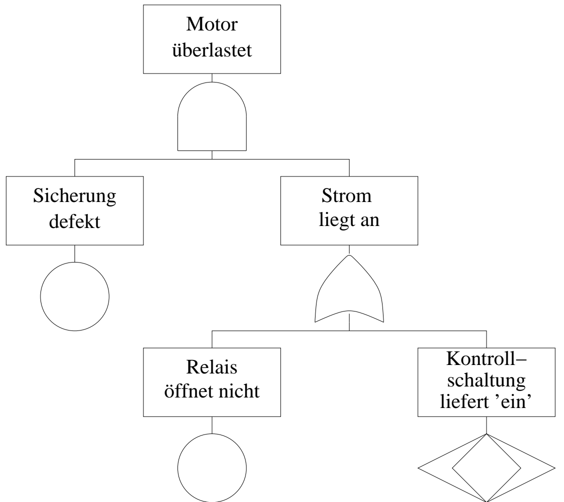


Abbildung 2.2: Fehlerbaum

2.2.2 Minimale Schnittmengen

Die Basisereignisse des Fehlerbaums werden in sogenannte *Schnittmengen* (*cut sets*) zusammengefasst. Eine Schnittmenge beschreibt eine Menge von Basisereignissen, die zusammen zur untersuchten Gefahr führen. Eine Schnittmenge ist *minimal*, wenn keine echte Teilmenge wiederum eine Schnittmenge ist. Eine minimale Schnittmenge besteht also aus einer kleinsten Menge von Basisereignissen, die zusammen die Gefährdung auslösen können. Kann ein Basisereignis einer minimalen Schnittmenge ausgeschlossen werden, so kann diese Schnittmenge das Top-Ereignis nicht mehr verursachen.

Schnittmengen können leicht aus der Struktur des Fehlerbaums abgeleitet werden. Interpretiert man ein *Und*-Gatter als Konjunktion, ein *Oder*-Gatter als Disjunktion, die Basisereignisse als atomare Formeln und den Fehlerbaum als Syntaxbaum einer booleschen Formel, so sind die minimalen Schnittmengen die Disjunktive Normalform (DNF) dieser Formel. Für den Fehlerbaum aus Abbildung 2.2 ergeben sich die beiden minimalen Schnittmengen

$$\{\{\text{Sicherung def.}, \text{Rel. öffnet nicht}\}, \{\text{Sicherung def.}, \text{Kontr. liefert ein}\}\}.$$

Die minimalen Schnittmengen werden zur qualitativen Untersuchung des Systems eingesetzt. Einelementige minimale Schnittmengen entsprechen sogenannten *single point of failures*. Der Ausfall einer Komponente genügt, um die Systemgefährdung zu verursachen. Ein Hinweis auf weitere Schwachstellen eines Systems ist, wenn ein Basisereignis in mehreren Schnittmengen auftritt (Sicherung defekt, im obigen Beispiel). Der Schwachpunkt ist die Komponente, die das entsprechende Basisereignis verursacht.

Neben einer qualitativen Analyse können die minimalen Schnittmengen auch zu einer quantitativen Sicherheitsbewertung eingesetzt werden. Ausgangspunkt für eine quantitative Bewertung sind die Eintrittswahrscheinlichkeiten der Basisereignisse. Im Allgemeinen

sind Ausfallwahrscheinlichkeiten von Komponenten oder Bauteilen bekannt, denn technische Anlagen bestehen häufig aus Standardkomponenten, für die statistische Daten vorhanden sind. Diese Daten können in der Analyse eingesetzt werden.

Für eine einfache quantitative Analyse muss das Eintreten der Ereignisse der minimalen Schnittmengen statistisch unabhängig sein. Die Wahrscheinlichkeit, dass eine minimale Schnittmenge die Gefährdung verursacht, wird dann durch Multiplikation der Eintrittswahrscheinlichkeit der einzelnen Ereignisse in der minimalen Schnittmenge berechnet. Die Eintrittswahrscheinlichkeit der Gefährdung ergibt sich (für kleine Werte) aus der Summe der Wahrscheinlichkeiten aller minimalen Schnittmengen. Unter den gegebenen Voraussetzungen ist für die Schnittmengen $s_i = \{e_1, \ldots, e_{n_i}\}$, $i = 1 \ldots m$, einer Gefährdung h die Wahrscheinlichkeit, dass die Gefährdung eintritt:

$$P(h) = \sum_{i=1}^{m} \prod_{j=1}^{n_i} P(e_j)$$

Die Ausfallwahrscheinlichkeit für den Motor aus dem Fehlerbaum in Abbildung 2.2 berechnet sich mit dieser Methode durch

$$\begin{aligned} P(\text{Motor überl.}) &= P(\text{Sicherung def.}) * P(\text{Rel. öffnet nicht}) + \\ &\quad P(\text{Sicherung def.}) * P(\text{Kontr. liefert ein}). \end{aligned}$$

Präzisere Berechnungsmodelle, die auch bedingte Wahrscheinlichkeiten betrachten, werden in [VGRH81] beschrieben.

2.2.3 Bewertung

Für die Fehlerbaumanalyse ist ein detailliertes Systemdesign notwendig. Deshalb kann eine aussagekräftige FTA erst in den späteren Entwicklungsphasen, jedoch vor der Systemfertigstellung, durchgeführt werden. Die FTA wird teilweise auch in den früheren Entwicklungsphasen durchgeführt, um frühzeitig das Gefährdungspotential einer Anlage wenigstens einschätzen zu können.

Die FTA strukturiert die Systemanalyse und geht dabei komponentenbasiert vor. Ursachen für ein Fehlerereignis einer Komponente stammen aus den Unterkomponenten der defekten Komponente. Durch die schrittweise Anwendung der FTA beschreibt jede Ebene im Fehlerbaum Fehler von Komponenten, die auch in der Komponentenhierarchie auf der gleichen Ebene stehen.

Die graphische Darstellung der Fehlerbäume beschreibt übersichtlich die Ergebnisse der Analyse. Neben der Analyse von Schwachstellen dient die FTA auch dazu, ein tiefgehendes Verständnis für die Funktionalität des Systems zu bekommen. Durch die qualitative und quantitative Bewertungsmöglichkeit kann die FTA sogar verschiedene Design-Alternativen eines Systems miteinander vergleichen. Allerdings ist die Fehlerbaumanalyse sehr zeitaufwendig und bei der quantitativen Analyse kann es problematisch werden, wenn nicht alle Ausfallwahrscheinlichkeiten vorhanden sind. Denn Schätzungen resultieren häufig in einem irreführenden Ergebnis.

2.3 FTA in der Softwaretechnik

Es gibt verschiedene Ansätze, die FTA in der Softwaretechnik anzuwenden. Einige Ansätze führen die klassische Fehlerbaumanalyse unverändert für softwarebasierte Systeme durch. Z. B. integriert Dugana [Dug96] die Softwareanalyse in den herkömmlichen Sicherheitsanalyseprozess, indem für das softwarebasierte System eine klassische FTA durchgeführt wird. Sobald man bei der Analyse auf Softwarekomponenten trifft, werden diese vorerst nicht weiter analysiert, sondern als Basisereignisse definiert. Treten schließlich Softwarekomponenten in den minimalen Schnittmengen des Fehlerbaums auf, werden diese Softwarekomponenten bei den folgenden Softwaretests besonders kritisch betrachtet und getestet.

Die FTA kann auch auf Softwarekomponenten oder das Softwaredesign angewendet werden. Eine Schwierigkeit besteht darin, die Software zu strukturieren. Für die Komponentenbildung können Softwaremodule und ihre Hierarchie, oder das Softwaredesign verwendet [Mai97] werden. Schwierigkeiten bereiten die komplexen Abhängigkeiten zwischen verschiedenen Softwarekomponenten, die meist nicht so klar in ihrer Aufgabenverteilung getrennt sind wie Komponenten in (elektro-) technischen Systemen.

Čepin et al. [ČdLM+97] stellen eine Sicherheitsanalysetechnik für die objekt-basierte Modellierung vor. Hier definiert das Objektmodell die Softwarekomponenten und die Abhängigkeiten zwischen den Komponenten. Der Fehlerbaum wird aus dem Objektmodell abgeleitet.

Joyce und Wong [JW98] weichen von der klassischen FTA ab und benutzen Fehlerbäume, um Sicherheitseigenschaften für Software zu berechnen. Ausgehend von einem unerwünschten Ereignis werden schrittweise dessen logische Ursachen abgeleitet. Bei diesem Schritt sind häufig zusätzliche Voraussetzungen an die Software notwendig. Diese Voraussetzungen sind die Sicherheitseigenschaften für die Software. Ziel ist es, schlussendlich einen Widerspruch zwischen Ursachen, die als Blätter im Fehlerbaum vorkommen, und der Systemspezifikation zu zeigen. Aus dem Widerspruch folgt, dass die Ursachen nicht auftreten können. Das System ist also sicher, wenn zusätzlich die bei der Analyse abgeleiteten Sicherheitseigenschaften durch die Softwarekomponenten erfüllt werden.

Leveson hat eine Software-Fehlerbaumanalyse (Software-FTA [LC91, LH83]) für Programme entwickelt, die den vorhandenen Quelltext untersucht und direkt in die System-FTA eingebunden ist. Zunächst wird für das komplette System eine klassische Fehlerbaumanalyse durchgeführt und Fehlerereignisse, die von Software Komponenten stammen, als Basisereignisse markiert. Diese Basisereignisse, beispielsweise „falscher Ergebniswert“ oder „kein Ergebnis“, werden dann mit der Software-FTA genau untersucht.

Die Software-FTA verfolgt im Programmcode den Pfad vom fehlerhaften Ergebnis bis zur Eingabe. Der Programmcode wird so rückwärts, von der Ausgabe zur Eingabe hin, untersucht und analysiert die Werte einzelner Programmvariablen. Als Ergebnis der Software-FTA erhält man die Eingaben für das Programm, die zum Fehler führen. Sogenannte Fehlerbaummuster (*fault tree template*) beschreiben, wie einzelne Programmanweisungen Variablenwerte ändern. Jeder Programmanweisung ist ein Fehlerbaummuster zugeordnet, das definiert, welche Werte bzw. Wertebereiche die Variablen *vor* der Programmanweisung

haben müssen, damit sie zu dem vorhandenen Variablenwert *nach* der Programmanweisung führen. So können schrittweise die Eingaben berechnet werden, die zu einer fehlerhaften Ausgabe führen. Die Software-FTA entspricht damit Ansätzen der Programmverifikation, die für eine gegebene Nachbedingung eines Programms die schwächste Vorbedingung berechnen, die nach Programmausführung zu dieser Nachbedingung führt (siehe *weakest precondition predicate transformer* [Dij76]). Ein Beispiel für eine Software-FTA ist im Anhang A beschrieben.

Zusammenfassend betrachtet gibt es vielversprechende Ansätze zur FTA softwarebasierter Systemen. Die Gemeinsamkeit aller Ansätze besteht aber in einem Methodenbruch zwischen der Analyse der Anlage, die mittels herkömmlicher FTA durchgeführt wird, und der Analyse der Software. Bis auf die Software-FTA, die auf der Semantik des Quelltextes basiert, setzen alle Vorschläge auf informellen Systemmodellen auf. Eine Validierung der Analyse ist damit nicht möglich. Deshalb schlagen wir für die Analyse softwarebasierter Systeme die formale FTA vor. Das Systemmodell und die Kontrollsoftware werden in ein und demselben Formalismus beschrieben. Die formale FTA setzt auf dem präzisen Systemmodell auf, untersucht einheitlich das gesamte Modell und ermöglicht es, die Analyse zu validieren.

KAPITEL 3

Beispiele

In diesem Kapitel stellen wir zwei Beispielszenarien vor, die in dieser Arbeit immer wieder zur Veranschaulichung verschiedener Techniken herangezogen werden. Zur ausführlichen Erläuterung der FTA präsentieren wir für beide Beispiele auch ihre entsprechenden Fehlerbäume. Das erste Beispiel beschreibt eine Steuerung für einen Drucktank. Die Steuerung soll dafür sorgen, dass beim Füllen des Behälters nicht zu viel Flüssigkeit hineingepumpt wird und der Behälter zerplatzt. Dieses Beispiel ist dem Fehlerbaumhandbuch (*Fault Tree Handbook*, [VGRH81]), Beispiel „Pressure Tank", Abschnitt VIII, entnommen, in dem auch die zugehörige klassische FTA beschrieben ist. Deshalb eignet sich dieses Beispiel sehr gut, um die klassische FTA und die formale FTA zu vergleichen.

Das zweite Beispiel ist eine Referenzfallstudie aus einem DFG-Schwerpunktprogramm[1] und beschreibt die dezentrale Ansteuerung eines Bahnübergangs durch den sich annähernden Zug. Der Zug übernimmt selbstständig das Einschalten des Bahnübergangs und kann deshalb den Schließzeitpunkt in Abhängigkeit von seiner Geschwindigkeit wählen. Durch die Betrachtung der Zuggeschwindigkeit sollen Wartezeiten anderer Verkehrsteilnehmer am Bahnübergang minimiert werden. Für das zweite Beispiel, dem funkbasierten Bahnübergang, ist im Rahmen des DFG-Schwerpunktprogramms eine Sicherheitsanalyse mit den beiden Analysetechniken FTA und FMEA (failure modes and effects analysis [Rei79]) durchgeführt worden. Diese Sicherheitsanalyse deckte neben mehreren Inkonsistenzen eine eklatante Sicherheitslücke in der informellen Beschreibung des funkbasierten Bahnübergangs auf [RST00b, RST00a], die daraufhin in Zusammenarbeit mit den Erstellern der Fallstudie geschlossen wurde. Die FTA für die funkbasierte Bahnübergangsteuerung und die entdeckte Sicherheitslücke fassen wir hier kurz zusammen.

[1] DFG-Schwerpunktprogramm „Integration von Techniken der Softwarespezifikation für ingenieurwissenschaftliche Anwendungen" [Sof98].

3.1 Drucktank

Der Drucktank versorgt einen Verbraucher kontinuierlich mit Flüssigkeit. Sobald der Tank leer ist sorgt die Steuerung dafür, dass er wieder vollständig gefüllt wird. Dabei ist zu beachten, dass die Pumpe nicht zu viel Flüssigkeit in den Behälter pumpt und ihn damit zum Platzen bringen.

Die Steuerung des Drucktanks ist in Abbildung 3.1 dargestellt. Drei Stromkreise, c_1, c_2

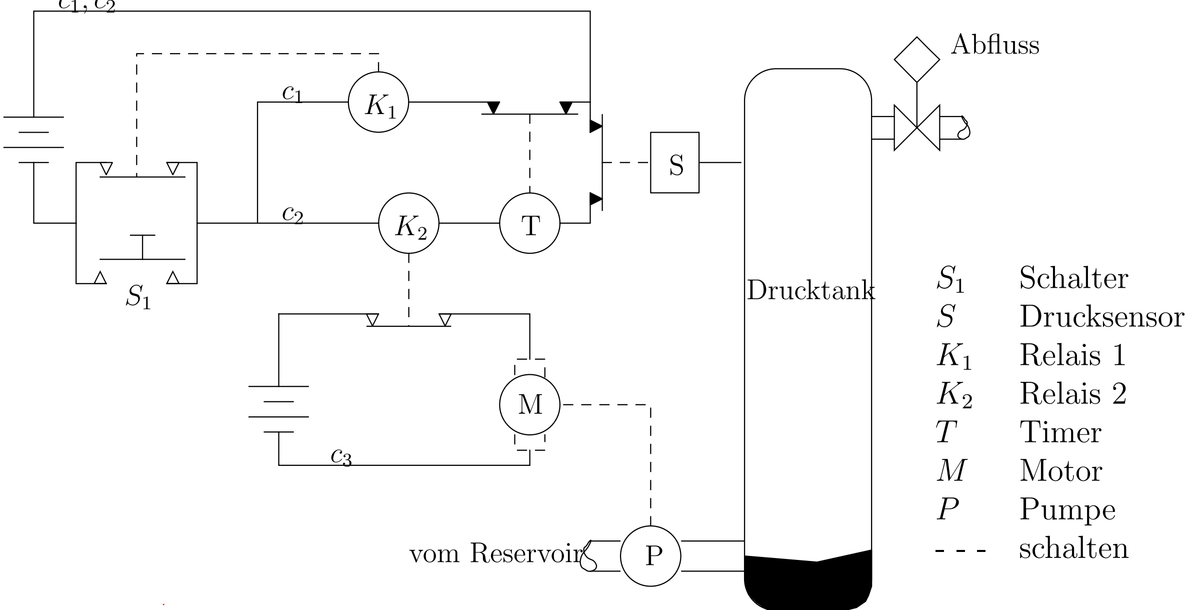


Abbildung 3.1: Der Drucktank

und c_3, versorgen die Steuerung mit Strom. Sie sind mit durchgezogenen Linien gekennzeichnet. Gestrichelte Linien kennzeichnen, dass eine Komponente einen Schalter betätigt. Der Stromkreis c_1 versorgt das Relais K_1 mit Strom und wird durch den Schalter S_1 oder das Relais K_1 und den Timer T geschlossen. Der Stromkreis c_2 versorgt das Relais K_2 und den Timer T mit Strom und wird durch S_1 oder K_1 und den Drucksensor S geschlossen. Von diesen beiden Stromkreisen galvanisch getrennt, versorgt der Stromkreis c_3 den Motor M mit Strom und wird durch das Relais K_2 geschlossen.

Die Steuerung des Drucktanks wird durch Schließen des Tippschalters S_1 in Betrieb genommen, indem der Stromkreis c_1 geschlossen wird. Die Rückkopplungsschleife über das Relais K_1 versorgt die Steuerung auch dann mit Strom, wenn S_1 nicht mehr gedrückt ist. Die Steuerung arbeitet nun autonom. Ist der Behälter leer und damit der Drucksensor S geschlossen, erhält auch der Stromkreis c_2 Strom. Über das nun geschlossene Relais K_2 wird der Motor M für die Pumpe P angesteuert. Der Pumpenmotor ist eingeschaltet und die Pumpe füllt den Behälter. Diese Situation ist in Abbildung 3.1 dargestellt.

Der Sensor S erkennt einen gefüllten Tank und unterbricht den Steuerstromkreis c_2. Das Relais K_2 erhält keinen Strom mehr, unterbricht den Stromkreis c_3 und das Füllen des

Behälters wird beendet. Nun kann dem Behälter solange Flüssigkeit entnommen werden, bis er geleert ist. Dann schließt der Drucksensor S wieder den Stromkreis c_2, das Relais K_2 den Stromkreis c_3 und der Pumpenmotor füllt den Behälter erneut.

Ein Defekt des Drucksensors S (Sensor öffnet nicht) führt dazu, dass der Steuerstromkreis c_2 nicht unterbrochen wird, wenn der Behälter vollständig gefüllt ist. Damit in diesem Fall der Drucktank nicht platzt, unterbricht der Timer T den Versorgungsstromkreis c_1, sobald der Drucktank „lange genug" gefüllt wurde (hier: 60 sek). Der Timer T ist ein Schutzmechanismus für den Ausfall des Drucksensors S. Er unterbricht den Stromkreis c_1 und damit die Rückkopplung über das Relais K_1. Der Steuerstromkreis c_2 erhält keinen Strom mehr, das Relais K_2 öffnet den Stromkreis c_3 und der Motor schaltet ab. Ein erneutes Füllen des Tanks ist in diesem Fall erst wieder möglich, wenn durch Drücken des Tippschalters S_1 der Stromkreis c_1 und der Steuerstromkreis c_2 erneut mit Strom versorgt wird.

Damit bei korrektem Funktionieren des Sensors S ein mehrmaliges Füllen des Drucktanks möglich ist, wird der Timer zurückgesetzt, sobald S den Steuerstromkreis c_2 unterbricht.

3.1.1 Fehlerbaumanalyse

Der Fehlerbaum für den Drucktank ist dem Fehlerbaumhandbuch [VGRH81] entnommen, das neben der FTA auch methodische Aspekte der Analyse beschreibt. Der Fehlerbaum in Abbildung 3.2 untersucht die Gefährdung „Platzen des Drucktanks nach Start des Pumpvorgangs". Nach obiger Beschreibung kann dies geschehen, wenn länger als 60 sek Flüssigkeit in den Behälter gepumpt wird. Nach Konstruktion der Steuerung ist das der Fall, wenn das Relais K_2 länger als 60 sek geschlossen ist. Dazu muss entweder länger als 60 sek Strom am Relais anliegen oder das Relais öffnet nicht, obwohl kein Strom mehr anliegt (Defekt von K_2). Dieses Ereignis wollen wir nicht weiter untersuchen.

Erhält das Relais K_2 länger als 60 sek Strom, liegt ein Fehler in der Steuerung vor. Damit K_2 Strom erhält, muss sowohl der Sensor S geschlossen sein, als auch Strom im Stromkreis c_2 fließen. Dies ist nur der Fall, wenn entweder K_1 oder S_1 geschlossen ist. Der zweite Fall ist wieder ein Basisereignis. K_1 kann geschlossen sein, obwohl kein Strom anliegt (Defekt von K_1), oder es fließt Strom durch den Stromkreis c_1. Dazu muss der Timer allerdings länger als 60 sek geschlossen sein. Dies kann nur dann geschehen, wenn der Timer fehlerhaft arbeitet. Der Defekt des Timers wird in der Analyse nicht weiter untersucht. Die Basisereignisse im Fehlerbaum sind mit dem Namen der Komponente gekennzeichnet, deren Defekt das entsprechende fehlerhafte Ereignis bedingt.

Aus dem Fehlerbaum lassen sich dann die folgenden minimalen Schnittmengen ableiten (siehe Abschnitt 2.2.2):

$$\{\{K_2,\},\{S,S_1\},\{S,K_1\},\{S,T\}\}$$

Damit kann das System qualitativ bewertet werden. Wie man sieht, ist das Relais K_2 eine kritische Komponente, da ihr alleiniger Ausfall zu einem Unfall führen kann. Die drei restlichen Schnittmengen sind weniger kritisch, denn es müssen immer zwei Komponenten

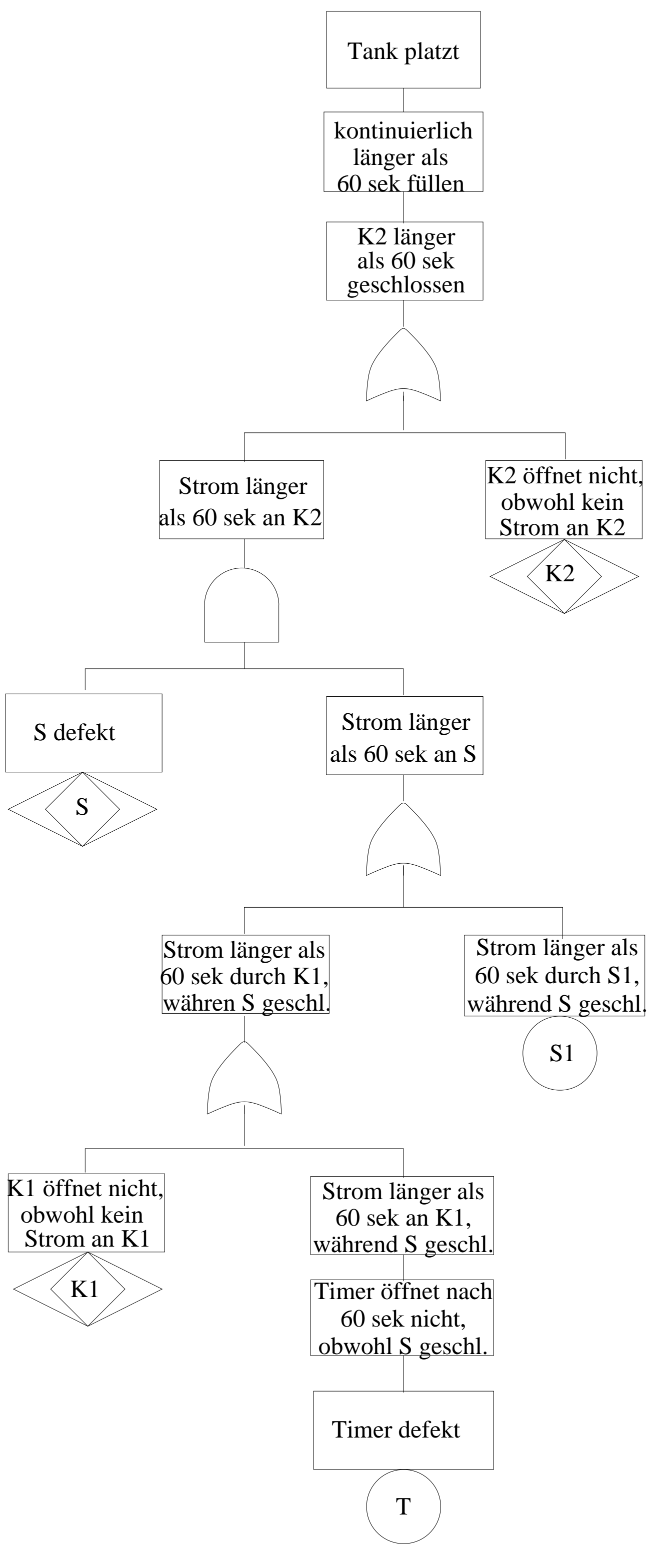


Abbildung 3.2: FTA Drucktank

ausfallen, damit der Tank platzt.

Diese Beobachtung wird durch die quantitative Analyse in Abbildung 3.1 verstärkt. Die

Komponenten:		**Schnittmengen:**	
P(K_2)	3 * 10^{-5}	$\{K_2\}$	3 * 10^{-5}
P(S)	1 * 10^{-4}	$\{S, S_1\}$	3 * 10^{-9}
P(K_1)	3 * 10^{-5}	$\{S, K_1\}$	3 * 10^{-9}
P(T)	1 * 10^{-4}	$\{S, T\}$	1 * 10^{-8}
P(S_1)	3 * 10^{-5}	Gesamt	≈ 3 * 10^{-5}

Tabelle 3.1: Ausfallwahrscheinlichkeiten

Ausfallwahrscheinlichkeiten für die Komponenten und der minimalen Schnittmengen sind ebenfalls dem Fehlerbaumhandbuch entnommen. Dieses quantitative Ergebnis bestätigt die qualitative Analyse. Der Schwachpunkt der Anlage ist das Relais K_2. Ein zweites, redundantes Relais zu K_2 in Reihe geschaltet, würde nicht nur die Wahrscheinlichkeit dieser Schwachstelle auf $9 * 10^{-10}$ drastisch reduzieren, sondern auch die Sicherheit des Gesamtsystems fast um den Faktor 2000 erhöhen. Die Schwachstelle des geänderten Systems wäre dann die Schnittmenge $\{S, T\}$.

3.2 Funkbasierter Bahnübergang

Der funkbasierte Bahnübergang [JS99] ist an ein Projekt der Deutschen Bahn AG, dem sogenannten „FunkFahrBetrieb“ (FFB, [FFB96]) angelehnt. Ziel ist es, auf Strecken mit maximalen Zuggeschwindigkeiten von 160$\frac{\text{km}}{\text{h}}$ das Schließen von Bahnübergängen dezentral durch den sich annähernden Zug zu steuern. Dazu werden Signale und Sensoren auf der Strecke durch Kommunikation zwischen Zug und Bahnübergang und der Berechnung des optimalen Schließzeitpunktes ersetzt. Dies ergibt ein kostengünstigeres Verfahren und erlaubt eine flexible Steuerung des Bahnübergangs in Abhängigkeit von der Zuggeschwindigkeit. Die Wartezeiten am Bahnübergang werden so verkürzt.

Die funkbasierte Bahnübergangsteuerung ist in Abbildung 3.3 skizziert. Anstatt, wie bei herkömmlichen Bahnübergängen, den Zug durch Sensoren zu erkennen, überwacht die Zugsteuerung selbst die Position des Zuges. Nähert sich ein Zug einem Bahnübergang an, dem sogenannten *Gefahrenpunkt*, sendet der Zug dem Bahnübergang ein „Schließsignal“. Die Positionen von Bahnübergängen sind im *Streckenprofil* des Zuges hinterlegt. Die Position und Geschwindigkeit des Zugs misst das sogenannte Odometer. Aus der Position des Bahnübergangs, der aktuellen Position des Zuges und dessen Geschwindigkeit berechnet die Zugsteuerung einen optimalen Zeitpunkt für das Senden des Schließsignals. Der Zeitpunkt ist so gewählt, dass der Bahnübergang rechtzeitig geschlossen ist, bevor der Zug ankommt. Aber auch die Gesamtschließzeit der Schranken wird minimiert, um die Verkehrsteilnehmer nicht unnötig zu behindern.

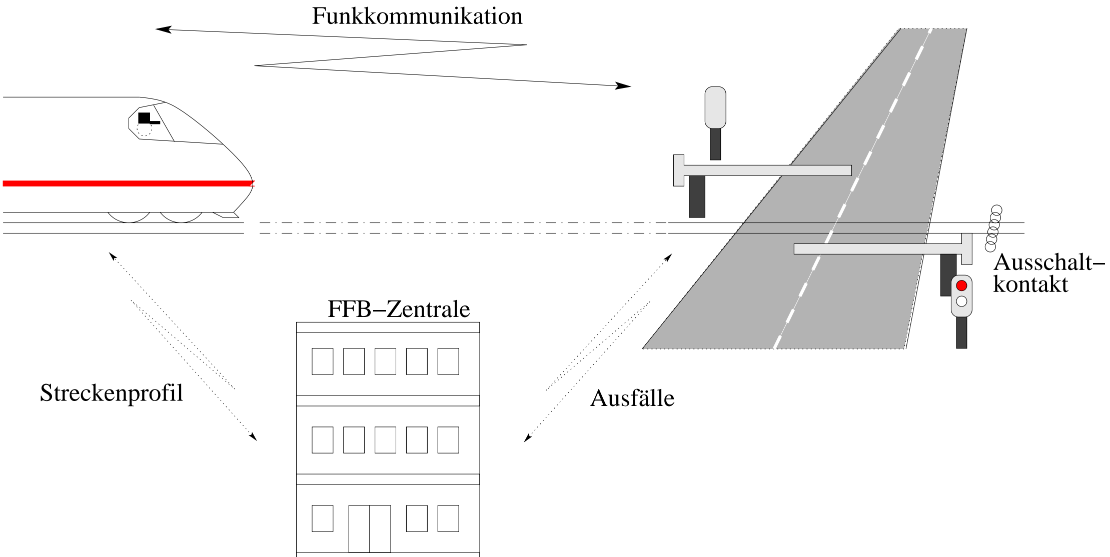


Abbildung 3.3: Funkbasierter Bahnübergang

Sobald die Bahnübergangsteuerung das Schließsignal erhält, sichert sie den Bahnübergang. Es wird die Lichtzeichenanlage eingeschaltet, d. h. zuerst das gelbe Licht und etwas später das rote Licht. Dann beginnen sich die Schranken zu schließen. Sind sie geschlossen, ist der Bahnübergang *gesichert*. Kann jedoch aus technischen Gründen der Sicherungsvorgang nicht vollständig durchgeführt werden, verbleibt der Bahnübergang im *ungesicherten* Zustand. Um lange Wartezeiten am Bahnübergang zu verhindern, öffnen sich die Schranken nach einer maximalen Schließzeit wieder und der Bahnübergang ist ungesichert. Dies geschieht auch, wenn der Zug noch nicht passiert hat.

Kurz bevor der Zug den Punkt erreicht, an dem er spätestens den Bremsvorgang einleiten muss, damit er noch vor dem Bahnübergang zum Stehen kommt, sendet er eine Statusanfrage an den Bahnübergang. Antwortet der Bahnübergang mit *gesichert*, erhält der Zug eine *Freigabe* und passiert den Bahnübergang. Beim Verlassen des Bahnübergangs wird der *Ausschaltkontakt* ausgelöst, der die Sicherung des Bahnübergangs beendet und die Steuerung zum Öffnen veranlasst. Erhält der Zug jedoch bei der Statusanfrage keine Freigabe, ist der Bahnübergang ungesichert. Der Zug muss vor dem Bahnübergang anhalten und kann nur mittels einer manuellen Sicherung den Bahnübergang passieren.

Neben der Schließkontrolle führt die Bahnübergangsteuerung auch noch eine Fehlerdiagnose der Lichtzeichenanlage, der Schranken und des Ausschaltkontakts durch. Bei Defekten muss der Schließvorgang abgeändert, oder kann überhaupt nicht durchgeführt werden. Ist z. B. das Gelblicht defekt, wird bei der Bahnübergangsicherung sofort das Rotlicht eingeschaltet. Es bleibt dafür aber länger aktiv, bevor die Schranken geschlossen werden, so dass die Gesamtvorwarnzeit gleich bleibt. Bei anderen Defekten verweigert die Bahnübergangsteuerung die Sicherung und der Zug erhält bei einer Statusabfrage keine Freigabe. Alle Ausfälle und Defekte werden der *FFB-Zentrale* gemeldet. Sie ist eine übergeordnete

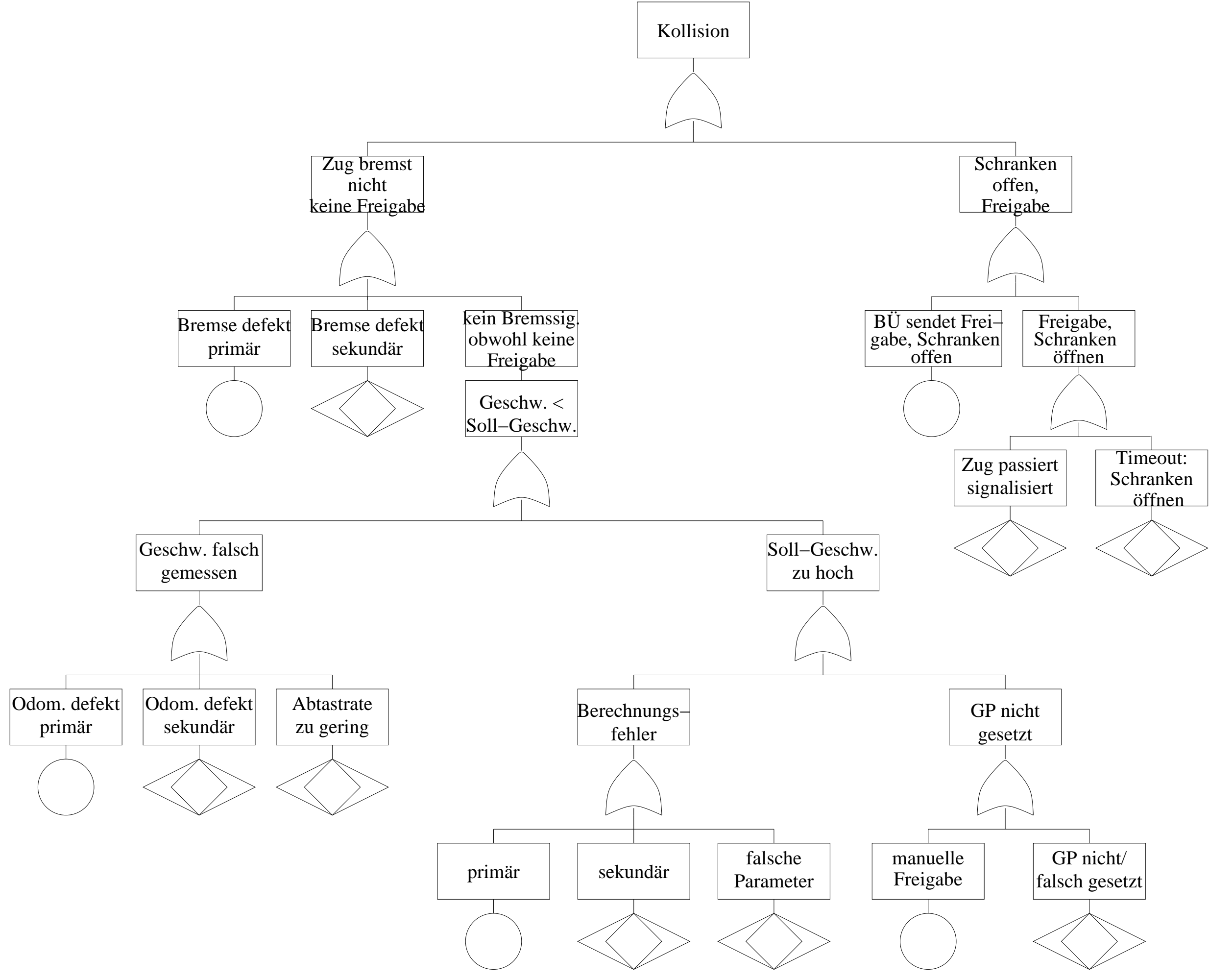


Abbildung 3.4: FTA Kollision

Stelle, die neben der Verwaltung der Ausfälle (die Reparatur wird veranlasst) auch die Daten für das Streckenprofil der Zugsteuerung bereitstellt.

Eine detaillierte Beschreibung der funkbasierten Bahnübergangsteuerung findet sich in [JS99]. Um diese Beschreibung weiter zu präzisieren und Inkonsistenzen zu entdecken und zu beheben, wurde ein (semi-)formales Modell der Bahnübergangsteuerung im Spezifikationswerkzeug STATEMATE [HLN+90] erstellt [KT02, KT00]. Auf dieser Modellierung basiert die folgende FTA.

3.2.1 Fehlerbaumanalyse

In Abbildung 3.4 sehen wir den Fehlerbaum für die Gefahr einer *Kollision*. Eine Kollision des Zuges mit Fahrzeugen ist möglich, wenn sich der Zug auf dem Bahnübergang befindet und die Schranken nicht geschlossen sind. Die FTA versucht nun die Ursachen dafür auf

die einzelnen Komponenten zu verteilen.

Eine Kollision kann entstehen, wenn der Zug nicht bremst, obwohl der Bahnübergang nicht signalisiert hat, dass er gesichert ist (keine Freigabe), *oder* wenn der Bahnübergang sich als gesichert betrachtet (sendet Freigabe), obwohl die Schranken noch offen sind. Das ist die oberste Zerlegung im Fehlerbaum. Die Ursache dafür, dass der Zug nicht bremst, ist ein Defekt der Bremsen (primär oder sekundär, siehe Abschnitt 2.1.1) oder die Bremsen haben kein Brems-Signal bekommen. Dann liegt ein Fehler in der Zugsteuerung vor. Die weitere Analyse der Zugsteuerung kann aus dem Fehlerbaum entnommen werden und wird hier nicht weiter beschrieben.

Auf Seite des Bahnübergangs ist der Grund für eine Freigabe bei offenem Bahnübergang entweder das Senden der Freigabe, obwohl die Schranken nicht geschlossen sind (dies ist ein Basisfehler, den die Bahnübergangsteuerung nicht zulassen darf), oder der Bahnübergang hat eine Freigabe gesendet, öffnet aber die Schranken, obwohl der Zug noch nicht passiert hat. Die Gründe dafür sind entweder, dass die Schranken wieder geöffnet werden, da die maximale Schrankenschließzeit erreicht wurde, oder, dass fälschlicherweise das Passieren des Zugs signalisiert wird. Gründe dafür werden nicht weiter untersucht.

Bei der Auswertung des Fehlerbaums sieht man, dass die Modellierung keine Redundanzen enthält, da keine *Und*-Gatter vorkommen. Deshalb sind die minimalen Schnittmengen einelementig, d. h. alle Basisereignisse sind single point of failures. Des Weiteren erkennt man, dass die manuelle Freigabe, bei der das Zugpersonal den Bahnübergang sichert, ein Risiko darstellt, das nicht durch technische Maßnahmen verringert werden kann. Deshalb müssen strenge Vorschriften für diesen Vorgang definiert werden. Weiterhin kann ein defekter Ausschaltkontakt den Bahnübergang zum Öffnen veranlassen, obwohl der Zug noch nicht passiert hat.

Die FTA zeigt außerdem eine Sicherheitslücke in der Anforderungsbeschreibung der Steuerung auf. Die Beschreibung der Bahnübergangsteuerung fordert, dass nach einer maximalen Schließzeit der Bahnübergang öffnet und als ungesichert zu betrachten ist. Dies gilt auch dann, wenn der Zug den Bahnübergang noch nicht passiert hat. In der ersten Variante der Beschreibung der funkbasierten Bahnübergangsteuerung gab es keinen Hinweis darauf, wie in diesem Fall weiter zu verfahren ist.

Das Szenario für eine mögliche Kollision sieht nun folgendermaßen aus. Der Zug nähert sich einem Bahnübergang und stößt den Schließvorgang an. Die Bahnübergangsteuerung sichert den Bahnübergang, der Zug fordert die Freigabe an und erhält sie. Wegen einer Störung benötigt der Zug aber länger als die maximale Schrankenschließzeit, um den Bahnübergang zu passieren. Ist die maximale Schrankenschließzeit abgelaufen, gilt der Bahnübergang als ungesichert und die Schranken öffnen wieder. Nun kann der Zug, der ja eine Freigabe für den Bahnübergang besitzt, in den geöffneten Bahnübergang einfahren. Das Fehlerszenario ist im Fehlerbaum der Abbildung 3.4 ganz rechts zu sehen („Timeout: Schranken öffnen").

Die Fallstudie hat gezeigt, dass die kombinierte Anwendung der Sicherheitsanalyse mit formalen Modellen sich für softwarebasierte Systeme eignet. Sie hat eine Sicherheitslücke in der Fallstudie entdeckt, die zur Überarbeitung der Beschreibung führte. In der aktuellen Version der Fallstudienbeschreibung ist diese Sicherheitslücke nicht mehr enthalten. Einmal

erkannt, ist die Lösung sehr einfach: die Freigabe für den Zug verfällt nach der maximalen Schrankenschließzeit und der Zug muss erneut eine Freigabe anfordern.

KAPITEL 4

Statecharts

Wir wählen Statecharts zur Spezifikation dynamischer Systeme und als Grundlage für die Validierung formaler Fehlerbäume. Statecharts sind erweiterte Zustandsübergangssysteme. Ein Zustand in einem Statechart repräsentiert eine bestimmte Konfiguration des modellierten Systems. Übergänge beschreiben, welche Konfigurationsfolgen möglich sind. Damit lassen sich dynamische Systeme präzise beschreiben. Statecharts [Har87] haben insbesondere durch das Spezifikationswerkzeug STATEMATE[1] [HP98] eine weite Verbreitung im industriellen Umfeld erlangt. Deshalb werden wir STATEMATE-Statecharts [PS91] zur Spezifikation der Systeme für die formale FTA einsetzen.

Im Folgenden erläutern wir die Grundlagen und das Ausführungsmodell von STATEMATE-Statecharts. Anschließend formalisieren wir die *asynchrone* oder *Makro-Schritt*-Semantik von Statecharts [DJHP98]. Diese Semantik verwenden wir in Kapitel 6 als Grundlage für die formale Statechart-Verifikation. Bei einer formalen FTA beschreiben die Modelle das gesamte System, bestehend aus Software- und Hardwarekomponenten. Hardware kann im Betrieb ausfallen und da wir bei der formalen FTA von diesen Ausfällen nicht abstrahieren, beschreiben wir im letzten Abschnitt, wie mögliche Ausfälle und Fehlverhalten mit Statecharts spezifiziert werden können.

4.1 Spezifikation mit Statecharts

Statecharts [Har87] sind graphische Mittel zur Beschreibung des operationellen Verhaltens dynamischer Systeme. Die wesentlichen Eigenschaften von Statecharts haben Harel et al. [HPSS87] folgendermaßen charakterisiert:

Statechart = Zustandsdiagramm + Hierarchie + Parallelität +
Broadcast-Kommunikation

Statecharts sind im Wesentlichen erweiterte Zustandsdiagramme. Das Hierarchiekonzept erlaubt es, einzelne Zustände detaillierter zu beschreiben. Dazu werden wiederum Zu-

[1] STATEMATE ist ein kommerzielles Werkzeug für modellbasierte Spezifikationen und eingetragenes Warenzeichen von i-Logix.

standsdiagramme benutzt. Parallele Zustände beschreiben nebenläufige Prozesse, die über Broadcast-Kommunikation Daten austauschen und sich synchronisieren.

4.1.1 Grundlagen

Statecharts gibt es in einer Reihe von Varianten, die sich in ihrer Ausdrucksmächtigkeit und Semantik unterscheiden. Eine umfassende Übersicht über verschiedene Statechart-Semantiken hat von der Beeck [vdB94] zusammengestellt. Wir stellen im Folgenden STATEMATE-Statecharts vor.

Zustandsdiagramm

Statecharts bestehen aus einer Menge von *Zuständen* und einer Menge beschrifteter *Übergänge*. Übergänge verbinden die Zustände und sind mit einem Tripel der Form

$$Ereignis[Bedingung]/Aktion$$

beschriftet. Der Zustandsübergang kann ausgeführt werden, wenn das *Ereignis* e vorhanden und die *Bedingung* c erfüllt ist. Wir nennen den Übergang dann *aktiviert*. Beim Übergang wird die *Aktion* a ausgeführt. Bei der Beschriftung sind alle drei Bestandteile optional (Syntax: [e]['['c']'][/a]).

Ein einfaches Statechart besteht aus einer Menge von *Basis*-Zuständen, die mit Übergängen zusammenhängend verknüpft sind. Ein solches Diagramm ist in Abbildung 4.1 zu sehen und wird *Oder*-Zustand genannt. In einem *Oder*-Zustand ist immer *genau* ein *Unterzustand* aktiv.

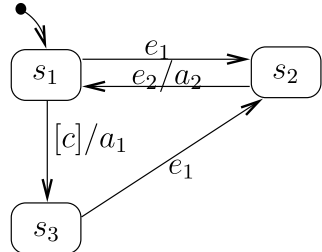


Abbildung 4.1: Oder-Zustand

Ein *Oder*-Zustand wird deshalb auch als *XOr*-Zustand bezeichnet. Der Ausgangszustand s_1 ist durch die eingehende Kante gekennzeichnet. Sobald das Ereignis e_1 eintritt, wird in den Zustand s_2 übergegangen. Ist jedoch die Bedingung c erfüllt, wird die Aktion a_1 ausgeführt und der Zustand s_3 aktiviert. Wenn sowohl das Ereignis e_1 vorliegt als auch die Bedingung c gilt, sind beide Übergangsbedingungen erfüllt, und es wird indeterministisch entweder in den Zustand s_2 oder s_3 gewechselt. *Indeterminismus* ist eine weitere Eigenschaft von Statecharts und besagt, dass Nachfolgezustände nicht immer eindeutig bestimmt sein müssen.

Hierarchie

Das Hierarchiekonzept von Statecharts erlaubt Zustände zusammenzufassen bzw. das Verhalten innerhalb eines Zustands detaillierter zu beschreiben. Das hierarchische Statechart

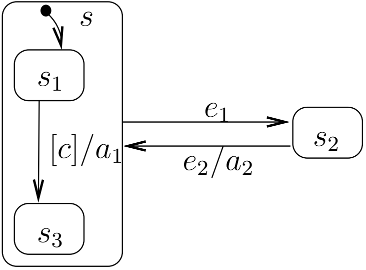


Abbildung 4.2: Hierarchie

in Abbildung 4.2 fasst die beiden Zustände s_1 und s_3 zu s zusammen. Sobald das Ereignis e_1 vorhanden ist, wird der Zustand s, und damit auch die beiden Unterzustände s_1 und s_3, verlassen und der Zustand s_2 aktiv. Die beiden Übergänge von s_1 und s_3 nach s_2 aus Abbildung 4.1 können damit zu einem Übergang zusammengefasst werden.

Das Hierarchiekonzept von Statecharts ermöglicht die *Priorisierung* von Übergängen. So haben Übergänge, die innerhalb eines Zustands stattfinden, eine geringere Priorität als Übergänge, die aus einem Zustand herausführen. Der Übergang von s_1 nach s_3 hat also eine niedrigere Priorität als der Übergang von s nach s_2. Ist der Zustand s_1 (und damit auch s) und das Ereignis e_1 aktiv, während die Bedingung c erfüllt ist, wird im Statechart aus Abbildung 4.2 deterministisch der Übergang in den Zustand s_2 ausgeführt (vgl. Abbildung 4.1, dort entsteht ein Indeterminismus).

Tritt im Zustand s_2 das Ereignis e_2 ein, wird wieder der Zustand s aktiv und, da s_1 durch die eingehende Kante als Ausgangszustand gekennzeichnet ist, auch s_1. Übergänge können auch über Hierarchiegrenzen hinweg gehen. Soll nach dem Zustand s_2 immer der Zustand s_3 aktiv werden, würde der mit e_2/a_2 beschriftete Übergang über die Grenze des Zustands s hinweg direkt zum Zustand s_3 geführt werden.

Parallelität

Für die Darstellung paralleler Prozesse oder Abläufe besitzen Statecharts sogenannte *Und*-Zustände. Wie in Abbildung 4.3 zu sehen, werden zwei (oder mehrere) durch eine gestrichelte Linie getrennte *Oder*-Zustände, zu einem *Und*-Zustand zusammengefasst. Im *Und*-Zustand s sind gleichzeitig beide Unterzustände s_1 und s_2 und z. B. die initialen Zustände s_{1_1} und s_{2_1} aktiv. Die Möglichkeit *Und*-Zustände zu verwenden, vereinfacht die Modellierung nebenläufiger Systeme. Sie erweitern zwar nicht die Ausdrucksmächtigkeit der Sprache – parallele Zustände können auch durch einen Produktautomaten dargestellt werden –, führen aber zu übersichtlicheren und verständlicheren Modellen.

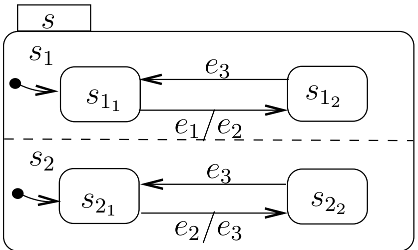


Abbildung 4.3: Und-Chart

Broadcast Kommunikation

Parallele Zustände kommunizieren über den *Broadcast*-Mechanismus miteinander. Das bedeutet, dass Ereignisse und Variablen über die Grenzen des Zustands, in dem sie erzeugt wurden, hinaus sichtbar sind. Sind in Abbildung 4.3 die beiden Ausgangszustände s_{1_1} und s_{2_1} aktiv und es tritt das Ereignis e_1 ein, findet ein Zustandsübergang von s_{1_1} nach s_{1_2} statt und erzeugt das Ereignis e_2. Dieses Ereignis ist wegen der Kommunikation mittels Broadcast im *Oder*-Zustand s_2 sichtbar und führt zum Übergang nach s_{2_2}. Dabei wird das Ereignis e_3 erzeugt, das wiederum in den Zuständen s_1 und s_2 sichtbar ist und in beiden zu einem Übergang nach s_{1_1} bzw. s_{2_1} führt.

In diesem Beispiel hat das Ereignis e_1 zu einer *Kettenreaktion* geführt. Erst nachdem alle oben beschriebenen Übergänge durchgeführt wurden, ist das Statechart wieder in einem *stabilen* Zustand. Es können keine weiteren Übergänge ausgeführt werden, bis Ereignisse „von außen“ (Stimuli der Umgebung) wieder Übergänge aktivieren.

Ereignisse, Bedingungen und Aktionen

Ereignisse unterscheiden sich von Bedingungen dadurch, dass sie nur einmalig *aktiv* sind (wenn sie nicht neu erzeugt werden). Die Kettenreaktion, die im Statechart aus Abbildung 4.3 durch das Ereignis e_1 ausgelöst wird, erzeugt die Ereigniskette $< e_1, e_2, e_3 >$. e_1 ist nicht mehr aktiv, wenn e_2 gilt. e_2 wird beim Übergang von s_{1_1} nach s_{1_2} erzeugt und ist *nur* für den nachfolgenden Übergang von s_{2_1} nach s_{2_2} aktiv, danach nicht mehr.

Übergänge können nicht nur mit einzelnen Ereignissen beschriftet werden. Ereignisse können mit booleschen Operationen ($\wedge$, $\vee$, $\neg$) verknüpft werden, und der entsprechende Übergang ist nur dann möglich, wenn der Ereignis-Ausdruck über der aktuellen Variablenbelegung gilt.

Bedingungen können Vergleichsoperationen zwischen Variablenwerten sein. In STATEMATE sind Operatoren auf natürlichen und reellen Zahlen, Arrays und darauf aufbauenden benutzerdefinierten Datentypen möglich. Außerdem gibt es noch abgeleitete Prädikate wie z. B. $in(s)$. $in(s)$ ist erfüllt, wenn das Statechart „in“ s ist, also der Zustand s aktiv ist. Bedingungen können ebenso wie Ereignisse verknüpft werden und sind erfüllt, wenn der Ausdruck unter der gegebenen Variablenbelegung wahr ist.

Wie erwähnt, können Aktionen Ereignisse erzeugen ($/e$) und Variablen einen Wert

zuweisen ($/x := 5$). Ein Übergang kann auch mehrere Aktionen parallel ausführen ($/a_1;\ a_2$). Komplexe Aktionen können bedingt (**if then else**) und iterativ sein (**for**-Schleife). Für eine ausführliche Behandlung der *Action-Language* von STATEMATE verweisen wir auf [HP98]. Eine korrekte Übergangsbeschriftung wäre z. B.:

$$e_1 \wedge (\neg e_2 \vee e_3)[in(s) \vee x \leq 5 * y]/e;\ \textbf{if } y < z \textbf{ then } x := x^2 \textbf{ else } x := 0$$

Timeout-Ereignisse und Schedule-Aktionen

Timeout-Ereignisse für ein Ereignis e und eine natürliche Zahl n haben die Form $tm(e, n)$ (n kann eine Funktion mit Wertebereich der natürliche Zahlen sein). Ein Übergang $tm(e, n)/a$ wird n Schritte nach dem letzten Auftreten von e aktiv und führt die Aktion a aus. Wird das Ereignis e innerhalb von n Schritten nochmals generiert, wird der Timeout zurückgesetzt. Durch wiederholtes Zurücksetzen kann das Eintreten des Timeouts verhindert werden, der Übergang und die Aktion wird nicht ausgeführt.

Schedule-Aktionen $sc(a, n)$ führen eine Aktion a um n Schritte verzögert aus. Im Gegensatz zum Timeout kann die Schedule-Aktion nicht zurückgesetzt werden und wird sicher ausgeführt.

Konfiguration

Durch die Verwendung paralleler Zustände kann sich ein Statechart gleichzeitig in mehreren Zuständen befinden. Um diesen Systemzustand von einem einzelnen Zustand zu unterscheiden, nennen wir ihn *Zustandskonfiguration*. Eine Zustandskonfiguration ist eine maximale Menge von Zuständen, in denen sich ein Statechart gleichzeitig befinden kann. Jedes Statechart hat implizit einen *Oder*-Zustand r als Ausgangszustand gegeben, der alle weiteren Zustände des Statecharts umfasst. Dieser wird in der graphischen Notation häufig weggelassen. Eine Zustandskonfiguration c enthält nach [HN96] i) den Ausgangszustand r, ii) für jeden *Oder*-Zustand in c *genau* einen der Unterzustände, iii) für jeden *Und*-Zustand in c auch *alle* seine Unterzustände und iv) sie enthält keine weiteren Zustände. Mit r als implizit gegebene Ausgangszustand des Statecharts aus Abbildung 4.3 ist die initiale Zustandskonfiguration: $\{r, s, s_1, s_2, s_{1_1}, s_{2_1}\}$.

Eine *Konfiguration* enthält neben der Zustandskonfiguration die aktiven Ereignisse und die Variablenbelegung der Statechart-Variablen.

Terminierung

Bisher haben wir nur Statecharts betrachtet, die eine unendliche Lebensdauer besitzen und damit unendliche Sequenzen von Konfigurationen beschreiben. Abbildung 4.4 zeigt ein Statechart, das *terminiert*, wenn im Zustand s_2 das Ereignis e_3 erfüllt ist. Die Terminierung wird durch einen sogenannten *Terminierungszustand* markiert, der durch ein doppelt eingekreistes T gekennzeichnet ist. Damit kann die Ausführung von Statecharts gestoppt werden. Sobald der Terminierungszustand erreicht wird, führt das Statechart keine Übergänge mehr aus.

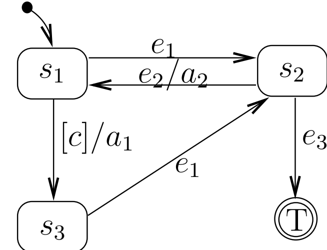


Abbildung 4.4: Terminierung

Aktionen und Bedingungen bezüglich Zuständen

Statecharts erzeugen beim Betreten und Verlassen eines Zustands s sogenannte *entered*- bzw. *exited*-Ereignisse (*entered*(s) bzw. *exited*(s)). Durch die Broadcast-Kommunikation sind diese Ereignisse in parallelen Zuständen sichtbar und Zustände können sich so synchronisieren. Ist ein Zustand s aktiv, gilt das Prädikat *in*(s), das ebenfalls in Übergangsbeschriftungen verwendet werden kann.

Weiterhin gibt es die Möglichkeit, beim Eintreten in einen Zustand eine Aktion auszuführen (*entry*-Aktion). Diese wird so ausgeführt, als ob alle eingehenden Übergänge zusätzlich mit dieser Aktion beschriftet wären. Ebenso können einem Zustand Aktionen zugeordnet werden, die beim Verlassen ausgeführt werden (*exit*-Aktion). Exit-Aktionen verhalten sich so, als ob alle ausgehenden Übergänge zusätzlich mit dieser Aktion beschriftet wären.

Statische Reaktionen

Statecharts bieten mit *statischen Reaktionen* die Möglichkeit, *innerhalb* von Zuständen auf Ereignisse zu reagieren und Aktionen auszuführen, ohne dabei Zustandsübergänge durchzuführen. Statische Reaktionen haben, wie Beschriftungen von Übergängen, die Form $e[c]/a$ (siehe Abbildung 4.5 a)). Sie werden innerhalb eines aktiven Zustands ausgeführt, wenn dieser nicht verlassen wird und die Aktivierungsbedingung erfüllt ist. Eine statische Re-

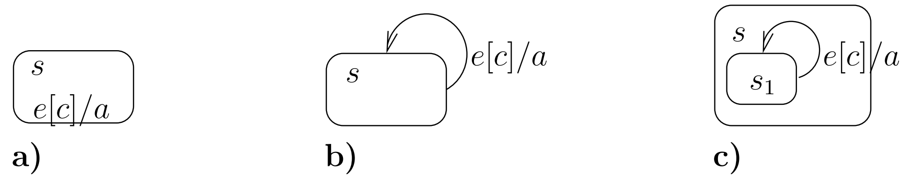


Abbildung 4.5: Statische Reaktionen

aktion kann als Übergang angesehen werden, dessen Ausgangs- und Endzustand derselbe ist, also eine Schleife beschreibt. Ist einem Zustand s eine statische Reaktion zugeordnet, kann die Schleife jedoch nicht von s nach s gehen, wie es in Abbildung 4.5 b) zu sehen

ist. Sonst würde die Schleife mit den weiteren, vom Zustand s ausgehenden Übergängen in Konflikt stehen. Eine statische Reaktion hat nach Definition aber eine niedrigere Priorität. Außerdem würden *exit*- und *enter*-Ereignisse für s erzeugt werden. Dies geschieht bei einer statischen Reaktion nicht. Deshalb ist die semantisch äquivalente Umformung die Einführung eines Unterzustandes s_1, der die Übergangsschleife enthält, wie es Abbildung 4.5 c) zeigt.

4.1.2 Ausführen von Statecharts

Statecharts werden in *Schritten* ausgeführt. Ein Schritt ändert die aktuelle Konfiguration, und durch die auszuführenden Übergänge berechnet sich eine Nachfolgekonfiguration[2]. Durch das Hintereinanderausführen von Schritten wird so eine Sequenz von Konfigurationen erzeugt, die einem Lauf durch das modellierte System entspricht. Eine Konfiguration enthält

- eine Liste aller aktiven Zustände,
- die Belegung der Variablen,
- eine Liste aller Ereignisse, die im vorigen Schritt erzeugt wurden,
- eine Liste der Schedule-Aktionen mit ihrem Ausführungszeitpunkt als Paar $(a, next_a)$ und
- eine Liste der Timeout-Ereignisse mit ihren Auftrittszeitpunkten als Paar $(tm(e, n), next_{tm})$.

Ein Schritt führt Übergänge paralleler Zustände gemeinsam aus, für einen *Oder*-Zustand jedoch maximal einen. Erlaubt die aktuelle Konfiguration verschiedene Übergänge innerhalb eines *Oder*-Zustands, so stehen sie in *Konflikt*. Ein Schritt führt dann indeterministisch einen dieser Übergänge aus.

Ein Statechart-Schritt

Wir beschreiben die Ausführung eines Schritts nach dem Algorithmus von Harel und Naamad [HN96]. Mit dem aktuellen Systemzustand und der aktuellen Zeit t als Eingabe berechnet der Algorithmus den nachfolgenden Systemzustand wie folgt:

1. für alle Schedule-Aktionen $(a, next_a)$:

   wenn $next_a \leq t$ führe a aus und lösche dieses Paar aus der Liste.

2. für jedes Timeout-Ereignis $(tm(e, n), next_{tm})$:

[2]Für Konfiguration, Zustandskonfiguration und Zustand wird oft der Begriff ‘Zustand’ synonym verwendet.

- wenn e erzeugt wurde, setze $next_{tm} := t + n$
- sonst, wenn $next_{tm} \leq t$, erzeuge $tm(e, n)$ und setze $next_{tm} := \infty$.

Diese beiden Schritte erzeugen einen neuen Systemzustand, der Ausgangspunkt für die folgende Berechnung ist.

3. berechne eine maximale konfliktfreie Menge aktivierter Übergänge
   (a) ist die Menge leer, erhält man einen *leeren* Schritt (keine Änderung des Systemzustandes)
   (b) ist die Menge einelementig, wird dieser Schritt ausgeführt
   (c) ansonsten entsteht Indeterminismus und ein Schritt wird gewählt.
4. berechne die aktivierten statischen Reaktionen der aktiven Zuständen, die nicht verlassen werden.
5. lösche die Liste der aktiven Ereignisse.
6. führe die Aktionen der Übergänge und statischen Reaktionen aus.
7. führe alle initialen Übergänge neu betretener Zustände aus.
8. lösche alle Zustände, die verlassen wurden, und füge alle Zustände, die neu betreten werden, zur Liste der aktiven Zustände hinzu.
9. führe die entry- und exit-Aktionen aus.

Aktionen erzeugen dabei neue Ereignisse, die in die Liste generierter Ereignisse aufgenommen werden, und ändern Variablenbelegungen.

Zeitmodelle

STATEMATE unterstützt zwei Modelle, um das Fortschreiten von *Zeit* zu beschreiben. Im *synchronen* Zeitmodell geht man davon aus, dass nach jedem einzelnen Schritt Zeit vergeht.

Im *asynchronen* Zeitmodell können mehrere Statechart-Schritte innerhalb eines Zeitschrittes ausgeführt werden. Ein Statechart-Schritt wird dann als *Mikro*-Schritt, ein Zeitschritt als *Makro*-Schritt bezeichnet. Es werden so lange Mikro-Schritte durchgeführt, bis das Statechart eine *stabile* Konfiguration erreicht hat, also kein Übergang mehr ausgeführt werden kann. Danach folgt ein Makro-Schritt.

Wenn Zeit vergeht, werden auch Änderungen der Umgebung betrachtet. Um auf die Umgebung zu reagieren und die Zeit anzupassen, wird im synchronen Zeitmodell bei jedem Schritt, im asynchronen Modell bei jedem Makro-Schritt, folgende Stufe dem Schritt-Algorithmus vorangestellt:

0. füge die externen Ereignisse zur Liste der Ereignisse hinzu, übernehme externe Variablenbelegungen und erhöhe die aktuelle Zeit auf $t := t + 1$.

In beiden Zeitmodellen geht man davon aus, dass die Übergänge keine Zeit benötigen und das System „schnell genug“ reagiert. Deshalb müssen während der Übergänge keine Umgebungsänderungen berücksichtigt werden.

4.2 Formalisierung von Statecharts

Wir formalisieren die Semantik von STATEMATE-Statecharts analog zur operationellen Semantik von Damm et al. [DJHP98]. Sie formalisiert die *Makro-Schritt*-Semantik von Statecharts, die der *perfect synchrony hypothesis* aus [BG92] genügt: innerhalb eines Makro-Schritts ändern sich die Eingabevariablen nicht, es werden keine externen Ereignisse erzeugt und es vergeht keine Zeit. Die berechneten Ausgaben werden am Ende des Makro-Schritts der Umgebung sichtbar gemacht. Wie Damm et al. behandeln wir Ereignisse als (spezielle) boolesche Variablen und fassen die Übergangsbedingung $e[c']$ zu $c := e \wedge c'$ zusammen.

Allerdings führen wir Statecharts explizit in Mikro-Schritten aus und kennzeichnen einen Makro-Schritt durch ein *tick*-Ereignis, das ein Statechart in einem Makro-Schritt generiert. Die Semantik von Statecharts basiert damit auf Folgen von Mikro- und Makro-Schritten. Möchten wir über Ereignisse zu Makro-Schritten sprechen, verwenden wir das *tick*-Ereignis als Referenz. Das *tick*-Ereignis unterscheidet sich von booleschen *stable*-Variablen[3] anderer Ansätze [DJHP98, CAB$^+$01] dadurch, dass es nicht nur zwischen Mikro- und Makro-Schritten unterscheidet, sondern ein Ereignis ist, das auf den Übergängen verwendet werden kann. Dadurch können wir auf die explizite Unterstützung der *Timer*- und *Schedule*-Konstrukte verzichten und beide Konstrukte mit Hilfe des *tick*-Ereignisses modellieren (siehe z. B. statische Reaktion in Abbildung 4.6 und Abschnitt 6.2.2).

Bevor wir die Semantik von Statecharts angeben, definieren wir die zugrundeliegende abstrakte Syntax.

4.2.1 Abstrakte Syntax

Die Definition der abstrakten Syntax veranschaulichen wir im Folgenden an einem kleinen Beispiel einer Zeitschaltuhr, die in Abbildung 4.6 zu sehen ist. Die Zeitschaltuhr sorgt nach dem Einschalten des Lichts durch das Umgebungsereignis *press* dafür, dass ein Timer gesetzt wird, der die Brenndauer des Lichts auf 6 Zeiteinheiten beschränkt. Eine Zeiteinheit entspricht in der asynchronen Semantik von Statecharts einem Makro-Schritt. Die statische Reaktion *cnt*, die in jedem Makro-Schritt – durch das Ereignis *tick* ausgelöst – die Variable x um eins erhöht, garantiert die maximale Brenndauer. Sobald x größer 5 ist, schaltet der Timer das Licht aus. Um im Folgenden das Statechart vom Zustand *System* unterscheiden zu können, nennen wir das Statechart *SYS*.

[3] *stable* steht für stabile Konfiguration, es ist kein Mikro-Schritt mehr möglich, ein Makro-Schritt folgt und liefert neue Eingaben.

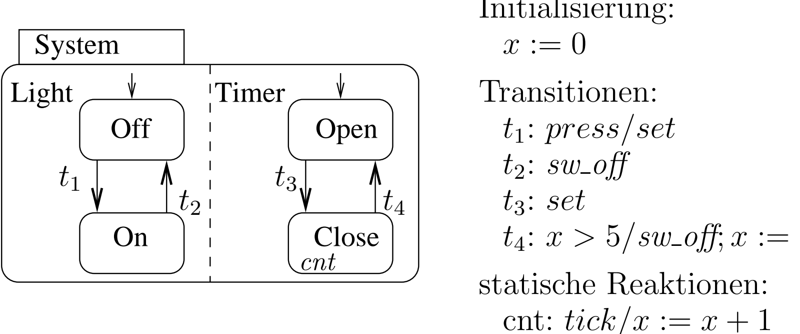


Abbildung 4.6: Spezifikation einer Zeitschaltuhr

Zustände

Ein Statechart *SC* besteht aus einer Menge hierarchisch angeordneter, getypter Zustände *states*(*SC*), die einem Ausgangszustand *root* = *root*(*SC*) untergeordnet sind. Wir verlangen, dass *root*(*SC*) ein *Oder*-Zustand ist und fügen einem Statechart, dessen Ausgangszustand kein *Oder*-Zustand ist, implizit den *Oder*-Zustand *root* als Ausgangszustand hinzu, der alle anderen Zustände umfasst. Die Hierarchie der Zustände bildet damit einen Zustandsbaum, dessen Wurzel nach Definition ein *Oder*-Zustand ist. D. h. das Beispiel der Zeitschaltuhr definiert implizit den Ausgangszustand *root*, der dem *Und*-Zustand *System* übergeordnet ist. Damit sind die Zustände des Statecharts *SYS*

$$\mathit{states}(\mathit{SYS}) = \{\mathit{root}, \mathit{System}, \mathit{Light}, \mathit{Off}, \mathit{On}, \mathit{Timer}, \mathit{Open}, \mathit{Close}\}.$$

Zustände in *states*(*SC*) sind entweder *Basis*-, *Oder*-, *Und*- oder *Terminierungs*-Zustände. Der Typ eines Zustands $s \in \mathit{states}(\mathit{SC})$ wird durch die Funktion $\mathit{mode} : \mathit{states}(\mathit{SC}) \rightarrow \{\mathit{BASIC}, \mathit{OR}, \mathit{AND}, \mathit{TERM}\}$ bestimmt. $\mathit{term}(\mathit{SC}) := \{s \mid s \in \mathit{states}(\mathit{SC}) \text{ und } \mathit{mode}(s) = \mathit{TERM}\}$ ist die Menge der Terminierungszustände eines Statecharts. Ein *Terminierungs*-Zustand tritt im obigen Beispiel nicht auf, wurde aber in Abschnitt 4.1.1 besprochen. *Off* ist ein *Basis*-, *Light* ein *Oder*- und *System* ein *Und*-Zustand.

Das Hierarchiekonzept führt eine Relation zwischen Zuständen ein. Direkte Unterzustände eines Zustands s werden durch die Relation

$$\mathit{childs} : \mathit{states}(\mathit{SC}) \rightarrow \wp(\mathit{states}(\mathit{SC}))$$

definiert. Alle Zustände in *states*(*SC*) sind von *root* aus über die *childs*-Relation erreichbar. Jeder *Oder*-Zustand s besitzt einen Zustand $\mathit{default}(s) \in \mathit{childs}(s)$, der initial aktiv ist. Für alle $s' \in \mathit{childs}(s)$ ist s ein Vorfahre und wird durch $s = \mathit{father}(s')$ berechnet. Dadurch ergibt sich eine partielle Ordnung $\leq$ auf den Zuständen, wobei $\mathit{root} \leq s$ für alle $s \in \mathit{states}(\mathit{SC})$ und $s < s'$ wenn $s \leq s'$ und $s \neq s'$ gilt. Die Tiefe im Statechart *depth*(s) eines Zustands s entspricht der Tiefe im Zustandsbaum. In der Zeitschaltuhr hat der *Und*-

Zustand *System* zwei Unterzustände $childs(System) = \{Light, Timer\}$. Damit ist $Light \in childs^*(root)$[4], $root < System < Light$ und die Tiefe $depth(Light)$ ist 2.

Der kleinste gemeinsame Vorfahre $lca : \wp(states(SC)) \rightarrow states(SC)$ einer nichtleeren Zustandsmenge S ist der Zustand i) größter Tiefe, der ii) alle Zustände in S umfasst. Für alle $s \in S$ und alle $s' \in states(SC)$ gilt:

i) wenn $s' \leq S$ dann $s' \leq lca(S)$ und

ii) $lca(S) \leq s$

Analog ist der kleinste gemeinsame *Oder*-Vorfahre $lca_+(S)$ als *Oder*-Zustand größter Tiefe definiert, der alle Zustände in S umfasst. Da *root* ein *Oder*-Vorfahre aller Zustände ist, existiert immer $lca(S)$ und $lca_+(S)$. Der kleinste gemeinsame Vorfahre der Zustände *On* und *Timer*, $lca(\{On, Timer\})$, ist *System* und der kleinste gemeinsame *Oder*-Vorfahre ist der Ausgangszustand $root = lca_+(\{On, Timer\})$.

Die Zustände s und s' sind orthogonal ($s \perp s'$), wenn sie sich in parallelen Zuständen befinden, d. h. $s \notin childs^*(s')$, $s' \notin childs^*(s)$ und der kleinste gemeinsame Vorfahre $lca(\{s, s'\})$ ein *Und*-Zustand ist. Eine Menge von Zuständen S ist orthogonal ($\perp(S)$), wenn alle Zustände paarweise orthogonal sind. Eine Menge von Zuständen bezeichnet man als konsistent ($\downarrow(S)$) wenn für jedes Paar von Zuständen s und s' entweder $s \in childs^*(s')$, $s' \in childs^*(s)$ oder $s \perp s'$ gilt. Konsistente Mengen beschreiben Zustände, die gemeinsam aktiv sein können. Die beide Zustände *On* und *Timer* liegen parallel zueinander und sind deshalb orthogonal, d. h. es gilt $\perp(\{On, Timer\})$. Die Menge $\{On, Light, Timer\}$ ist nicht orthogonal, da $On \in childs(Light)$. Die Menge ist aber konsistent, $\downarrow(On, Light, Timer)$. Dagegen ist $\{On, Off, Timer\}$ nicht konsistent, da *On* und *Off* Nachfahren desselben *Oder*-Zustands *Light* sind.

Für $S \subseteq states(SC)$ und $\downarrow(S)$ ist die vervollständigte Zustandskonfiguration $C := decompl(S)$[5] die kleinste Menge, so dass

i) $S \subseteq C$,

ii) $s \in C$ und $s \neq root(SC)$, dann ist auch $father(s) \in C$,

iii) $s \in C$, $mode(s) = OR$ und $childs^+(s)$[6] $\cap S = \emptyset$, dann ist $default(s) \in C$ und

iv) $s \in C$, $mode(s) = AND$, dann auch $childs(s) \subseteq C$.

Eine vervollständigte Zustandskonfiguration der Zeitschaltuhr ist

$$decompl(\{On, Timer\}) = \{root, System, Light, On, Timer, Open\},$$

da *root*, *System* und *Light* Vorfahren von *On* sind und *Open* der initiale Nachfolger von *Timer* ist. Die Menge aller konsistenten, vollständigen Zustandskonfigurationen berechnet

$$conf(SC) := \{S \mid S \subseteq states(SC), \downarrow(S) \text{ und } S = decompl(S)\}.$$

[4] * berechnet die reflexiv transitive Hülle einer Funktion.

[5] *decompl* steht für *default completion*.

[6] $^+$ berechnet die transitive Hülle einer Funktion.

Übergänge und statische Reaktionen

Die Menge $\mathit{trans}(SC)$ enthält alle Übergänge eines Statecharts SC. Für einen Übergang $t \in \mathit{trans}(SC)$ liefert $\mathit{source}(t)$ den Ausgangszustand und $\mathit{target}(t)$ den Endzustand. Ein Übergang t ist mit einem Bedingungs/Aktions-Paar c/α, beschriftet. $\mathit{guard}(t)$ selektiert die Bedingung c, $\mathit{action}(t)$ die Aktion α. Der Übergang t_4 geht von $\mathit{source}(t_4) = \mathit{Close}$ nach $\mathit{target}(t_4) = \mathit{Open}$, hat die Aktivierungsbedingung $\mathit{guard}(t_4) = x > 5$ und die Übergangsaktion $\mathit{action}(t_4) = \mathit{sw_off}; x := 0$, die das Ereignis $\mathit{sw_off}$ erzeugt und x auf Null setzt.

Für einen Übergang t ist seine Region der kleinste gemeinsame *Oder*-Vorfahre des Ausgangs- und Endzustandes und wird durch

$$\mathit{scope}(t) := \mathit{lca}_+(\{\mathit{source}(t), \mathit{target}(t)\})$$

berechnet. Zwei Übergänge t_1 und t_2 sind konsistent oder konfliktfrei (bezeichnet durch $\downarrow(t_1, t_2)$), d. h. sie können gemeinsam ausgeführt werden, wenn sie in orthogonalen Regionen liegen, d. h. $\downarrow(t_1, t_2) :\Leftrightarrow \mathit{scope}(t_1) \perp \mathit{scope}(t_2)$. Eine Menge von Übergängen T heißt konsistent oder konfliktfrei $\downarrow(T)$, wenn die Übergänge paarweise konsistent sind. Die Übergänge t_1 und t_4 der Zeitschaltuhr sind konsistent $\downarrow(t_1, t_4)$, da ihre Regionen $\mathit{scope}(t_1) = \mathit{Light}$ und $\mathit{scope}(t_4) = \mathit{Timer}$ orthogonal sind. Die Übergänge t_1 und t_2 sind dagegen nicht konsistent, da beide in der Region von *Light* liegen.

Statische Reaktionen $\mathit{sreactions}(SC)$ sind eine Menge von Paaren $\mathit{sr} = (c, \alpha)$, die an einen Zustand s gebunden sind. Die Funktion

$$\mathit{sreact} : \mathit{states}(SC) \to \wp(\mathit{sreactions}(SC))$$

berechnet die Menge der statischen Reaktionen eines Zustands. Für (c, α) ist ebenfalls $\mathit{guard}((c, \alpha)) = c$ und $\mathit{action}((c, \alpha)) = \alpha$ definiert. Die Menge aller Aktionen eines Statecharts SC ist

$$\mathit{actions}(SC) := \bigcup_{t \in \mathit{trans}(SC)} \{\mathit{action}(t)\} \cup \bigcup_{\mathit{sr} \in \mathit{sreactions}(SC)} \{\mathit{action}(\mathit{sr})\}.$$

Die Zeitschaltuhr hat die statische Reaktion $\mathit{sreact}(\mathit{Close}) = \{\mathit{cnt}\}$ mit $\mathit{guard}(\mathit{cnt}) = \mathit{tick}$ und $\mathit{action}(\mathit{cnt})$ ist $x := x + 1$.

Variablen

Wir interpretieren Zustände als boolesche Variablen. Ein Statechart SC definiert neben den Zustandsvariablen $\mathit{states}(SC)$ eine davon disjunkte Menge von Ereignissen $\mathit{events}(SC)$, die ebenfalls boolesche Variablen sind, und Datenvariablen $\mathit{vars}(SC)$. Wir unterteilen Ereignisse und Variablen in lokale Ereignisse und Variablen und Umgebungsereignisse und -variablen ($\mathit{events}_{loc}(SC)$ bzw. $\mathit{vars}_{loc}(SC)$ und $\mathit{events}_{env}(SC)$ bzw. $\mathit{vars}_{env}(SC)$). Zusätzlich gibt es noch implizit definierte Ereignisse $\mathit{events}_{imp}(SC)$. Es gilt:

$$\begin{aligned} \mathit{events}(SC) &:= \mathit{events}_{loc}(SC) \cup \mathit{events}_{env}(SC) \cup \mathit{events}_{imp}(SC) \\ \mathit{vars}(SC) &:= \mathit{vars}_{loc}(SC) \cup \mathit{vars}_{env}(SC) \end{aligned}$$

Die Mengen $events_{loc}(SC)$, $events_{env}(SC)$, $events_{imp}(SC)$, $vars_{loc}(SC)$ und $vars_{env}(SC)$ sind paarweise disjunkt. Das implizite Ereignis

tick	kennzeichnet einen Makro-Schritt,
$en(s)$	kennzeichnet das Betreten des Zustands s und
$ex(s)$	kennzeichnet das Verlassen des Zustands s.

Diese werden u. a. in den statischen Reaktionen dazu verwendet, um *entry*- und *exit*-Aktionen auszulösen (siehe Abschnitt 4.1.1). Auf das Prädikat $in(s)$, das immer dann wahr ist, wenn der Zustand s aktiv ist, kann hier verzichtet werden, da die (boolesche) Zustandsvariable s über der aktuellen Belegung genau dann wahr ist, wenn s aktiv ist.

Variablen $v \in vars(SC)$ können mit Werten aus ihrem Wertebereich initialisiert werden. $default : v \to \mathcal{D}$ liefert für initialisierte Variablen den Initialwert und einen beliebigen Wert aus dem Wertebereich $\mathcal{D}$ von v, wenn v nicht initialisiert wird. Die Menge aller Variablen $variables(SC)$ eines Statecharts fasst Ereignisse, Daten- und Zustandsvariablen zusammen.

$$variables(SC) := states(SC) \cup events(SC) \cup vars(SC)$$

Das Beispiel der Zeitschaltuhr definiert neben den Zustandsvariablen $states(SYS)$ lokale Ereignisse $events_{loc}(SYS) = \{sw_off, set\}$, das Umgebungsereignis $events_{env}(SYS) = \{press\}$ und das implizit definierte Ereignis $events_{imp}(SYS) = \{tick\}$. Implizite Ereignisse, die das Betreten und Verlassen von Zuständen kennzeichnen, werden hier nicht benötigt und deshalb nicht aufgeführt. Die lokale Variable $vars_{loc}(SYS) = \{x\}$ wird mit $default(x) = 0$ initialisiert. Dies ist in der graphischen Darstellung jedoch nicht zu sehen.

Aktivierungsbedingungen und Aktionen

Wie schon bemerkt, werden Ereignisse e als boolesche Variablen betrachtet, die speziell behandelt werden (sie sind immer nur einen Schritt aktiv). Im Gegensatz zu Damm et al. geben wir keine Daten, Funktionen und Prädikate vor, sondern beschreiben sie durch eine algebraische Spezifikation, die dem Statechart SC zugrunde liegt. Die Aktivierungsbedingungen $guard(t)$ bzw. $guard(sr)$ von Übergängen t und statischen Reaktionen sr sind also boolesche Ausdrücke, die über der Signatur der Algebra und den Variablen $variables(SC)$ gebildet werden. Damit können wir Statecharts über beliebigen algebraischen Spezifikationen definieren und sind nicht auf die vordefinierten Datentypen und Operatoren von STATEMATE eingeschränkt. $ev(guard(t)) := \{e \mid e \in vars(guard(t)) \cap events(SC)\}$ ist die Menge alle Ereignisse einer Aktivierungsbedingung, wobei $vars(expr)$ alle Variablen eines Ausdrucks $expr$ wie üblich berechnet. Ein Überblick über algebraische Spezifikationen findet sich in [Wir90].

Analog erweitern wir die Menge der STATEMATE-Aktionen, die bei einem Zustandsübergang ausgeführt werden, auf sequentielle Programme α. Die formale Semantik von sequentiellen Programmen findet sich im Anhang E. Um während der Statechart-Schrittausführung auf Variablenwerte zurückgreifen zu können, die vor Beginn des Schrittes galten, benötigen

wir für jede Statechart-Variable eine Hilfsvariable. Die Schrittausführung ändert zunächst nur diese Hilfsvariablen. Wir definieren die Mengen der Hilfsvariablen

$$\widetilde{variables}(SC) := \{\tilde{x} \mid x \in variables(SC)\}.$$

Analog bezeichnet $\widetilde{events}(SC)$, $\widetilde{vars}(SC)$, ... die Menge der Hilfsvariablen für Ereignisse, Datenvariablen, ...

Damit wir nun während der Ausführung keine Variablenwerte überschreiben, vereinbaren wir, dass Programme nur die Hilfsvariablen $\widetilde{variables}(SC)$ schreiben. Die Programme sind damit über den Variablen $variables(SC) \cup \widetilde{variables}(SC)$ und der zugrundeliegenden algebraischen Spezifikation definiert. Ereignisse $e \in events(SC)$ werden durch die Zuweisung $\tilde{e}$:= true erzeugt. Ein Programm α kann sowohl lesend als auch schreibend auf Variablen eines Statecharts SC zugreifen. Die Menge der gelesenen Variablen berechnet $read(\alpha)$ und der geschriebenen Variablen $write(\alpha)$ (formale Definition siehe Anhang E).

Wohlgeformte Statecharts

Ein Statechart SC ist wohlgeformt, wenn i) *root* keine aus- und eingehenden Übergänge besitzt, ii) Übergänge nicht über parallele Regionen hinweggehen, iii) Aktionen auf keine Ereignisse lesend und iv) nur auf die Hilfsvariablen $\widetilde{vars}(SC) \cup \widetilde{events}(SC)$ schreibend zugreifen. Es dürfen in den Aktionen also keine Zustandsvariablen geschrieben werden. Ein Statecharts ist wohlgeformt, wenn

i) $root \notin \bigcup_{t \in trans(SC)} \{source(t), target(t)\}$,

ii) $\forall\, t \in trans(SC).\ source(t) \not\perp target(t)$,

iii) $(events(SC) \cup \widetilde{events}(SC)) \cap \bigcup_{\alpha \in actions(SC)} read(\alpha) = \emptyset$ und

iv) $\bigcup_{\alpha \in actions(SC)} write(\alpha) \subseteq (\widetilde{vars}(SC) \cup \widetilde{events}(SC))$.

Im Folgenden betrachten wir nur wohlgeformte Statecharts.

4.2.2 Semantik

Die Semantik von Statecharts ist eine Menge von endlichen und unendlichen Traces oder Intervallen. Ein Trace ist eine Folge von Belegungen $\sigma : V \to \mathcal{D}$, die Variablen V in den entsprechenden (typisierten) Wertebereich $\mathcal{D}$ abbilden. Die Menge aller Belegungen über den Variablen V nennen wir $\sum(V)$. Für eine Untermenge $V' \subseteq V$ von Variablen ist die Einschränkung $\sigma\mid_{V'}$ einer Belegung $\sigma \in \sum(V)$ auf V' eine Belegung aus $\sum(V')$ gegeben durch

$$\sigma\mid_{V'} : V' \to \mathcal{D} : v \mapsto \sigma(v).$$

Für boolesche Variablen $b \in V$ kürzen wir $\sigma(b) =$ true mit $\sigma(b)$ ab. Im Folgenden beschreiben wir, welche Variablenmenge, Ausgangsbelegung und Relation zwischen Belegungen ein Statechart definiert und welche Belegungen konsistent sind.

Konsistente Belegung

Eine Belegung σ nennen wir konsistent bezüglich eines Statecharts SC, wenn sie einer möglichen Konfiguration des Statecharts entspricht. In einer Konfiguration ist genau ein Unterzustand eines aktiven *Oder*-Zustands aktiv und alle Unterzustände eines aktiven *Und*-Zustands. Weiterhin werden beim Ausführen von Statecharts nur dann Umgebungsereignisse erzeugt, wenn keine Mikro-Schritte mehr möglich sind, die lokale Ereignisse erzeugen. Da vor jedem Statechart-Schritt die lokalen Ereignisse zurückgesetzt werden, können bei der Ausführung von Statecharts, startend von einer initialen Konfiguration, in einem Makro-Schritt niemals lokale Ereignisse aktiv sein. Eine Belegung heißt also konsistent, wenn i) die Menge der aktiven Zustände in σ konsistent ist und einer vollständigen Zustandskonfiguration entspricht und ii) in einem durch *tick* gekennzeichneten Makro-Schritt keine lokalen Ereignisse aktiv sind. Für $S := \{s \mid s \in \mathit{states}(SC) \text{ und } \sigma(s)\}$ definieren wir die Konsistenz einer Belegung σ:

$$
\begin{array}{ll}
\downarrow(\sigma, SC) :\Leftrightarrow & \\
\quad \downarrow(S) \text{ und } S = \mathit{decompl}(S) \text{ und} & \text{(i)} \\
\quad \text{entweder } \{ev \mid ev \in \mathit{events}_{loc}(SC) \cup \mathit{events}_{imp}(SC) \text{ und } \sigma(ev)\} = \emptyset & \multirow{2}{*}{\text{(ii)}} \\
\quad \text{oder } \sigma(\mathit{tick}) = \text{false} &
\end{array}
$$

Initiale Konfiguration

Die *initiale Konfiguration* oder auch *Ausgangskonfiguration* beschreibt die Belegung der Variablen beim Systemstart. Das Statechart befindet sich in den Ausgangszuständen und keine Ereignisse sind aktiv. Die Variablen haben Werte entsprechend ihrer Initialisierung bzw. einen beliebigen Wert aus ihrem Wertebereich, wenn sie nicht initialisiert werden. Die Menge der Ausgangsbelegungen berechnet $\mathit{init}(SC)$ und eine Belegung σ ist darin enthalten, $\sigma \in \mathit{init}(SC)$, genau dann wenn

$$
\sigma(v) = \begin{cases}
\text{true}, & \text{wenn } v \in \mathit{decompl}(\mathit{root}(SC)) \\
\text{false}, & \text{wenn } v \in \mathit{states}(SC) \setminus \mathit{decompl}(\mathit{root}(SC)) \\
\text{false}, & \text{wenn } v \in \mathit{events}(SC) \\
\mathit{default}(v), & \text{wenn } v \in \mathit{vars}(SC).
\end{cases}
$$

Aktivierungsbedingungen und Aktionen

Die Aktivierungsbedingungen $\mathit{guard}(t)$ von Übergängen t sind boolesche Ausdrücke und werden über der zugrundeliegenden Algebra $\mathcal{A}$ und der aktuellen Belegung σ ausgewertet. Die Semantik von $[\![\mathit{guard}(t)]\!]_{\mathcal{A},\sigma}$ über einer algebraischen Spezifikation ist wie üblich definiert [Wir90]. Für $\mathcal{A}, \sigma \models \mathit{guard}(t)$ schreiben wir auch $\sigma \models \mathit{guard}(t)$, wenn die zugrundeliegende Algebra $\mathcal{A}$ für die weitere Betrachtung nicht wichtig ist.

Die Aktionen $\alpha \in \mathit{actions}(SC)$ sind sequentielle Programme, deren Semantik $[\![\alpha]\!]_{\mathcal{A},\sigma}$ im Anhang E definiert ist. An dieser Stelle muss auf einen Unterschied zur STATEMATE-Semantik von Aktionen aufmerksam gemacht werden. In STATEMATE wird ';' als Parallelkomposition verstanden, d. h. die Variablen beziehen sich immer auf den Wert *vor* Beginn

der Schrittausführung. So ist y nach Ausführen der Aktion $x := 5; y := x+1$ um eins größer als der Wert von x *vor* der Aktion. y ist also nicht notwendigerweise 6. Im Gegensatz dazu kennzeichnet ';' in sequentiellen Programmen eine sequentielle Programmkomposition und der Wert von y nach Ausführen der obigen Aktion ist 6, unabhängig von der Belegung von x vor der Ausführung. Für die sequentielle Semantik gibt es in STATEMATE sogenannte Kontextvariablen $\$x$, die sich auf den Wert der Variablen beziehen, der sich aus dem Kontext (vorhergehende Aktionen) ergibt. Die STATEMATE-Aktion $\$x := 5; y := \$x + 1$ berechnet also dasselbe Resultat wie das sequentielle Programm $x := 5; y := x + 1$. Kontextvariablen sind in Schleifen notwendig, um Variablen in Schleifenrümpfen überhaupt ändern zu können.

Unserer Erfahrung nach muss man eher selten auf Variablenwerte zurückgreifen, die vor der Schrittausführung galten. Um näher an der üblichen Programmsemantik zu bleiben, verwenden wir Variablen deshalb wie Kontextvariablen. Wir haben Aktionen über den Hilfsvariablen $\widetilde{\mathit{variables}}(SC)$ definiert und weisen vor Ausführung aller Aktionen den Variablen $\tilde{x}$ den aktuellen Wert von x zu. Das Programm $\tilde{x} := 5; \tilde{y} := \tilde{x}+1$ entspricht dann der STATEMATE-Aktion $\$x := 5; y := \$x+1$. Deshalb ändern in wohlgeformten Statecharts Aktionen die Variablen $x \in \mathit{variables}(SC)$ nicht, sondern sie enthalten immer den Variablenwert von x *vor* Ausführung der Aktionen. Die STATEMATE-Aktion $x := 5; y := x + 1$ erhalten wir deshalb durch das Programm $\tilde{x} := 5; \tilde{y} := x + 1$. Wenn wir STATEMATE-Aktionen in Programme übersetzen, ersetzen wir alle Kontextvariablen und in schreibenden Zugriffen die Variablen $x \in \mathit{variables}(SC)$ durch die entsprechende Hilfsvariable $\tilde{x}$.

Berechnung maximaler Mengen konfliktfreier Übergänge

Ist in einem Zustand ein Übergang möglich, so muss er ausgeführt werden. In parallelen Zuständen können mehrere Übergänge gleichzeitig ausgeführt werden. Es ist aber nicht erlaubt, mehrere Übergänge eines *Oder*-Zustands gleichzeitig auszuführen. Mengen von Übergängen, die diese Bedingungen erfüllen, nennen wir *maximale Menge konfliktfreier Übergänge*.

Ein Übergang $t \in \mathit{trans}(SC)$ kann ausgeführt werden (es gilt $\mathit{en}(t)$), wenn der Ausgangszustand aktiv ist und die Aktivierungsbedingung zu true evaluiert. Auf Umgebungsereignisse darf zusätzlich nur zu Beginn eines Makro-Schrittes reagiert werden. Deshalb fordern wir für Umgebungsereignisse in $\mathit{guard}(t)$, dass zusätzlich das *tick*-Ereignis gelten muss, d. h. wir erweitern die Definition von $\mathit{guard}(t)$ zu

$$\mathit{guard}(c/\alpha) := \begin{cases} c \wedge \mathit{tick}, & \text{wenn } \mathit{ev}(c) \cap \mathit{events}_{\mathit{env}}(SC) \neq \emptyset \\ c, & \text{sonst.} \end{cases}$$

Ein Übergang ist dann aktiviert, d. h. es gilt $\mathit{en}(t)$, wenn der Ausgangszustand aktiv ist und die Aktivierungsbedingung gilt.

$$\sigma \models \mathit{en}(t) :\Leftrightarrow \sigma(\mathit{source}(t)) \text{ und } \sigma \models \mathit{guard}(t)$$

Relativ zu einem Zustand s und einer Belegung σ berechnet $\mathit{steps}(\sigma, s) \in \wp(\wp(\mathit{trans}(SC)))$ nach Damm et al. [DJHP98] die maximale Menge konfliktfreier Übergänge, und ist wie folgt definiert:

1. $\mathit{mode}(s) \in \{\mathit{BASIC}, \mathit{TERM}\}$:

   $\mathit{steps}(\sigma, s) := \emptyset$

2. $\mathit{mode}(s) = \mathit{AND}$:

   Sei $\{s_1, \ldots, s_n\} = \mathit{childs}(s)$:

   $\mathit{steps}(\sigma, s) := \{st_1 \cup \ldots \cup st_n | st_i \in \mathit{steps}(\sigma, s_i)\}$

3. $\mathit{mode}(s) = \mathit{OR}$:

   Sei $T = \{t_1, \ldots, t_k\} = \{t | \mathit{scope}(t) = s \text{ und } \sigma \models \mathit{en}(t)\}$

   (a) $T = \emptyset$:
       s' ist der (einzige) Nachfolger von s in σ[7]:
       $\mathit{steps}(\sigma, s) := \mathit{steps}(\sigma, s')$

   (b) $T \neq \emptyset$:
       die Übergänge in T haben dieselbe Priorität und stehen deshalb alle in Konflikt zueinander.
       $\mathit{steps}(\sigma, s) := \{\{t_1\}, \ldots, \{t_k\}\}$

Für konsistente Belegungen σ berechnet $\mathit{steps}(\sigma, s)$ konsistente Übergangsmengen. Es gilt:

$$\text{wenn } \downarrow(\sigma, SC) \text{ und } st \in \mathit{steps}(\sigma, s) \text{ dann } \downarrow(st).$$

Die Konsistenz von st folgt induktiv über die Tiefe von s, da in den Berechnungsschritten 1-3 Übergänge nur konsistent zur Lösungsmenge hinzugefügt werden.

Bezüglich eines Statecharts SC berechnet, ausgehend von einer Belegung σ, $\mathit{Steps}(\sigma) = \mathit{steps}(\sigma, \mathit{root}(SC))$ alle möglichen Übergangsmengen, die ausgeführt werden können. Für $st \in \mathit{Steps}(\sigma)$ berechnen die Funktionen $\mathit{exited}(st)$, $\mathit{entered}(st)$ und $\mathit{active}(st)$ je eine Menge von Zuständen, die beim Ausführen von st verlassen werden, betreten werden, bzw. vor und nach dem Ausführen von st aktiv sind durch

$$\begin{array}{lcl} \mathit{exited}(st) & := & \{s \mid \exists\, t.\ t \in st \text{ mit } \mathit{scope}(t) < s\} \cap \{s \mid \sigma(s)\}, \\ \mathit{active}(st) & := & \{s \mid \sigma(s)\} \backslash \mathit{exited}(st) \text{ und} \\ \mathit{entered}(st) & := & \mathit{decompl}(\mathit{active}(st) \cup \bigcup_{t \in st} \{\mathit{target}(t)\}) \backslash \mathit{active}(st). \end{array}$$

Nun können alle Aktionen eines Schritts $\mathit{action_set}(st, \sigma)$ berechnet werden. Neben den Aktionen der selektierten Übergänge werden in einem Schritt die Aktionen der statischen

[7] $\{s'\} = S'$ wobei $S' := \{\hat{s} \mid \hat{s} \in \mathit{childs}(s) \text{ und } \sigma(\hat{s})\}$. Die Menge S' ist in einer konsistenten Belegung σ immer einelementig.

Reaktionen ausgeführt, die in aktiven Zuständen $s \in \mathit{active}(st)$ sind und deren Aktivierungsbedingung gilt. Analog zu Übergängen können wir ein Prädikat $en(sr)$ für statische Reaktionen definieren.

$$\sigma \models en(sr) :\Leftrightarrow \exists\ s.s \in \mathit{active}(st) \text{ und } sr \in \mathit{sreact}(s) \text{ und } \sigma \models \mathit{guard}(sr)$$

Die Menge aller Aktionen eines Schritts berechnet sich dann durch

$$\mathit{action_set}(st, \sigma) := \bigcup_{t \in st}\{\mathit{action}(t)\} \cup \{\mathit{action}(sr) \mid \sigma \models en(sr)\}.$$

Ausführen eines Schrittes

Ein Statechart-Schritt ist der Übergang einer Belegung σ in die Nachfolgebelegung σ' und wird relativ zu einem Statechart SC und einer Algebra $\mathcal{A}$ durch die Übergangsrelation $\rho_{step_{SC,\mathcal{A}}}$ beschrieben. Ist das zugrundeliegende Statechart und die Algebra gegeben, schreiben wir ρ_{step} statt $\rho_{step_{SC,\mathcal{A}}}$. Ein Schritt besteht entweder aus einem Mikro-Schritt ρ_{micro} oder einem Makro-Schritt ρ_{macro}, d. h.

$$\rho_{step} := \rho_{micro} \cup \rho_{macro}.$$

Ausgehend von der aktuellen Belegung σ berechnen wir die Menge der möglichen Schritte $\mathit{Steps}(\sigma)$. Ist diese Menge leer, findet ein Makro-Schritt statt. Sonst wählen wir (nichtdeterministisch) eine Übergangsmenge $st \in \mathit{Steps}(\sigma)$ und führen einen Mikro-Schritt aus.

Mikro-Schritt Wir können einen Mikro-Schritt ausführen, wenn aus der aktuellen Konfiguration heraus Schritte möglich sind und das Statechart nicht terminiert wurde. Zuerst kopieren wir die Werte der Ereignisse und Datenvariablen x in die Hilfsvariablen $\tilde{x} \in \widetilde{\mathit{events}}(SC) \cup \widetilde{\mathit{vars}}(SC)$, damit diese die aktuellen Werte enthalten, und führen die Schrittberechnung auf diesen Kopien aus. Damit werden während der Schrittberechnung die Werte der Statechart-Variable nicht überschrieben und wir können auf die Werte vor Beginn der Schrittberechnung zurückgreifen. Bevor wir dann die Aktionen des Schritts ausführen, setzen wir die lokalen Ereignisse, da sie immer nur einen Schritt aktiv sind, und das *tick*-Ereignis für den nächsten Schritt zurück. Umgebungsereignisse setzen wir nur nach einem Makro-Schritt zurück (wenn also $\sigma(\mathit{tick})$ gilt). Damit sind Umgebungsereignisse, die während einer Kette von Mikro-Schritten berechnet werden, bis zum Ende eines Makro-Schrittes sichtbar (sie lösen aber nur dann einen Übergang aus, wenn *tick* den ersten Mikro-Schritt nach einem Makro-Schritt kennzeichnet). Damit können wir alle Ausgabeereignisse, die während der Mikro-Schritte berechnet werden, am Ende eines Makro-Schrittes beobachten. Das Zurücksetzen der Ereignisse beeinflusst die Ausführung der Aktionen α nicht, denn in den Aktionen wird auf Ereignisse nur schreibend zugegriffen. Nach dem Ausführen der Aktionen müssen noch die abgeleiteten Ereignisse aktualisiert, die verlassenen Zustände aus der aktuellen Konfiguration gelöscht, die neu betretenen Zustände in die aktuelle Konfiguration aufgenommen und schließlich die Werte der Hilfsvariablen in die Statechart-Variablen zurückkopiert werden. Formal beschreiben wir dies

durch die Relation

$$\sigma_0 \rho_{micro} \sigma_1 \quad :\Leftrightarrow \quad \exists\ st \in Steps(\sigma), n \in \mathbb{N}, \alpha = \alpha_1; \ldots; \alpha_n \text{ mit } \{\alpha_1, \ldots, \alpha_n\} \in action_set(st, \sigma) \text{ und } \sigma_0 \rho_{micro_{\alpha,st}} \sigma_1 \tag{1}$$

mit

$$\rho_{micro_{\alpha,st}} := \rho_{alive} \circ \rho_{copy} \circ \rho_{reset} \circ \rho_{exec_\alpha} \circ \rho_{upd_{st}} \circ \rho_{set} \circ \rho_{states_{st}}.$$

Wir können einen Mikro-Schritt ausführen, wenn in der aktuellen Belegung kein Terminierungszustand $s \in term(SC)$ aktiv ist.

$$\sigma \rho_{alive} \sigma' :\Leftrightarrow \forall\ s \in term(s).\ \sigma(s) = \text{false und } \sigma = \sigma'$$

Die Funktion *copy* kopiert die aktuellen Werte der Variablen und Ereignisse eines Statecharts SC aus $events(SC) \cup vars(SC)$ in die Hilfsvariablen, die im Weiteren verändert werden.

$$copy(\sigma)(v) := \begin{cases} \sigma(x), & \text{wenn } v \equiv \tilde{x}, \text{ und } \tilde{x} \in \widetilde{vars}(SC) \cup \widetilde{events}(SC) \\ \sigma(v), & sonst \end{cases}$$

Es gilt:

$$\sigma \rho_{copy} \sigma' :\Leftrightarrow \sigma' = copy(\sigma)$$

Dann werden alle lokalen Ereignisse zurückgesetzt und, falls zuvor ein durch *tick* gekennzeichneter Makro-Schritt stattgefunden hat, auch alle Umgebungsereignisse.

$$reset(\sigma)(v) := \begin{cases} \text{false}, & \text{wenn } v \in \widetilde{events}_{loc}(SC) \cup \widetilde{events}_{imp}(SC) \cup \{\widetilde{tick}\} \\ \text{false}, & \text{wenn } v \in \widetilde{events}_{env}(SC) \text{ und } \sigma(tick) \\ \sigma(v), & sonst \end{cases}$$

Es gilt:

$$\sigma \rho_{reset} \sigma' :\Leftrightarrow \sigma' = reset(\sigma)$$

Danach werden die Aktionen als sequentielle Programme ausgeführt. In wohlgeformten Statecharts ändern Programme nur die Hilfsvariablen $\tilde{x}$ und wir erhalten beim Zugriff auf x immer den Wert vor Schrittausführung. Wir können deshalb die Aktionen sequentiell ausführen. Damit Schreibkonflikte (zwei Übergänge weisen einer Variablen verschiedene Werte zu) indeterministisch aufgelöst werden, müssen sämtliche Ausführungsreihenfolgen der Aktionen betrachtet werden (siehe Gleichung (1)). Für eine feste Ausführungsreihenfolge α gilt

$$\sigma \rho_{exec_\alpha} \sigma' :\Leftrightarrow \sigma [\![\alpha]\!] \sigma'$$

mit der üblichen Programmsemantik (siehe Anhang E). Anschließend werden die abgeleiteten Ereignisse gesetzt. Es gilt:

$$\sigma \rho_{upd_{st}} \sigma' :\Leftrightarrow \sigma' = upd_{impl}(st, \sigma)$$

für

$$upd_{impl}(st,\sigma)(v) := \begin{cases} \text{true}, & \text{wenn } v \equiv \tilde{e},\ e = ex(s) \text{ und } s \in exited(st) \\ \text{true}, & \text{wenn } v \equiv \tilde{e},\ e = en(s) \text{ und } s \in entered(st) \\ \sigma(v), & sonst. \end{cases}$$

Nun stehen in den Hilfsvariablen die durch den Statechart-Schritt berechneten Werte, die wir schließlich noch in die Statechart-Variablen zurückkopieren müssen. Mit

$$set(\sigma)(v) := \begin{cases} \sigma(\tilde{v}), & \text{wenn } v \in events(SC) \cup vars(SC) \\ \sigma(v), & \text{sonst} \end{cases}$$

gilt

$$\sigma \rho_{set} \sigma' :\Leftrightarrow \sigma' = set(\sigma).$$

Schließlich müssen die aktiven Zustände des nächsten Schrittes gesetzt werden. Mit

$$upd_{states}(st,\sigma)(v) := \begin{cases} \text{true}, & \text{wenn } v \in active(st) \cup entered(st) \\ \text{false}, & \text{wenn } v \in states(SC)/(active(st) \cup entered(st)) \\ \sigma(v), & sonst \end{cases}$$

gilt

$$\sigma \rho_{upd_st} \sigma' :\Leftrightarrow \sigma' = upd_{states}(st,\sigma).$$

Mit $\sigma \rho_{micro} \sigma'$ führen wir dann einen Mikro-Schritt von der Belegung σ nach σ' aus.

Makro-Schritt Sobald sich das Statechart SC in einem stabilen Zustand befindet, in dem kein Mikro-Schritt mehr möglich ist, und das Statechart noch nicht terminiert wurde, führt es einen Makro-Schritt aus.

$$\rho_{macro} := \rho_{stable} \circ \rho_{alive} \circ \rho_{copy} \circ \rho_{reset} \circ \rho_{set} \circ \rho_{tick} \circ \rho_{input}$$

Er setzt zunächst alle lokalen Ereignisse zurück, erzeugt dann das *tick*-Ereignis und liest (zufällige) Ereignisse und Variablenbelegungen von der Umgebung ein. Die Relation ρ_{set} muss gelten, da das Zurücksetzen der lokalen Variablen nur in den Hilfsvariablen geschieht und ρ_{set} die Änderungen an die Statechart-Variablen der Nachfolgebelegung weitergibt. Mit

$$\begin{aligned} \sigma \rho_{stable} \sigma' &:\Leftrightarrow Steps(\sigma) = \emptyset \wedge \sigma = \sigma', \\ \sigma \rho_{tick} \sigma' &:\Leftrightarrow \sigma' = \sigma[tick/\text{true}], \\ \sigma \rho_{input} \sigma' &:\Leftrightarrow \forall\ v.\ v \in variables(SC)/(vars_{env}(SC) \cup events_{env}(SC)) \\ &\qquad \Rightarrow \sigma(v) = \sigma'(v), \end{aligned}$$

und $\sigma \rho_{macro} \sigma'$ führen wir eine Makro-Schritt von σ nach σ' aus.

Statechart Trace

Ein Statechart SC definiert die Menge der Variablen $variables(SC)$ und Folgen von Belegungen $(\sigma_0, \sigma_1, \ldots, \sigma_n)$, auch *Traces* genannt, mit $\sigma_i \in \sum(variables(SC))$. Traces können sowohl endliche als auch unendliche Folgen sein. Die Länge eines endlichen Traces $\pi = (\sigma_0, \sigma_1, \ldots, \sigma_n)$ ist $len(\pi) = n$, die eines unendlichen Traces π' ist $len(\pi') = \infty$. Einschränkungen auf Belegungen können auf eine Folge von Belegungen $\pi = (\sigma_0, \sigma_1, \ldots)$ wie folgt ausgedehnt werden:

$$\pi \mid_{V'} := (\sigma_0 \mid_{V'}, \sigma_1 \mid_{V'}, \ldots)$$

Ein Statechart SC definiert eine Menge maximaler Traces $traces^i_{\mathcal{A}}(SC)$ über einer Algebra $\mathcal{A}$, durch

$$traces^i_{\mathcal{A}}(SC) := \{(\sigma_0, \ldots, \sigma_n) \mid n \in \mathbb{N}_\infty, \sigma_0 \in init(SC), \sigma_j \; \rho_{step_{SC,\mathcal{A}}} \; \sigma_{j+1} \text{ und } \sigma_n \notin dom(\rho_{step_{SC,\mathcal{A}}})\}.$$

Ein Statechart-Trace beginnt in einer initialen Belegung, die einzelnen Belegungen genügen der Statechart-Relation und die letzte Belegung eines endlichen Traces hat keinen Nachfolger.

Die Semantik eines Statecharts SC definieren wir schließlich über der Menge V aller Variablen. Die Belegung der Variablen $variables(SC)$ sollen sich dabei entsprechend der Statechart-Relation ändern, während die Belegung aller anderen Variablen nicht eingeschränkt wird. Die auf die Statechart-Variablen eingeschränkte Folge von Belegungen muss also in der Tracemenge des Statecharts enthalten sein.

$$[\![SC]\!]^i_{\mathcal{A}} := \{\pi \mid \pi[j] \in \sum(V), \pi \mid_{variables(SC)} \in traces^i_{\mathcal{A}}(SC)\}$$

Wir sprechen damit in der Semantik explizit auch über Mikro-Schritte. Dies ist für die spätere interaktive Verifikation notwendig, damit der Benutzer die einzelnen Beweisschritte nachvollziehen kann. Die Änderungen, die sich von Makro- zu Makro-Schritt ergeben, sind meist zu komplex, als dass sie der Benutzer verstehen kann. Trotzdem haben wir eine Makro-Schritt-Semantik definiert, denn die Umgebungsereignisse werden nur zu Makro-Schritten eingelesen.

4.3 Spezifikation von Fehlverhalten

Im Folgenden beschreiben wir einige Möglichkeiten, wie man Fehlverhalten (von Hardwarekomponenten) mit Statecharts beschreiben kann. Die Beschreibung von Fehlverhalten ist für die formale FTA sehr wichtig, da sie auf einem Modell basiert, das die Wirklichkeit realistisch widerspiegeln soll. Mögliche Ausfälle von Komponenten sollen also im formalen Modell berücksichtigt werden. Wir beschränken uns auf den Ausfall oder Defekt von Komponenten und beschreiben keine Fehler, wie sie in der Netzwerkkommunikation auftreten (z. B. Nachricht geht verloren, kommt zu früh, ...).

Ausfälle und Defekte einer Komponente können verschiedener Art sein. Entweder eine Komponente agiert, obwohl es nicht nötig ist (fault of commission), oder eine Komponente

agiert nicht, obwohl es nötig ist (fault of omission). Die erste Fehlerart tritt beispielsweise bei einem Relais auf, das nur geöffnet ist, wenn Strom anliegt. Bei einem Stromausfall kann es sich dann unerwünscht schließen und einen Vorgang starten, der von einem zweiten Stromkreis gesteuert wird. Modelliert man diesen Fehlerfall mit Statecharts, so wird ein Zustandsübergang oder eine Aktion ausgelöst, ohne dass sie durch ein Ereignis oder eine Bedingung verursacht wird. Sehen wir uns dazu die Abbildung 4.7 a) an, in der das eben

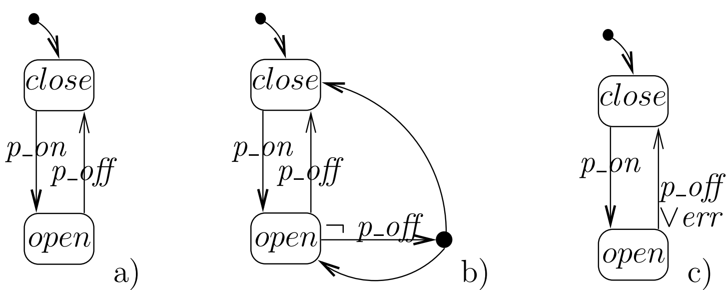


Abbildung 4.7: Komponente agiert, obwohl es nicht nötig ist

besprochene Relais modelliert ist. Der Zustand *open* kann durch Abschalten des Stroms durch das Ereignis *p_off* (power off) verlassen werden. In Abbildung 4.7 b) ist das Relais mit dem Fehler „Stromausfall" modelliert. Zusätzlich wurde der Fehlerübergang ¬ *p_off* hinzugefügt, durch den *open* indeterministisch verlassen werden kann, ohne auf das Ereignis *p_off* zu reagieren. Das Relais kann so spontan schließen und modelliert einen plötzlichen Stromausfall. Der Indeterminismus wurde gewählt, da ein Fehler passieren *kann*, aber nicht *muss*. Durch diese Modellierung kann zu jedem Zeitpunkt der Fehlerfall auftreten. Alternativ dazu kann eine boolesche Umgebungsvariable *err* eingeführt werden, die beliebig wahr oder falsch werden kann. Diese Möglichkeit ist in Abbildung 4.7 c) spezifiziert. Die Umgebung löst dann den Fehler irgendwann aus.

Im Allgemeinen müssen bei Fehlern, die unnötige Aktionen oder Zustandswechsel auslösen, zusätzliche Übergänge oder Aktivierungsereignisse in das Statechart-Modell eingeführt werden.

Betrachten wir nun den Fehler einer Komponente, die nicht reagiert, obwohl es nötig ist. Z. B. kann ein verklebtes Relais, das im Normalfall bei anliegendem Strom schließen soll, trotz vorhandenen Stroms nicht schließen. Diese Fehlerart kann weiter in drei verschiedene Kategorien eingeteilt werden.

- permanenter Fehler

  Eine Komponente hat einen permanenten Fehler, wenn sie dauerhaft ihre geforderte Funktionalität nicht erfüllen kann.

- intermittierender Fehler (Aussetzer)

  Ein intermittierender Fehler ist ein sich sporadisch wiederholendes Fehlverhalten.

- transienter Fehler (flüchtiger Fehler)

  Bei einem transienten Fehler kann eine Komponente ihre geforderte Funktionalität vorübergehend nicht erfüllen.

In Abbildung 4.8 betrachten wir ein Relais, das bei anliegendem Strom schließen soll und die Fehlerfälle, dass trotz Anlegen von Strom das Relais nicht schließt. In Abbildung 4.8 a)

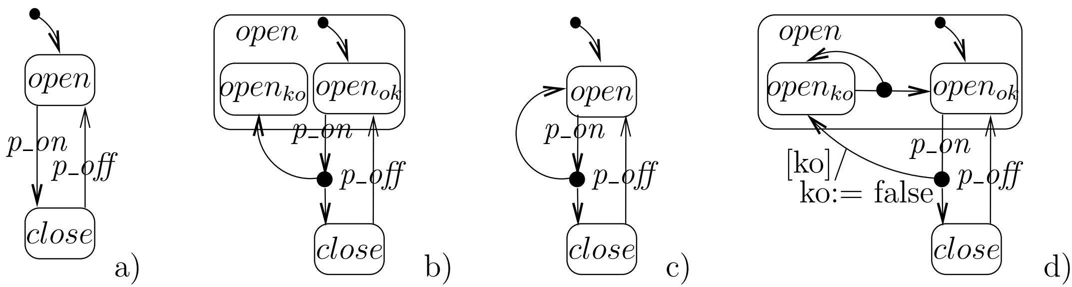


Abbildung 4.8: Komponente agiert nicht, obwohl es nötig ist

ist das korrekte Verhalten modelliert. Das Relais ist initial geöffnet und durch Anlegen von Strom (Ereignis *p_on*) wird es geschlossen. Bei einem permanenten Fehler verliert das Relais die Möglichkeit zu öffnen. Um dies zu modellieren, fügen wir in Abbildung 4.8 b) einen Fehlerzustand $open_{ko}$ hinzu, der indeterministisch eingenommen werden kann. Da das Relais auch im Fehlerfall geöffnet ist, führen wir einen Zustand $open_{ok}$ ein, dem die Übergänge des Zustands *open* im Modell ohne Fehler zugeordnet werden, und fassen die beiden Zustände $open_{ok}$ und $open_{ko}$ zu einem Zustand *open* zusammen. Der übergeordnete Zustand *open* kennzeichnet, dass das Relais fehlerhaft oder korrekterweise geöffnet ist.

Ein intermittierender Fehler kann einfacher modelliert werden. Hier verliert die Komponente zeitweise die Möglichkeit, auf Ereignisse zu reagieren. Anstatt beim Ereignis *p_on* in den Zustand *close* zu wechseln, kann das Statechart indeterministisch im Zustand *open* verbleiben (siehe Abbildung 4.8 c)). Dies kann immer wieder geschehen, solange der Zustand *open* aktiv ist.

Ein transienter Fehler unterscheidet sich vom intermittierenden Fehler dadurch, dass er einmal auftritt und dann eine bestimmte Zeit lang anhält. Wir modellieren einen flüchtigen Fehler wie in Abbildung 4.8 d) zu sehen. Wir können indeterministisch in den Fehlerzustand $open_{ko}$ übergehen (*ko* ist initial wahr), von dem aus nicht auf die Ereignisse anderer Komponenten reagiert wird. Nach einer bestimmten Zeit verlassen wir durch den indeterministischen Übergang den Fehlerzustand wieder. Wie im Fall 4.8 b) fassen wir die Zustände $open_{ok}$ und $open_{ko}$ zu einem Zustand *open* zusammen. Damit der Übergang von $open_{ok}$ nach $open_{ko}$ nur einmal genommen werden kann, setzen wir die Übergangsbedingung *ko* beim ersten Übergang auf falsch. Damit wird dieser Übergang für jeden weiteren Schritt deaktiviert.

Im Allgemeinen modellieren wir Fehler, bei denen eine Komponente nicht reagiert, obwohl sie sollte, durch zusätzliche Übergänge und Fehlerzustände. In keinem der Beispiele

wurden Fehler durch Weglassen von Übergängen modelliert. Dadurch würde die Möglichkeit, korrekt zu reagieren, verloren gehen. Wir würden eine Komponente modellieren, die *nur* falsch reagiert. Um dies zu vermeiden, werden mögliche Fehler immer durch indeterministische Übergänge modelliert.

Wird die Makro-Schritt-Semantik von STATEMATE benutzt, ist folgendes zu beachten. Diese Semantik führt bei obigen Modellierungen von Fehlverhalten durch indeterministische Übergänge zu endlosen Folgen von Mikro-Schritten. Beim asynchronen Zeitmodell muss daher dafür gesorgt werden, dass der Fehlerübergang nur einmal in einem Makro-Schritt erfolgt. Dies kann leicht durch Synchronisation der zusätzlich eingeführten Fehlerübergänge mit einem Makro-Schritt geschehen. In der Statechart-Semantik aus Abschnitt 4.2.2, die immer Makro-Schritte betrachtet, benutzen wir das *tick*-Ereignis zur Kennzeichnung eines Makro-Schrittes. Nur wenn *tick* gilt, wird ein zusätzlich eingeführter Fehlerübergang ausgeführt.

Diese Beispiele sollen genügen, um die Möglichkeit zur Spezifikation von Fehlverhalten mit Statecharts zu demonstrieren. In den Beispielen „Drucktank“ und „funkbasierter Fahrbetrieb“ genügte uns für die Modellierung die Muster aus Abbildung 4.7 c) und 4.8 c) für intermittierende Fehler (siehe z. B. Abbildung 7.2 für den Drucktank).

KAPITEL 5

Formale Fehlerbaumanalyse

Mit formalen Methoden werden präzise Systemspezifikationen erstellt. Mathematische Notationen helfen, Inkonsistenzen zu erkennen und nachzuweisen. Mit Beweiskalkülen kann zusätzlich überprüft werden, ob formal spezifizierte (Sicherheits-) Anforderungen von der Systemspezifikation erfüllt werden. Die Integration der FTA mit formalen Methoden ermöglicht es, dieses Vorgehen auch auf Fehlerbäume anzuwenden. Die Idee der formalen FTA ist, aus einem Fehlerbaum Anforderungen zu erzeugen, die beschreiben, dass (i) Ursachen im Fehlerbaum zur Wirkung führen und (ii) es keine weiteren Ursachen für die Wirkung gibt. Die Anforderungen drücken die *Korrektheit* und *Vollständigkeit* eines Fehlerbaums aus. Die Korrektheits- und Vollständigkeitsbedingungen können dann über dem formalen Modell geprüft werden. Die FTA wird eine formale Technik.

Die Vollständigkeit ist *die* wesentliche Eigenschaft eines Fehlerbaums, denn sie garantiert, dass bei der FTA nachweisbar alle Ursachen für die untersuchte Gefährdung berücksichtigt wurden. Die Korrektheit ist eine wünschenswerte Eigenschaft. Sie zeigt, dass in der FTA nur Ereignisse als Ursachen betrachtet werden, die etwas zum Gefährdungspotential beitragen. Die Korrektheit validiert den Fehlerbaum.

Die Stärke der FTA ist, dass die berechneten minimalen Schnittmengen zur qualitativen und quantitativen Sicherheitsbewertung herangezogen werden können. Um die Sicherheitsbewertung auch für formale Modelle anwenden zu können, darf die Bedeutung der minimalen Schnittmengen nicht verloren gehen. Die wichtigste Eigenschaft der minimalen Schnittmengen ist, dass, sobald ein Ereignis einer minimalen Schnittmenge ausgeschlossen werden kann, diese Schnittmenge die untersuchte Gefährdung nicht mehr verursachen kann. Die Formalisierung der FTA muss also so gewählt werden, dass sie die Eigenschaften der Schnittmengen erhält. Gelingt dies, profitiert nicht nur die FTA von der Integration mit den formalen Methoden. Die formalen Methoden gewinnen die Möglichkeit, neben der funktionalen Korrektheit auch Aussagen über die Robustheit eines Modells bezüglich Ausfällen und Defekten von Komponenten zu treffen. Dies ist eine neue Qualität formaler Methoden. Außerdem hilft die FTA Sicherheitseigenschaften systematisch abzuleiten, die für ein Modell geprüft werden sollen. Ein globales Sicherheitsziel, wie der Ausschluss einer Gefährdung, wird mit der FTA auf Anforderungen an einzelne Systemkomponenten

heruntergebrochen.

Wir arbeiten im Folgenden präzise Anforderungen für eine FTA-Semantik heraus. Dabei wird sich zeigen, dass zeitliche Eigenschaften für dynamische Systeme eine wichtige Rolle spielen. Um zeitliche Eigenschaften ausdrücken zu können, formalisieren wir die FTA in Intervalltemporallogik (ITL), die wir kurz vorstellen werden. Dann definieren wir die Semantik der FTA in ITL und beweisen mit dem minimalen Schnittmengen-Theorem, dass die Bedeutung der Schnittmengen nach Abschnitt 2.2.2 durch unsere Formalisierung erhalten bleibt. Abschließend vergleichen wir unseren Ansatz der formalen FTA mit Arbeiten aus der Literatur. Diesen Vergleich greifen wir in Kapitel 8 nochmals auf und gehen dort detailliert auf die semantischen Unterschiede verschiedener FTA-Formalisierungen ein.

5.1 Motivation der FTA-Semantik

Die Integration der FTA mit formalen Methoden soll den Nachweis von Korrektheits- und Vollständigkeitseigenschaften von Fehlerbäumen ermöglichen. Ein Fehlerbaum ist korrekt, wenn die Basisereignisse in den minimalen Schnittmengen die untersuchte Gefährdung verursachen können, und vollständig, wenn nur die in den minimalen Schnittmengen enthaltenen Basisereignisse die Gefährdung verursachen können (es also keine anderen Ursachen gibt). Die Korrektheit und Vollständigkeit stellen damit eine Beziehung zwischen den minimalen Schnittmengen und der untersuchten Gefährdung dar.

Für die Integration der FTA mit formalen Methoden müssen wir die Ereignisse einer FTA im Sprachgebrauch formaler Modelle ausdrücken. Für die formale FTA sind Ereignisse Formeln über dem formalen Modell. Da wir die Semantik der FTA in ITL spezifizieren, werden Ereignisse durch ITL-Formeln beschrieben. Wir unterscheiden punktuelle und temporale Ereignisse (siehe Abschnitt 2.2.1, Seite 9). Im Gegensatz zu einem temporalen Ereignis hat ein punktuelles Ereignis keinen zeitlichen Verlauf und wird durch prädikatenlogische Formeln – eine Teilmenge der ITL-Formeln – beschrieben. Entsprechend definieren wir ein temporales Ereignis durch ITL-Formeln, die nicht in der Teilmenge der prädikatenlogischen Formeln liegen.

Eine mögliche Alternative der Semantikdefinition für die FTA ist, die Beziehung zwischen den minimalen Schnittmengen und der Gefährdung direkt zu formalisieren und die Gatter eines Fehlerbaums überhaupt nicht zu betrachten. Wenn jedoch die FTA-Semantik nur über der Gefährdung und den minimalen Schnittmengen definiert ist, verliert man eine Stärke der FTA: die strukturierte Analyse von Ursachen für eine Gefährdung über die Zwischenereignisse im Fehlerbaum. Deshalb haben wir diese Alternative verworfen und formalisieren die einzelnen Gatter im Fehlerbaum. Jedem Gatter ordnen wir eine Formel zu, die lokal die Semantik des Gatters beschreibt. Diese Formel beschreibt die Beziehung zwischen der Ursache und den Unterereignissen eines Gatters. Die Semantik eines Fehlerbaums ist dann die Konjunktion aller Formeln der Fehlerbaumgatter.

Im Folgenden definieren wir die Anforderungen für die Formalisierung der Fehlerbaumgatter. Wir betrachten Gatter, die eine Wirkung ψ mit zwei Ursachen φ_1 und φ_2 verknüpfen. Die Verallgemeinerung auf mehrere Ursachen ist offensichtlich und wird der

Übersichtlichkeit halber weggelassen. Um die Anforderungen herauszuarbeiten, betrachten wir nochmals den Fehlerbaum „Kollision“ für den funkbasierten Bahnübergang (siehe Abschnitt 3.4). Die für diesen Abschnitt wichtigen Details sind nochmals in Abbildung 5.1 zusammengefasst.

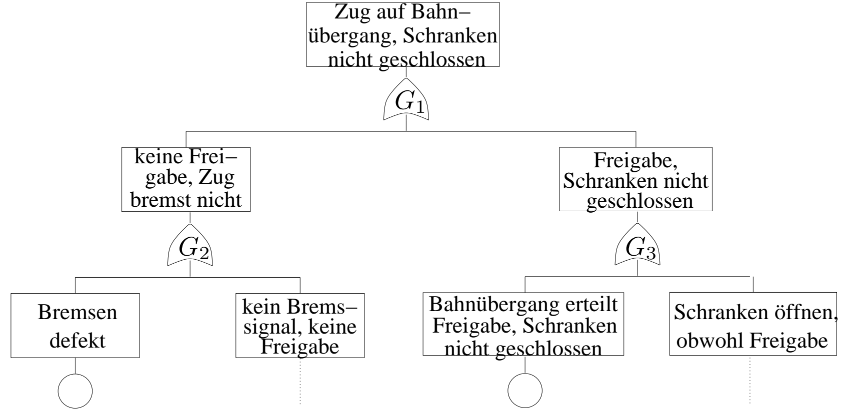


Abbildung 5.1: Fehlerbaum „Kollision“ des FFBs

Die klassische FTA beschreibt die Bedeutung der *Und*- bzw. *Oder*-Gatter informell als Konjunktion bzw. Disjunktion [VGRH81, Lei95]. Die Korrektheit – Ursachen führen zur Wirkung – ist formal die Implikation $\varphi_1 \wedge \varphi_2 \rightarrow \psi$ für ein *Und*- bzw. $\varphi_1 \vee \varphi_2 \rightarrow \psi$ für ein *Oder*-Gatter. Die Vollständigkeit – es kann nicht sein, dass die Wirkung auftritt und nicht die Ursachen, formalisiert als $\neg\,(\psi \wedge \neg\,(\varphi_1 \wedge \varphi_2))$ bzw. $\neg\,(\psi \wedge \neg\,(\varphi_1 \vee \varphi_2))$ – ist die umgekehrte Implikation. D. h. $\psi \rightarrow (\varphi_1 \wedge \varphi_2)$ formalisiert die Vollständigkeit des *Und*- und $\psi \rightarrow (\varphi_1 \vee \varphi_2)$ des *Oder*-Gatters.

Bei einer formalen FTA führt die Konjunktions/Disjunktions-Semantik jedoch leicht dazu, dass der Fehlerbaum nach der Formelstruktur der Ereignisse aufgebaut wird. Die Wirkung $\psi := \varphi_1 \wedge \varphi_2$ wird durch ein *Und*-Gatter mit den Ursachen φ_1 und φ_2 verknüpft. Diese Vorgehensweise führt nicht immer zum Erfolg. Sehen wir uns dazu die Gefährdung „Kollision“

$$\textit{Zug auf Bahnübergang, Schranken nicht geschlossen} \tag{1}$$

an. Die Ursachen für die Kollision sind nach diesem Ansatz *Zug auf Bahnübergang* und *Schranken nicht geschlossen*. Jedoch ist keine der beiden Ursachen für sich genommen ein Fehler, und es ist sicher nicht gewünscht, einen Bahnübergang zu spezifizieren, der entweder *Zug auf Bahnübergang* oder *Schranken nicht geschlossen* verhindern muss, um einen sicheren Betrieb zu gewährleisten. Deshalb ist im Allgemeinen die Zerlegung nach der Formelstruktur abzulehnen, denn die Ereignisse im Fehlerbaum müssen immer „unerwünscht“ sein.

Hansen et al. definieren in [HRS98] die Semantik der Fehlerbaumgatter als Konjunktion bzw. Disjunktion und präsentieren prompt einen Beispiel-Fehlerbaum, der diesen methodischen Fehler enthält.

Kehren wir nochmals zur Abbildung 5.1 zurück, in der die Kollision in die beiden Ursachen

keine Freigabe, Zug bremst nicht (2)

oder

Freigabe, Schranken nicht geschlossen (3)

zerlegt ist. Die Analyse des Zwischenereignisses 3 ergibt die beiden Ursachen

Bahnübergang erteilt Freigabe, obwohl Schranken nicht geschlossen (4)

oder

Schranken öffnen, obwohl Freigabe. (5)

Die Zerlegung G_3 von Ereignis (3) zu Ereignis (4) oder Ereignis (5) zeigt einen weiteren Grund, warum eine boolesche Konjunktions/Disjunktions-Semantik nicht ausreichend ist. Die Ursache (5) kann nicht gleichzeitig mit der Wirkung (3) auftreten. Die Schranken müssen zuerst öffnen, d. h. die Ursache muss zeitlich *vor* der Wirkung liegen. Solche *Ursache/Wirkungs*-Verknüpfungen (UW-Verknüpfungen) können nicht mit der booleschen Semantik ausgedrückt werden. In einer früheren Arbeit haben Hansen et al. [HRS94] eine Fehlerbaumsemantik angegeben, die Ursache/Wirkungs-Beziehungen beschreibt. Diese Semantik hat jedoch eine Schwäche bei temporalen Ereignissen (siehe dazu Abschnitt 5.5 und Kapitel 8) und wurde in späteren Arbeiten wieder verworfen.

Um Ursache/Wirkungs-Beziehungen in Fehlerbäumen ausdrücken zu können, führen wir zusätzlich zu den herkömmlichen *Und*- bzw. *Oder*-Gattern weitere, sogenannte *Ursache/Wirkungs*-Gatter (*UW*-Gatter), ein und nennen die herkömmlichen Gatter *Zerlegungs*- bzw. *Z*-Gatter. Im Fehlerbaum „Kollision“ ist das Gatter G_1 ein *Z-Oder*-Gatter, das Gatter G_3 ein *UW-Oder*-Gatter. Eine Unterscheidung zwischen Zerlegungs- und Ursache/-Wirkungs-Gatter haben auch Bruns und Anderson [BA93] getroffen. Zusätzlich müssen wir bei *UW-Und*-Gattern noch zwei Fälle unterscheiden. Entweder die Ursachen müssen gleichzeitig, *synchron*, auftreten, damit die Ursache eintritt, oder es genügt, wenn sie *asynchron* auftreten.

Betrachten wir nun nochmals die erste Zerlegung G_1 in Abbildung 5.1. Das Ereignis (1) „Kollision“ fordert, dass sich der Zug auf dem Bahnübergang befindet. Diese Bedingung wird von der Ursache (3) nicht mehr gefordert. Es genügt, dass die Schranken nicht geschlossen sind und eine Freigabe gesendet wurde, unabhängig von der Position des Zuges. Damit wird die Menge der als *unsicher* betrachteten Zustände vergrößert. Es werden Sicherheitsmargen eingeführt. Diese Entscheidung führt dazu, dass nicht jeder als unsicher betrachtete Zustand tatsächlich zu einer Gefährdung führt. Liegt z. B. das fehlerhafte Ereignis (3) vor und der Zug hat den Bahnübergang noch nicht erreicht, kann der Bahnübergang in der Zeit, bis der Zug eintrifft, durchaus noch gesichert werden. Trotzdem wird das Ereignis (3) als unerwünschtes Ereignis weiter untersucht.

Für die Gatter im Fehlerbaum müssen wir nun eine Formalisierung finden, die dieses Problem löst. Eine Möglichkeit ist, die Korrektheit der Gatter so zu formalisieren, dass Ursachen zur Wirkung führen können, nicht aber notwendigerweise müssen. Wir nennen dies

schwache Korrektheit. Dies entspricht der informellen Interpretation der Fehlerbaumgatter, ist aber sehr schwammig. Bedeutet „kann“ *fast immer*, *häufig* oder *mindestens einmal*? Außerdem ist die Formalisierung für die Validierung der Gatter nicht restriktiv genug. Sehr viele Ereignisse erfüllen die schwachen Korrektheitsbedingungen, ohne tatsächlich eine Ursache für die Wirkung des Gatters zu sein. Die Validierung der Gatter hätte kaum eine Aussage. Wir fordern für die formale FTA deshalb eine *starke Korrektheit*, d. h. die Ursachen führen immer zur Wirkung. Die starke Korrektheit wird auch in den Ansätzen aus der Literatur gewählt, die Korrektheitsbedingungen formalisieren [BA93, GW95].

Mit der starken Korrektheit können wir aber das obige Beispiel, in dem eine Sicherheitsmarge eingeführt wird, nicht beschreiben. Wir können die starke Korrektheitsbedingung nicht nachweisen. Deshalb kennzeichnen wir Stellen im Fehlerbaum, an denen Sicherheitsmargen eingeführt werden, explizit mit einem *Block*-Gatter. Im Allgemeinen kennzeichnet ein *Block*-Gatter Stellen in einem Fehlerbaum, bei denen die Wirkung nur dann eintritt, wenn zusätzlich zur Ursache die angegebene Nebenbedingung gilt. Gilt die Nebenbedingung nicht, *blockiert* das Gatter die Wirkung. In unserem Ansatz beschreiben die Nebenbedingungen die eingeführten Sicherheitsmargen.

In einem formalen Fehlerbaum müssen wir diese Sicherheitsmargen formalisieren. Im obigen Beispiel ist dies einfach (Zug auf Bahnübergang). Es hat sich aber herausgestellt, dass es häufig nicht möglich ist, die Sicherheitsmargen exakt zu beschreiben. Wir müssen uns deshalb entscheiden, ob wir notwendige oder hinreichende Nebenbedingungen für ein *Block*-Gatter formalisieren. Notwendige Bedingungen müssen eintreten, damit die Wirkung folgt, es könnten aber weitere Bedingungen notwendig sein, um die Wirkung zu erzwingen. Im Gegensatz dazu erfolgt bei hinreichenden Bedingungen die Wirkung zwingend, es könnte aber eine schwächere Bedingung ebenfalls die Wirkung erzwingen. D. h. es kann die Wirkung eintreten, obwohl die Nebenbedingung nicht erfüllt ist. Der Fehlerbaum beschreibt die Gefährdung nicht vollständig. Nicht vollständige Fehlerbäume sind höchst unerwünscht, denn sie lassen Sicherheitslücken offen. Deshalb haben wir uns für die Formalisierung der Nebenbedingung als notwendige Bedingung entschieden. Da notwendige Nebenbedingungen zusammen mit der Ursache aber nicht die Wirkung erzwingen, erfüllen die *Block*-Gatter keine Korrektheitsbedingung. In unserem Ansatz werden sie genau zu diesem Zweck eingesetzt. Sie kennzeichnen im Fehlerbaum Stellen explizit, die nicht die Korrektheitsbedingungen für ein Gatter erfüllen.

Auch für das *Block*-Gatter unterscheiden wir zwischen Zerlegungs- und Ursache/Wirkungs-Gatter. Im obigen Fall beschreibt die Nebenbedingung die eingeführte Sicherheitsmarge exakt. Die Ursachen für die Kollision (1) sind im Beispiel dann

keine Freigabe, Zug bremst nicht, Zug auf Bahnübergang (6)

oder

Freigabe, Schranken nicht geschlossen, Zug auf Bahnübergang. (7)

Die Ursache (7) ist durch ein *Block*-Gatter und durch die Nebenbedingung $\chi :=$ *Zug auf Bahnübergang* mit dem Ereignis (3) verknüpft.

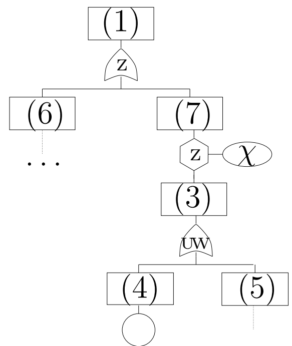


Abbildung 5.2: exakter Fehlerbaum

Mit den neu eingeführten Gattern können wir den Fehlerbaum „Kollision“ aus Abbildung 5.1 exakt beschreiben (siehe Abbildung 5.2). Die Kollision wird in die beiden Ursachen *keine Freigabe, Zug bremst nicht, Zug auf Bahnübergang* (6) und *Freigabe, Schranken nicht geschlossen, Zug auf Bahnübergang* (7) zerlegt. Die Fehlermenge von (7) wird durch das *Block*-Gatter mit der Nebenbedingung $\chi :=$ *Zug auf Bahnübergang* vergrößert. Die eingeführte Sicherheitsmarge ist, dass nicht mehr *Zug auf Bahnübergang* gelten muss, sondern schon *Freigabe, Schranken nicht geschlossen* (3) genügt, um einen unerwünschten Zustand zu erhalten. Dieser wird entweder durch *Bahnübergang erteilt Freigabe, obwohl Schranken nicht geschlossen* (4) oder *Schranken öffnen, obwohl Freigabe* (5) verursacht. Beide Ursachen können zeitlich vor (3) auftreten und werden deshalb mit einem *UW-Oder*-Gatter verknüpft.

Zusammenfassend erhalten wir 7 verschiedene Fehlerbaumgatter: die herkömmlichen *Z-Und*-, *Z-Oder*- und *Z-Block*-Gatter (Z, Z, Z), das *UW-Oder*- (UW) und *UW-Block*-Gatter (UW) sowie die synchronen und asynchronen *UW-Und*-Gatter (UW, A). Die klassische FTA definiert noch weitere Gatter, wie *priority*- oder *n aus m*-Gatter, bei denen Ereignisse in einer bestimmten Reihenfolge bzw. mindestens n der angegebenen m Unterereignisse auftreten müssen. Nach den Erfahrungen mit verschiedenen Fallstudien (z. B. [RST00b] oder [ORS$^+$02]), sind die 7 angegebenen Gatter für softwarebasierte Systeme ausreichend. Auch die meisten Semantikdefinitionen aus der Literatur (z. B. [HRS94, BA93]) beschränken sich auf diese Gattertypen.

Eine weitere Erkenntnis aus diesem Abschnitt ist, dass es im Allgemeinen nicht möglich ist, korrekte Fehlerbäume zu erstellen, in denen die Ereignisse der minimalen Schnittmengen immer zur Gefährdung führen. Trotzdem formulieren wir zur lokalen Validierung der Fehlerbäume Korrektheitsbedingungen für die einzelnen Gatter (mit Ausnahme der *Block*-Gatter, deren Ursache explizit nicht zur Wirkung führen muss). Die Semantikdefinition der FTA basiert jedoch nur auf den Vollständigkeitsbedingungen.

Schließlich müssen für eine formale FTA neben den Gattern auch die Ereignisse im Fehlerbaum formalisiert werden. Die formalisierten Ereignisse treten in den Gatterbedingungen als Ursachen bzw. Wirkungen wieder auf. In Abschnitt 2.1.1 haben wir die Unter-

scheidung in *punktuelle* und *temporale* Ereignisse getroffen. Punktuelle Ereignisse treten zu einem bestimmten Zeitpunkt ein. Sie entsprechen Ereignissen in der automatentheoretischen Sichtweise. In der Sicherheitsanalyse werden aber auch temporale Ereignisse – Ereignisse mit Dauern – betrachtet (z. B. "10 Sekunden bremsen"). Górski [GW95] hat in seinen Analysen drei typische Ereignismuster beobachtet.

1. ein Ereignis tritt (für eine bestimmte Zeit) ein, z. B. „(10 Sekunden) bremsen"
2. ein Ereignis tritt dauerhaft ein, z. B. „Kollision", sie kann nicht mehr rückgängig gemacht werden
3. eine Sequenz von Ereignissen (mit Zeitbedingungen) wird durchlaufen, z. B. „Ampel ist rot, dann gelb und schließlich grün"

Jedes dieser Muster kann einen zeitlichen Verlauf beschreiben. Für die formale FTA benötigen wir deshalb eine Temporallogik, die temporale Ereignisse und die Ursache/Wirkungs-Beziehungen der Fehlerbaumgatter adäquat beschreiben kann. Dafür wählen wir eine Intervalltemporallogik.

5.2 Intervalltemporallogik (ITL)

Die Semantikdefinition der formalen FTA basiert auf einer Intervalltemporallogik (ITL, [Mos85, CMZ02]) über einem diskreten Zeitmodell. Analog zu Statecharts, definieren wir ITL-Formeln als eine Erweiterung prädikatenlogischer Formeln erster Stufe über einer Signatur SIG und einer Menge von Variablen V. Die Syntax prädikatenlogischer Formeln und deren Semantikdefinition mit SIG-Algebren $\mathcal{A}$ und Belegungen (oder Zuständen) σ, die Variablen aus V auf passende Elemente der Trägermenge aus $\mathcal{A}$ abbilden, setzen wir als bekannt voraus [Wir90, Par93]. Die Menge aller möglichen Belegungen sei $\Sigma(V)$.

Die Semantik der ITL basiert auf einer endlichen oder unendlichen Sequenz von Zuständen, einem Intervall oder Trace $\mathcal{I} = (\sigma_0, \sigma_1 \ldots \sigma_n),\ n \in \mathbb{N} \cup \{\infty\}$. Für ein Intervall $\mathcal{I}$ berechnet $\mathit{length}(\mathcal{I}) = n$ die Länge, $\mathcal{I}\,|^i := (\sigma_0, \ldots, \sigma_i)$ selektiert einen Präfix, $\mathcal{I}\,|_i := (\sigma_i, \ldots)$ einen Postfix und $\mathcal{I}\,|_i^j := (\sigma_i, \ldots, \sigma_j)$ ein Sub-Intervall. Zustände σ_i sind eine Zuordnung von Variablen $v \in V$ zu ihren Werten $\sigma_i(v) \in \Sigma(V)$. Wir werten ITL-Formeln über einer Algebra $\mathcal{A}$ und einem Intervall $\mathcal{I}$ aus. Sei Fma_{ITL} die Menge aller ITL-Formeln und $Fma_{PL} \subset Fma_{ITL}$ die Menge aller prädikatenlogischen Formeln. Eine prädikatenlogische Formel $\Phi \in Fma_{PL}$ ist über $\mathcal{A}$ und $\mathcal{I}$ wahr, d. h. es gilt $\mathcal{A}, \mathcal{I} \models_{ITL} \Phi$, wenn Φ über $\mathcal{A}$ und dem aktuellen Zustand $\mathcal{I}(0)$ mittels der Semantikdefinition für Formeln erster Stufe $\mathcal{A}, \mathcal{I}(0) \models_{PL} \Phi$ zu true ausgewertet wird. Wenn die Bedeutung aus dem Zusammenhang klar ist, schreiben wir sowohl für $\models_{PL}$ als auch für $\models_{ITL}$ einfach $\models$. Ist die Algebra $\mathcal{A}$ für das Verständnis nicht wichtig, vereinfachen wir $\mathcal{A}, \mathcal{I} \models_{ITL} \Phi$ zu $\mathcal{I} \models \Phi$.

Wir bilden temporale Formeln $\varphi \in Fma_{ITL}$ über prädikatenlogischen Formeln erster Stufe $\Phi \in Fma_{PL}$ mit den üblichen aussagenlogischen Verknüpfungsoperatoren und den zusätzlichen temporalen Operatoren. Eine temporale Formel φ gilt (über einer Algebra $\mathcal{A}$),

wenn für alle $\mathcal{I}$ gilt: $\mathcal{I} \models \varphi$. An dieser Stelle definieren wir nur die Temporaloperatoren, die später für die Definition der FTA-Semantik notwendig sind. Die ITL definiert noch weitere Operatoren [CMZ02]. Der *Chop*-Operator „**;** “ ist die Basis der ITL-Operatoren. Im Folgenden sei $n = \mathit{length}(\mathcal{I})$ die Länge des entsprechenden Intervalls $\mathcal{I}$.

Definition 5.1 *Chop-Operator*

Das Intervall kann so geteilt werden, dass φ im ersten und ψ im zweiten Teilintervall gilt.

$$\mathcal{I} \models \varphi \mathbin{;} \psi \quad :\Leftrightarrow \quad \textit{es gibt ein ein } c \neq \infty \textit{ mit } c \leq n : \\ \mathcal{I}|^c \models \varphi \textit{ und } \mathcal{I}|_c \models \psi$$

Auf dem Chop-Operator aufbauend, leiten wir weitere Operatoren ab.

Definition 5.2 *Abgeleitete ITL-Operatoren*

$\Diamond\varphi := \text{true} \mathbin{;} \varphi$	*in einem Postfix-Interval gilt* φ
$\Box\varphi := \neg\,(\Diamond\neg\,\varphi)$	*in allen Postfix-Intervallen gilt* φ
$\Diamond_i\varphi := \varphi \mathbin{;} \text{true}$	*in einem initialen Interval gilt* φ
$\Box_i\varphi := \neg\,(\Diamond_i\neg\,\varphi)$	*in allen initialen Intervallen gilt* φ
$\Diamond_s\varphi := \text{true} \mathbin{;} \varphi \mathbin{;} \text{true}$	*in einem Sub-Interval gilt* φ
$\Box_s\varphi := \neg\,(\Diamond_s\neg\,\varphi)$	*in allen Sub-Intervallen gilt* φ

Zum besseren Verständnis geben wir die Semantik der abgeleiteten Operatoren explizit an.

Lemma 5.1 *Explizite Semantik der ITL-Operatoren*

$$\begin{array}{lll}
\mathcal{I} \models \Diamond\varphi :\Leftrightarrow & \textit{es gibt ein } c \neq \infty \textit{ mit } c \leq n : & \mathcal{I}|_c \models \varphi \\
\mathcal{I} \models \Box\varphi :\Leftrightarrow & \textit{für alle } c \neq \infty \textit{ mit } c \leq n : & \mathcal{I}|_c \models \varphi \\
\mathcal{I} \models \Diamond_i\varphi :\Leftrightarrow & \textit{es gibt ein } c \neq \infty \textit{ mit } c \leq n : & \mathcal{I}|^c \models \varphi \\
\mathcal{I} \models \Box_i\varphi :\Leftrightarrow & \textit{für alle } c \neq \infty \textit{ mit } c \leq n : & \mathcal{I}|^c \models \varphi \\
\mathcal{I} \models \Diamond_s\varphi :\Leftrightarrow & \textit{es gibt ein ein } c,\, d \neq \infty \textit{ mit } c \leq d \leq n : & \mathcal{I}|_c^d \models \varphi \\
\mathcal{I} \models \Box_s\varphi :\Leftrightarrow & \textit{für alle } c,\, d \neq \infty \textit{ mit } c \leq d \leq n : & \mathcal{I}|_c^d \models \varphi
\end{array}$$

Zur Spezifikation temporaler Ereignisse erweitern wir die ITL um quantifizierende Operatoren. Die Operatoren sind aus dem *Quantified Discrete-Time Duration Calculus (QDDC)* von Pandya [Pan01b] übernommen. Die quantifizierenden Operatoren approximieren die Duration Calculus Terme aus dem Duration Calculus (DC, [ZH97]) für den diskreten Fall. Ursprünglich ist der DC eine Erweiterung der ITL und über einer kontinuierlichen Zeitachse definiert. Der diskrete DC ist jedoch über einer diskreten Zeitachse definiert und entspricht damit wieder der ITL. Wir können deshalb die ITL um folgende quantifizierende Operatoren über Formeln der Prädikatenlogik $\Phi \in \mathit{Fma}_{PL}$ erweitern.

Definition 5.3 *quantifizierende Operatoren*

Für prädikatenlogische Formeln $\Phi \in \mathrm{Fma}_{PL}$ *und* $\oplus \in \{<, =\}$ *gilt*

$\ell \oplus c$:	*die Länge* ℓ *des Intervalls ist kleiner bzw. gleich* c
$\lceil \Phi \rceil^0$:	Φ *gilt im Intervall der Länge 0*
$\lceil\lceil \Phi \rceil$:	Φ *gilt im ganzen Intervall*

Definition 5.4 *Semantik quantifizierender Operatoren*

Für prädikatenlogische Formeln $\Phi \in \mathrm{Fma}_{PL}$*,* $\oplus \in \{<, =\}$ *und* $n = \mathrm{length}(\mathcal{I})$ *gilt*

$$\mathcal{I} \models \ell \oplus c :\Leftrightarrow \quad \textit{wenn } n \oplus c$$
$$\mathcal{I} \models \lceil \Phi \rceil^0 :\Leftrightarrow \quad \textit{wenn } n = 0 \textit{ und } \mathcal{I} \models \Phi$$
$$\mathcal{I} \models \lceil\lceil \Phi \rceil :\Leftrightarrow \quad \textit{wenn für alle c: } c < n \textit{ gilt: } \mathcal{I}|_c \models \Phi$$

In ITL ohne quantifizierende Operatoren kann $\lceil\lceil \Phi \rceil$ auch mit $\Box\Phi$ ausgedrückt werden. Ein temporales Ereignis „10 Sekunden bremsen“ definieren wir nun mit dem entsprechenden Prädikat *brake* folgendermaßen: $\lceil\lceil brake \rceil \wedge \ell = 10$. Eine Folge von Ereignissen $\varphi_1, \ldots, \varphi_m$ mit Dauern $n_1, \ldots, n_m$ mit $\lceil\lceil \varphi_1 \rceil \wedge \ell = n_1 \ ; \ \ldots \ ; \ \lceil\lceil \varphi_m \rceil \wedge \ell = n_m$.

5.3 Formalisierung in ITL

Wir definieren die Semantik der FTA lokal über den einzelnen Fehlerbaumgattern. Die globale Semantik eines Fehlerbaums ergibt sich dann aus der Semantik der einzelnen Gatter, die ein Fehlerbaum enthält, und in Abschnitt 5.3.2 definiert wird.

5.3.1 Semantik der Fehlerbaumgatter

In Abschnitt 5.1 haben wir die Anforderungen für die formale FTA abgeleitet und die Notwendigkeit für insgesamt 7 Fehlerbaumgatter herausgearbeitet. Drei Zerlegungsgatter, die Gattern der herkömmlichen FTA entsprechen, und vier Ursache/Wirkungs-Gatter die den dynamischen Aspekten eingebetteter Systeme gerecht werden. Um die dynamischen Aspekte in der Semantikdefinition abzubilden, formalisieren wir die Gatter in ITL. Jedem Gatter ordnen wir eine separate Formel für die Korrektheit und die Vollständigkeit zu [STR02]. Die Vollständigkeitsbedingung definiert die Semantik eines Gatters, die Korrektheitsbedingung dient zur Validierung. Betrachten wir als erstes die drei Zerlegungsgatter.

Definition 5.5 *Semantik* Z-Und-*Gatter*

ψ
z
φ_1 φ_2

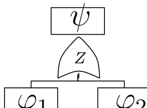
Korrektheit: $\boxtimes\ (\varphi_1 \wedge \varphi_2 \rightarrow \psi)$

Vollständigkeit: $\boxtimes\ (\psi \rightarrow \varphi_1 \wedge \varphi_2)$

Ein *Z-Und*-Gatter ist dann korrekt, wenn die Ursachen immer die Wirkung bedingen. Sobald beide Ursachen eintreten, tritt auch die Wirkung ein. Dies ist formal eine Implikation. Da die formale FTA auf dynamische Systeme angewendet wird, muss die Implikation zu jedem Zeitpunkt des Systemablaufs gelten. Deshalb fordern wir für alle Sub-Intervalle ($\boxtimes$), dass aus den Ursachen die Wirkung folgt.

Die Vollständigkeit ist dual zur Korrektheit definiert. Sobald die Wirkung eintritt, müssen die Ursachen vorhanden sein. Wäre dies nicht der Fall, müssten weitere Ursachen existieren und die Wirkung wäre nicht vollständig beschrieben.

Definition 5.6 *Semantik* Z-Oder-*Gatter*

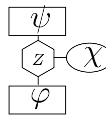


Korrektheit: $\boxtimes\ (\varphi_1 \vee \varphi_2 \rightarrow \psi)$

Vollständigkeit: $\boxtimes\ (\psi \rightarrow \varphi_1 \vee \varphi_2)$

Das *Z-Oder*-Gatter ist analog zum *Z-Und*-Gatter definiert. Sowohl für die Korrektheit als auch für die Vollständigkeit müssen aber nicht beide Ursachen vorhanden sein, sondern es genügt eine. Damit ist die Semantik der Z-Gatter analog der informellen Bedeutung als boolesche Konjunktion bzw. Disjunktion definiert.

Definition 5.7 *Semantik* Z-Block-*Gatter*



Vollständigkeit: $\boxtimes\ (\psi \rightarrow \varphi \wedge \chi)$

Das *Z-Block*-Gatter beschreibt eigentlich keine Zerlegung, sondern kennzeichnet, dass unter bestimmten Bedingungen die Ursache und die Wirkung gleichzeitig eintreten. Wir verwenden sie auch, um Sicherheitsmargen einzuführen, d. h. die Ursache eines *Block*-Gatters tritt häufiger als die Wirkung ein. Ein *Z-Block*-Gatter definiert, welche Bedingung χ gelten muss, damit die Ursache φ die Wirkung ψ verursacht. Bei der Diskussion in Abschnitt 5.1 hat sich gezeigt, dass die Nebenbedingung χ als notwendige Bedingung adäquat beschrieben wird. Da notwendige Bedingungen aber nicht die Wirkung erzwingen, gilt im Allgemeinen keine Korrektheitsbedingung für das *Z-Block*-Gatter. Formalisiert die Nebenbedingung χ exakt die Sicherheitsmarge, kann zusätzlich die Korrektheitsbedingung $\boxtimes(\varphi \wedge \chi \rightarrow \psi)$ für ein *Z-Block*-Gatter gezeigt werden. Für die Vollständigkeit verlangen wir, dass die Nebenbedingung χ zusammen mit der Ursache φ die Wirkung verursacht. Die Vollständigkeitsbedingung garantiert damit, dass wir „nur" eine notwendige Bedingung formalisieren. Würden wir eine stärkere Bedingung χ' formalisieren als tatsächlich für das Gatter notwendig ist, würden wir die Vollständigkeitsbedingung nicht zeigen können, da es eine Situation gibt, in der die Wirkung gilt, nicht aber die stärkere Bedingung χ'.

Interessanterweise würden wir bei der Formalisierung des *Block*-Gatters mit hinreichender Bedingung für die Vollständigkeit nur das Eintreten der Ursache verlangen, also $\boxtimes(\psi \rightarrow \varphi)$ nachweisen. Dies genügt, da die Nebenbedingung kein Ausfall oder Fehler ist. Den Nachweis, dass die Nebenbedingung hinreichend ist, würden wir mit der obigen Korrektheitsbedingung $\boxtimes\varphi \wedge \chi \rightarrow \psi$ erhalten. Der Nachweis für notwendige Nebenbedingun-

gen steckt also in der Vollständigkeitsbedingung, der Nachweis für hinreichende Nebenbedingung in der Korrektheitsbedingung.

Die Definition des *Z-Block*-Gatters entspricht dem *Z-Und*-Gatter (mit der Nebenbedingung χ als zweite Ursache). Jedoch ist bei der Anwendung der Gatter der methodische Unterschied zu betrachten. Bei der Fehlerbaumanalyse sind alle Ereignisse im Fehlerbaum unerwünschte Ereignisse. Deshalb sind die Ursachen beim *Und*-Gatter unerwünscht, während die Nebenbedingung χ kein Fehler sein muss, sondern durchaus im Normalbetrieb eines Systems auftreten darf.

Definition 5.8 *Semantik synchrones* UW-Und-*Gatter*

ψ — UW — φ_1, φ_2

Korrektheit: $\Diamond\, (\varphi_1 \wedge \varphi_2) \rightarrow \Diamond\, \psi$

Vollständigkeit: $\neg\, (\neg \Diamond\, (\varphi_1 \wedge \varphi_2)\ ;\ \diamond\, \psi)$

Ein synchrones *UW-Und*-Gatter ist genau dann korrekt, wenn irgendwann ($\Diamond$) beide Ursachen zusammen auftreten und dann die Wirkung irgendwann eintritt. Dies formalisiert die starke Korrektheit, denn in ITL ist die Gültigkeit einer Formel über allen Pfaden definiert. Die Korrektheit verlangt für jeden Pfad: Tritt die Ursachen ein, dann tritt auch die Wirkung ein.

Die Korrektheitsbedingung fordert aber nicht, dass die Ursachen vor der Wirkung eintreten, da dies nicht immer möglich ist. Es kann vorkommen, dass nach dem Eintritt einer Wirkung nochmals Ursachen (für dieselbe Wirkung) auftreten. Das erneute Auftreten muss aber keine zweite Wirkung bedingen. Z. B. kann nach einer Kollision (Wirkung, die durch eine andere Ursache bedingt wurde) ein offener Bahnübergang, der eine Freigabe für den kollidierten Zug sendet (Ursache nach der Wirkung), nicht nochmals zu einer Kollision führen. Trotzdem muss *vor* der Kollision eine Ursache eingetreten sein. Diese Forderung wird durch die Vollständigkeitsbedingungen abgedeckt.

Die Vollständigkeit verlangt, dass es keinen Pfad geben kann, auf dem irgendwann die Wirkung ψ eintritt, zuvor (**;**) aber nirgends ($\neg \Diamond$) die Ursachen aufgetreten sind. Das erste Auftreten von Ursachen liegt also immer vor der Wirkung. Das Teilen des Intervalls durch den Chop-Operator garantiert, dass temporale Ursachen schon beendet sind (sie müssen *innerhalb* der ersten Pfadhälfte auftreten), bevor die Wirkung eintreten kann. Die Wirkung ψ selbst muss auf einem Anfangsstück ($\diamond$) der zweiten Pfadhälfte gelten.

Definition 5.9 *Semantik asynchrones* UW-Und-*Gatter*

ψ — A — φ_1, φ_2

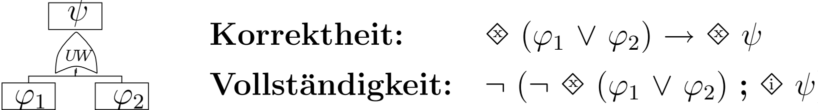
Korrektheit: $\Diamond\, \varphi_1 \wedge \Diamond\, \varphi_2 \rightarrow \Diamond\, \psi$

Vollständigkeit: $\neg\, (\neg \Diamond\, \varphi_1\ ;\ \diamond\, \psi)$
$\wedge\, \neg\, (\neg \Diamond\, \varphi_2\ ;\ \diamond\, \psi)$

Bei der asynchronen Variante des *UW-Und*-Gatters müssen auch beide Ursachen auftreten, aber nicht gleichzeitig. Deshalb genügt für die Korrektheit, dass irgendwann ($\Diamond$) die Ursache φ_1 und irgendwann die Ursache φ_2 eintritt, damit die Wirkung folgen muss. Wie

bei der synchronen Variante erzwingt die Vollständigkeit, dass die Ursachen vor der Wirkung auftreten. Sie fordert für die Abläufe des Systems, dass sie nicht so zerlegt werden können, dass die Wirkung ψ eintritt, ohne dass zuvor die Ursache φ_1 eingetreten ist *und* nicht so, dass ψ eintritt, ohne dass zuvor Ursache φ_2 eintrat. Damit können die Ursachen zeitlich versetzt eintreten, aber dennoch liegen beide vor der Wirkung.

Definition 5.10 *Semantik* UW-Oder-*Gatter*



Korrektheit: $\Diamond\,(\varphi_1 \vee \varphi_2) \rightarrow \Diamond\,\psi$

Vollständigkeit: $\neg\,(\neg\,\Diamond\,(\varphi_1 \vee \varphi_2)\ ;\ \Diamond\,\psi)$

Das *UW-Oder*-Gatter ist analog dem synchronen *UW-Und*-Gatter definiert. Sobald eine Ursache eintritt, folgt die Wirkung. Die Vollständigkeit verlangt, dass mindestens eine der Ursachen vor der Wirkung eintritt.

Definition 5.11 *Semantik* UW-Block-*Gatter*

Vollständigkeit: $\neg\,(\neg\,\Diamond\,\varphi\ ;\ \Diamond\,\psi)$
$\wedge\ \neg\,(\neg\,\Diamond\,\chi\ ;\ \Diamond\,\psi)$

Analog zu den *Und*- und *Oder*-Gattern gibt es auch eine *UW*-Variante des *Block*-Gatters. Sie folgt dem Definitionsschema der *UW*-Gatter und fordert, dass die Ursache φ und die Nebenbedingung χ zeitlich vor der Wirkung liegen. Die Nebenbedingung muss häufig nicht gleichzeitig mit der Ursache vorhanden sein, um die Wirkung zu bedingen. Deshalb verzichten wir hier auf eine synchrone Variante des *Block*-Gatters. Die Vollständigkeitsbedingung des *UW-Block*-Gatters entspricht der Vollständigkeitsbedingung des asynchronen *UW-Und*-Gatters mit dem methodischen Unterschied, dass die Nebenbedingung χ kein Fehler ist.

5.3.2 Semantik der FTA

Die Semantik eines Fehlerbaums ist strukturell über den verwendeten Gattern definiert. Die wesentliche Eigenschaft für einen Fehlerbaum ist die Vollständigkeit. Sie garantiert, dass Ursachen, die nicht im Fehlerbaum berücksichtigt sind, die untersuchte Gefährdung auch nicht verursachen können. Zusätzlich validieren die Korrektheitsbedingungen den Fehlerbaum. Die Formalisierung der *Block*-Gatter zeigt aber, dass wir nicht für jedes Gatter eine Korrektheitsbedingung angegeben haben und deshalb auch keine Korrektheitsbedingung für einen kompletten Fehlerbaum geben können.

Wir definieren die Semantik eines Fehlerbaums deshalb nur über den Vollständigkeitsbedingungen der Gatter. Die Korrektheitsbedingungen dienen zur weiteren Validierung der formalen FTA und zeigen möglicherweise unnötige Ursachen auf. Unnötig bedeutet, dass die Ursachen zwar vor der Wirkung auftreten, sie aber nicht bedingen.

Formal ist ein Fehlerbaum FT entweder ein Blatt φ oder ein Ereignis ψ, das über ein Gatter mit einer Menge von Fehlerbäumen $\mathcal{FT}$ verknüpft ist.

Definition 5.12 *Fehlerbaum*

Bezeichne $\mathcal{FT}$ eine Menge von Fehlerbäumen. Dann ist ein Fehlerbaum FT rekursiv definiert:

$$\begin{aligned} FT \ := \ & \mathrm{leaf}(\varphi) \mid \mathrm{zcon}(\psi, \mathcal{FT}) \mid \mathrm{zdis}(\psi, \mathcal{FT}) \mid \mathrm{zinh}(\psi, \chi, FT) \mid \\ & \mathrm{scon}(\psi, \mathcal{FT}) \mid \mathrm{acon}(\psi, \mathcal{FT}) \mid \mathrm{con}(\psi, \mathcal{FT}) \mid \mathrm{inh}(\psi, \chi, FT) \end{aligned}$$

*Ein Fehlerbaum ist entweder ein Blatt (*leaf(φ)*) oder verbindet ein Top-Ereignis ψ durch ein* Z-Und- *(*zcon$(\psi,\mathcal{FT})$*),* Z-Oder- *(*zdis$(\psi,\mathcal{FT})$*),* Z-Block- *(*zinh(ψ, χ, FT)*), synchronen oder asynchronen* UW-Und- *(*scon$(\psi,\mathcal{FT})$ *bzw.* acon$(\psi,\mathcal{FT})$*),* UW-Oder- *(*con$(\psi,\mathcal{FT})$*) oder* UW-Block-*Gatter (*inh$(\psi,\chi,\mathcal{FT})$*) mit einer Menge von Fehlerbäumen $\mathcal{FT}$. Die Funktion $f(FT)$ selektiert das oberste Fehlerereignis (die Wurzel) aus einem Fehlerbaum.*

Basierend auf der Semantik der Fehlerbaumgatter definieren wir strukturell die Vollständigkeit eines Fehlerbaums.

Definition 5.13 *Vollständigkeit eines Fehlerbaums*

Für Mengen von Fehlerbäumen $\mathcal{FT}$ und $FT_i \in \mathcal{FT}$ ist die Vollständigkeit $comp(FT)$ definiert als:

$$\begin{array}{lll}
FT \equiv \mathrm{leaf}(\varphi): & comp(FT) :\Leftrightarrow & true \\
FT \equiv \mathrm{zcon}(\psi, \mathcal{FT}): & comp(FT) :\Leftrightarrow & (\boxtimes(\psi \rightarrow \bigwedge_i f(FT_i))) \wedge \bigwedge_i comp(FT_i) \\
FT \equiv \mathrm{zdis}(\psi, \mathcal{FT}): & comp(FT) :\Leftrightarrow & (\boxtimes(\psi \rightarrow \bigvee_i f(FT_i))) \wedge \bigwedge_i comp(FT_i) \\
FT \equiv \mathrm{zinh}(\psi, \chi, FT'): & comp(FT) :\Leftrightarrow & (\boxtimes(\psi \rightarrow f(FT) \wedge \chi)) \wedge comp(FT') \\
FT \equiv \mathrm{scon}(\psi, \mathcal{FT}): & comp(FT) :\Leftrightarrow & (\neg\ (\neg\ \Diamond_{\mathrm{x}} \bigwedge_i f(FT_i) \mathbin{;} \Diamond_{\mathrm{i}} \psi)) \wedge \bigwedge_i comp(FT_i) \\
FT \equiv \mathrm{acon}(\psi, \mathcal{FT}): & comp(FT) :\Leftrightarrow & (\bigwedge_i \neg\ (\neg\ \Diamond_{\mathrm{x}} f(FT_i) \mathbin{;} \Diamond_{\mathrm{i}} \psi)) \wedge \bigwedge_i comp(FT_i) \\
FT \equiv \mathrm{dis}(\psi, \mathcal{FT}): & comp(FT) :\Leftrightarrow & (\neg\ (\neg\ \Diamond_{\mathrm{x}} \bigvee_i f(FT_i) \mathbin{;} \Diamond_{\mathrm{i}} \psi)) \wedge \bigwedge_i comp(FT_i) \\
FT \equiv \mathrm{inh}(\psi, \chi, FT'): & comp(FT) :\Leftrightarrow & (\neg\ (\neg\ \Diamond_{\mathrm{x}} f(FT) \mathbin{;} \Diamond_{\mathrm{i}} \psi)) \wedge (\neg\ (\neg\ \Diamond_{\mathrm{x}} \chi \mathbin{;} \Diamond_{\mathrm{i}} \psi)) \\
 & & \wedge comp(FT')
\end{array}$$

Die Vollständigkeitsbedingung der Fehlerbaumgatter geht dabei als jeweils erstes Konjunktionsglied in die Definition der Vollständigkeit eines Fehlerbaums ein. Die Korrektheit $corr(FT)$ eines Fehlerbaums ist analog definiert. Die Semantik eines Fehlerbaums FT ist dann als die Menge von Intervallen definiert, die die Vollständigkeitsbedingung $comp(FT)$ des Fehlerbaums erfüllen.

Definition 5.14 *Semantik eines Fehlerbaums*

Die Semantik $[\![FT]\!]$ eines Fehlerbaums FT ist definiert als

$$[\![FT]\!] :\Leftrightarrow \{\mathcal{I} \mid \mathcal{I} \models comp(FT)\}$$

Betrachten wir einen Fehlerbaum als Formel, so gilt $\mathcal{I} \models FT :\Leftrightarrow \mathcal{I} \in [\![FT]\!]$. Die Semantik einer FTA ist als Konjunktion über alle Fehlerbäume $\mathcal{FT}$, die bei der FTA erstellt wurden, definiert.

Definition 5.15 *Semantik der FTA*

Für eine FTA, bei der die Fehlerbäume $\mathcal{FT}$ erstellt wurden, gilt:

$$[\![\text{FTA}]\!] :\Leftrightarrow \{\mathcal{I} \mid \mathcal{I} \models \bigwedge_i FT_i \textit{ für } FT_i \in \mathcal{FT}\}$$

Wollen wir zeigen, dass ein System, das durch eine temporallogische Formel φ spezifiziert ist, eine FTA erfüllt, so zeigen wir die Gültigkeit von $\varphi \rightarrow$ FTA oder von $\varphi \rightarrow comp(FT_i)$ für alle $FT_i \in FTA$. Erfüllt φ alle Vollständigkeitsbedingungen eines Fehlerbaums $\varphi \rightarrow comp(FT)$, so nennen wir den Fehlerbaum *vollständig* bezüglich φ. Für die Validierung von Fehlerbäumen definieren wir, dass ein Fehlerbaum *korrekt* bezüglich φ ist, wenn φ alle Korrektheitsbedingungen der Gatter eines Fehlerbaums FT erfüllt.

5.4 Das minimale Schnittmengen-Theorem

Die minimalen Schnittmengen einer FTA liefern einen wichtigen Beitrag zur Sicherheitsbewertung des untersuchten Systems. Wie in Abschnitt 2.2.2 beschrieben, sind die minimalen Schnittmengen sowohl für die qualitative als auch für die quantitative Analyse notwendig. Damit die Sicherheitsbewertung der klassischen FTA auf die formale FTA übertragen werden kann, müssen die Eigenschaften der minimalen Schnittmengen erhalten bleiben. Die wichtigste Eigenschaft ist

1. kann von jeder minimalen Schnittmengen ein Ereignis ausgeschlossen werden, tritt die Gefährdung nicht ein (*Vollständigkeit*).

Ist die Vollständigkeit eines Fehlerbaums erfüllt, kann der Fehlerbaum zum Sicherheitsnachweis für das betrachtete System eingesetzt werden. Für die Sicherheitseigenschaft „keine Gefährdung“ müssen nur die entsprechenden Ereignisse aus den minimalen Schnittmengen ausgeschlossen werden (oder eine geringe Eintrittswahrscheinlichkeit haben).

Eine weitere Eigenschaft zur Validation des Fehlerbaums ist

2. wenn alle Ereignisse einer minimalen Schnittmenge eintreten, tritt auch die Gefährdung ein (*Korrektheit*).

Die Korrektheit soll zeigen, dass die Ereignisse in den minimalen Schnittmengen auch die Gefährdung bedingen. Diese Eigenschaft erfüllen aber nur sehr wenige Fehlerbäume, denn das Hinzufügen von Sicherheitsmargen während der FTA führt gerade dazu, dass die Basisereignisse nicht immer die untersuchte Gefährdung verursachen. Deshalb formulieren wir das minimale Schnittmengen-Theorem für die Vollständigkeit und geben ein schwächeres Lemma für die Korrektheit an.

Die minimalen Schnittmengen eines formalen Fehlerbaums werden wie für eine herkömmliche FTA berechnet, wobei ein *Z-Und* bzw. ein (a)synchrones *UW-Und*-Gatter als *Und*-Gatter, ein *Z-Oder*- bzw. *UW-Oder*-Gatter als *Oder*-Gatter interpretiert werden.

Definition 5.16 *Minimale Schnittmengen der formalen FTA*

Für Fehlerbäume FT' und einer Menge von Fehlerbäumen $\mathcal{FT}$ berechnet die Funktion $mcs(FT)$[1], für $FT_i \in \mathcal{FT}$, die minimalen Schnittmengen für einen Fehlerbaum FT wie folgt:

$$\begin{array}{lll}
FT \equiv \mathrm{leaf}(\varphi): & mcs(FT) := & \{\{\varphi\}\} \\
FT \equiv \mathrm{Zcon}(\psi, \mathcal{FT}): & mcs(FT) := & \{\bigcup_i s_i \mid s_i \in mcs(FT_i)\} \\
FT \equiv \mathrm{Zor}(\psi, \mathcal{FT}): & mcs(FT) := & \bigcup_i mcs(FT_i) \\
FT \equiv \mathrm{Zinh}(\psi, \chi, FT'): & mcs(FT) := & mcs(FT') \\
FT \equiv \mathrm{scon}(\psi, \mathcal{FT}): & mcs(FT) := & \{\bigcup_i s_i \mid s_i \in mcs(FT_i)\} \\
FT \equiv \mathrm{acon}(\psi, \mathcal{FT}): & mcs(FT) := & \{\bigcup_i s_i \mid s_i \in mcs(FT_i)\} \\
FT \equiv \mathrm{or}(\psi, \mathcal{FT}): & mcs(FT) := & \bigcup_i mcs(FT_i) \\
FT \equiv \mathrm{inh}(\psi, \chi, FT'): & mcs(FT) := & mcs(FT')
\end{array}$$

Theorem 5.1 *minimale Schnittmengen*
Sei FT ein Fehlerbaum und $\psi := f(FT)$ die Gefährdung des Fehlerbaums.

Wenn $\mathcal{I} \models comp(FT)$ gilt und
es für alle $cs \in mcs(FT)$ ein $\varphi \in cs$ gibt, mit $\mathcal{I} \models \boxtimes \neg\, \varphi$,
dann gilt $\mathcal{I} \models \boxtimes \neg\, \psi$

Wenn in einem Fehlerbaum für die Gefährdung ψ die Vollständigkeit aller Gatter nachgewiesen werden kann, gilt: Sobald von jeder minimalen Schnittmenge ein Ereignis ausgeschlossen werden kann, tritt die Gefährdung ψ nicht ein. Für den Beweis verweisen wir auf Anhang B.

Die Vollständigkeit der minimalen Schnittmengen für die formalen FTA garantiert, dass es keine weiteren Ursachen für die untersuchte Gefährdung gibt. Damit kann die formale FTA auch zum Nachweis von Sicherheitseigenschaften eingesetzt werden. Mit dem Beweis der Vollständigkeit liefert die formale FTA einen bedingten aber strukturierten Nachweis für die Eigenschaft „keine Gefährdung" ($\neg\, \psi$). Die Nebenbedingung für „keine Gefährdung" ist, dass von jeder minimalen Schnittmenge mindestens ein Ereignis ausgeschlossen werden kann.

Lemma 5.2 *Korrektheit der minimalen Schnittmengen*
Sei FT ein Fehlerbaum ohne Block-, *synchrone* UW-Und *und* Z-Und-*Gatter, $\psi := f(FT)$ die Gefährdung des Fehlerbaums und $cs \in mcs(FT)$ eine minimale Schnittmenge. Es gilt:*

Wenn $\mathcal{I} \models corr(FT)$ und für alle $\varphi_i \in cs.\ \mathcal{I} \models \diamondtimes \varphi_i$,
dann gilt $\mathcal{I} \models \diamondtimes \psi$

Das Korrektheitstheorem besagt, dass sobald für alle Gatter eines Fehlerbaums die Korrektheit nachgewiesen ist und alle Ereignisse einer minimalen Schnittmenge auftreten, auch die Gefährdung ψ auftritt. Der Beweis des Korrektheitstheorems folgt ebenfalls in Anhang B.

[1] mcs: minimal cut sets.

Die Korrektheit der minimalen Schnittmengen gilt nur für eingeschränkte Fehlerbäume. *Block*-Gatter werden in einem Fehlerbaum explizit dann eingesetzt, wenn die Ursache nur unter bestimmten Nebenbedingungen zur Wirkung führt. Da wir diese Nebenbedingung nicht immer exakt definieren wollen, haben wir kein Korrektheitskriterium für *Block*-Gatter angegeben und deshalb können korrekte Fehlerbäume keine *Block*-Gatter enthalten.

Synchrone *UW-Und*- und *Z-Und*-Gatter verletzen ebenfalls das Korrektheitskriterium. Der Grund ist, dass die Ursachen beider Gatter gleichzeitig eintreten müssen, um die Wirkung auszulösen. Bei dynamischen Systemen kann für Zwischenereignisse nicht garantiert werden, dass deren Ursachen gleichzeitig eintreten, denn zwischen dem Eintreten der Blatt-Ereignisse und der Zwischenereignisse kann, bedingt durch *UW*-Gatter, beliebig viel Zeit vergehen.

Das Korrektheitstheorem für minimale Schnittmengen zeigt, dass sich die Korrektheit einzelner Gatter nur sehr bedingt auf den kompletten Fehlerbaum überträgt. Dieses Resultat bestätigt nochmal den Entschluss, die Korrektheitsbedingungen der Fehlerbaumgatter nicht in die Semantik der FTA mit einzubeziehen.

Die notwendige Voraussetzung für das Korrektheitstheorem der minimalen Schnittmengen stellt aber keine Einschränkung für die Anwendung der formalen FTA dar. Werden *Block*-, synchrone *UW-Und*- oder *Z-Und*-Gatter im Fehlerbaum verwendet, kann zwar nicht die Korrektheit des Fehlerbaums gezeigt werden, die minimalen Schnittmengen liegen aber „auf der sicheren Seite“, d. h. die Gefährdung tritt höchstens seltener als erwartet ein. Wir setzen den Korrektheitsnachweis der Gatter dann zum Validieren der Fehlerbäume ein. Kann die Korrektheit eines Gatters nicht gezeigt werden, sind bestimmte Ursachen für die Wirkung nicht relevant und müssen nicht weiter analysiert werden.

Die Beweise für das Theorem (5.2) und das Lemma (5.1) werden mit struktureller Induktion über den Aufbau eines Fehlerbaums geführt. Wir haben beide Theoreme im Spezifikations- und Verifikationswerkzeug KIV [BRS+00] nachgewiesen. Für eine ausführliche Beschreibung der Fallstudie und der Beweise verweisen wir auf den Anhang B.

5.5 Verwandte Arbeiten

Bei der Formalisierung der FTA konnten wir auf bereits vorhandene Arbeiten aufbauen. Die Ziele und Ansätze dieser Arbeiten diskutieren wir nun kurz und arbeiten die Unterschiede zu unserem Ansatz der formalen FTA heraus. Für einen ausführlichen Vergleich verschiedener Formalisierungen der FTA-Gatter verweisen wir auf Kapitel 8.

Hansen et al. [HRS94] formalisieren die FTA im Duration Calculus (DC, [ZHR91]) mit dem Ziel, aus der Fehlerbaumanalyse formale Spezifikationen abzuleiten. Ausgangspunkt sind Systemgefährdungen, die im DC formalisiert sind. Die FTA bricht die Gefährdungen auf (Sicherheits-) Anforderungen an Systemkomponenten herunter und folgt dabei der DC-Formalisierung. Die formalen Anforderungen bilden dann die Systemspezifikation. Bei unserer formalen FTA wird dagegen das formale Modell und die FTA parallel erstellt und nachgewiesen, dass die FTA vollständig bezüglich dem Modell ist.

Neben diesem methodischen Unterschied differieren auch die Semantiken. Da der DC

eine Erweiterung der ITL für kontinuierliche Systeme ist, können wir die Formalisierungen leicht vergleichen. Sehen wir von der kontinuierlichen Zeitachse ab, die dem DC zugrundeliegt, können wir die Ursache/Wirkungs-Gatter von Hansen et al. [HRS94] wie folgt in ITL formalisieren:

$$\boxed{\mathrm{i}}(\Diamond_{\mathrm{x}}\psi \rightarrow \Diamond_{\mathrm{x}}(\varphi_1 \wedge \varphi_2)) \text{ bzw. } \boxed{\mathrm{i}}(\Diamond_{\mathrm{x}}\psi \rightarrow \Diamond_{\mathrm{x}}(\varphi_1 \vee \varphi_2)) \tag{8}$$

Die Formalisierung verlangt, wenn irgendwann ($\Diamond_{\mathrm{x}}$) die Wirkung ψ eintritt, müssen auch irgendwann die Ursachen $\varphi_1 \wedge \varphi_2$ bzw. $\varphi_1 \vee \varphi_2$ eintreten. Da dies für alle initialen Intervalle gilt ($\boxed{\mathrm{i}}$) und insbesondere auch für das kürzeste Intervall, in dem die Wirkung eintritt, müssen die Ursachen vor der Wirkung liegen. Für temporale Wirkungen ist die Formalisierung zu schwach, denn sie verlangt nur, dass die Ursachen eintreten müssen, bevor die Wirkung beendet ist und nicht, dass die Ursachen eintreten, bevor die Wirkung beginnt. Wir finden diese Betrachtungsweise nicht intuitiv und haben deshalb die Formalisierung von Hansen et al. [HRS94] nicht übernommen (für eine detailliertere Betrachtung verweisen wir auf Kapitel 8).

Offenbar wurde diese Schwäche auch von den Autoren selbst bemerkt, denn in späteren Arbeiten [Han96, HRS98] verwenden sie nur noch eine boolesche Semantik für die Fehlerbaumgatter ($\boxtimes\ (\psi \rightarrow \varphi_1 \wedge \varphi_2)$ bzw. $\boxtimes\ (\psi \rightarrow \varphi_1 \vee \varphi_2)$), die diese Schwäche umgeht. Wie in Abschnitt 5.1 motiviert, genügen aber Zerlegungsgatter für die Analyse dynamischer Systeme nicht. Hansen et al. geben keine Korrektheitsbedingungen für die Fehlerbaumgatter an und definieren kein asynchrones *UW-Und*-Gatter. Dafür formalisieren sie ein priority *Und*-Gatter, bei dem die Ursachen in einer bestimmten Reihenfolge eintreten müssen.

Ein Theorem für die Vollständigkeit der erstellten Fehlerbäume, analog zum minimalen Schnittmengen-Theorem 5.1, wurde von Hansen et al. nicht angegeben. Wir haben jedoch in der Fallstudie in Anhang B gezeigt, dass auch die Formalisierung der Ursache/Wirkungs-Gatter von Hansen et al. das minimale Schnittmengen-Theorem erfüllt.

Auch Bruns und Anderson [BA93] definieren die Semantik für Fehlerbäume aufbauend auf den Fehlerbaumgattern. Sie benutzen Fehlerbäume um das Systemmodell zu validieren und weisen dazu nach, dass das Systemmodell die Gatterbedingungen erfüllt. Dieses Vorgehen ist analog zu unserem Ansatz, in dem die Fehlerbaumgatter Beweisverpflichtungen für das Systemmodell beschreiben.

Bruns und Anderson beschreiben drei verschiedene Semantiken für Fehlerbaumgatter im μ-Kalkül. Sie unterscheiden ebenfalls zwischen Zerlegungs- und Ursache/Wirkungs-Gatter. Die Formalisierung der Zerlegungsgatter nennen sie *immediate causality*. Die immediate causality entspricht der Konjunktion der Korrektheits- und Vollständigkeitsdefinition unserer Zerlegungsgatter. Für Ursache/Wirkungs-Gatter betrachten auch Bruns und Anderson die Formalisierung der Korrektheit und Vollständigkeit getrennt. Sie nennen die Korrektheit der Ursache/Wirkungs-Gatter *sufficient causality* und die Vollständigkeit *necessary causality*. Für die immediate causality ist ein Korrektheits- und Vollständigkeitstheorem entsprechend dem minimalen Schnittmengen-Theorem formuliert, für die Ursache/Wirkungs-Gatter jedoch nicht. Die Korrektheitsbedingungen sind, wie in unserem Ansatz, als starke Korrektheit definiert. Eine Semantik für *Block*-Gatter wurden aber weder für die

Zerlegungsgatter, noch für die Ursache/Wirkungs-Gatter angegeben, und es werden auch keine asynchronen *UW-Und*-Gatter definiert. Die Formalisierung der sufficient und necessary causality für Ursache/Wirkungs-Gatter haben bei temporalen Ereignissen eine ähnliche Schwäche wie die Formalisierung von Hansen et al., die wir ebenfalls in Kapitel 8 genauer betrachten.

Górski gibt in [Gór94] eine Semantik der FTA im sogenannten *Common Safety Description Model (CSDM)* [BCG91] an. Er benutzt Fehlerbäume als Spezifikation für die Korrektheitsbeziehungen zwischen Ereignissen. In dem prädikatenlogischen Ansatz werden die Ursache/Wirkungs-Beziehungen zwischen Ereignissen über die Start- und Endezeitpunkte der Ereignisse explizit durch den Fehlerbaum spezifiziert. Durch die explizite Beschreibung der Start- und Endezeitpunkte von Ereignissen können auch zeitliche Zusammenhänge zwischen Ereignissen in den minimalen Schnittmengen berechnet werden. Sind diese erfüllt, verursachen die Ereignisse der Schnittmengen die untersuchte Gefährdung, selbst wenn Ursache/Wirkungs-Gatter im Fehlerbaum verwendet wurden (vgl. mit Lemma 5.2, Korrektheit der minimalen Schnittmengen, in dem die Ursache/Wirkungs-Gatter ausgenommen sind). Ein Algorithmus für die Berechnung der zeitlichen Zusammenhänge zwischen den Basisereignissen ist in [GW97] gegeben.

Górski definiert die Korrektheit der Gatter ebenfalls als starke Korrektheit und berücksichtigt keine *Block*-Gatter. Deshalb führen die minimalen Schnittmengen immer zur untersuchten Gefährdung. Wie wir gesehen haben, reicht diese Formalisierung im Allgemeinen nicht aus, um beliebige Fehlerbäume betrachten zu können. In Abbildung 5.1 haben wir z. B. einen Fehlerbaum präsentiert, der die starke Korrektheit nicht für jedes Gatter erfüllt. Die Vollständigkeit von Fehlerbäumen wird bei Górski nicht betrachtet. Der Nachteil seiner Methode ist, dass mit den üblichen Spezifikationssprachen für dynamische Systeme (etwa Zustandsübergangssysteme), Start- und Endzeitpunkte von Ereignissen nicht spezifiziert werden können. Damit ist dieser Ansatz nur im CSDM anwendbar.

Schließlich gibt es noch Ansätze, die Fehlerbäume aus Systemspezifikationen automatisch erzeugen. Papadopoulos et al. stellt in [PMM+01, Pap03, PM01] einen Ansatz vor, in dem ein strukturelles Modell der Systemspezifikation durch Module und deren Verknüpfungen über Ein- und Ausgaben erstellt wird. Für jedes Modul wird eine Fehleranalyse durchgeführt. Die Fehleranalyse untersucht lokal im Modul, ob und wie sich falsche Eingaben im Modul zu den Ausgaben fortpflanzen und welche Fehler im Modul entstehen können. Ein Fehlerbaumgenarator erstellt aus den lokalen Sicherheitsanalysen und dem strukturellen Modell dann einen Fehlerbaum für das Gesamtmodell.

In einem europäischen Projektverbund *ESACS* [B+03, Con01] werden verschiedene Methoden untersucht, um aus formalen Modellen Fehlerbäume zu generieren. Dabei werden sowohl Techniken aus der Modellprüfung eingesetzt, um die Basisereignisse zu berechnen [BV03b, BV03a], als auch Erreichbarkeitsgraphen, aus denen die Pfade zur Gefährdung extrahiert werden [BCS02, Rau02]. Jedoch werden immer „flache“ Fehlerbäume erzeugt, die keine Zwischenereignisse enthalten.

Einen anderen Weg untersucht Damm et al. im ESACS-Projekt [Con02]. Systemmodelle mit Fehlern werden in STATEMATE [HLN+90] komponentenbasiert spezifiziert und es wird versucht, die Eigenschaft „keine Gefährdung“ mit dem Modellprüfer STVE (*STATEMATE*

Verification Environment, [BDW00]) nachzuweisen. Gelingt der Nachweis nicht, ergeben die bei der Verifikation entstehenden Gegenbeispiele Fehlerpfade zur Systemgefährdung. Aus diesen Fehlerpfaden und der Komponentenstruktur der Systemspezifikation wird nun ein Fehlerbaum mit Zwischenereignissen erstellt.

Zusammenfassend ist zu bemerken, dass eine Reihe von Semantiken für Fehlerbäume existieren, die für verschiedene Anwendungen entwickelt wurden. In einigen Ansätzen werden aus formalen Modellen Fehlerbäume automatisch erzeugt. Wir haben eine Semantik in ITL vorgestellt, die Schwächen einzelner Ansätze bei temporalen Ereignissen beseitigt und auf Systembeschreibungen beruht, die Sequenzen von Systemzuständen beschreibt. Die Korrektheits- und Vollständigkeitsbedingungen der einzelnen Gatter werden über diesem formalen Modell nachgewiesen. Die formale FTA definiert die Korrektheit für Gatter, wie die anderen Ansätze aus der Literatur auch, als starke Korrektheit. Um trotzdem Fehlerbäume behandeln zu können, die in bestimmten Schritten Sicherheitsmargen hinzufügen, benutzen wir *Block*-Gatter. Das minimale Schnittmengen-Theorem 5.1 und das Lemma über die Korrektheit der Schnittmengen 5.2 zeigt, dass wir eine adäquate Semantik der Fehlerbaumgatter definiert haben und sich die Vollständigkeit des Fehlerbaums aus der Vollständigkeit der einzelnen Fehlerbaumgatter ableitet.

KAPITEL 6

Verifikation von Fehlerbäumen

Den Nachweis der FTA-Bedingungen aus dem vorherigen Kapitel möchten wir durch ein Werkzeug unterstützen. KIV ist ein interaktives Verifikations- und Spezifikationswerkzeug, das insbesondere ITL-Eigenschaften nachweisen kann und daher bestens für die Validierung von Fehlerbäumen geeignet ist. Daten werden in KIV algebraisch spezifiziert und temporallogische Eigenschaften können sowohl für parallele Programme als auch für ITL-Spezifikationen formuliert und nachgewiesen werden. Die Grundidee der temporallogischen Verifikation ist *symbolisches Ausführen* und *Induktion*. Dabei werden parallele Programme, aber auch die ITL-Formeln, schrittweise abgearbeitet. Es entstehen prädikatenlogische Beweisverpflichtungen für die erste Programmanweisung und temporallogische Beweisverpflichtungen für das nun kürzere Restprogramm. Programme ohne Schleifen können so schrittweise abgearbeitet werden, für Programme mit Schleifen benötigt man zusätzlich Induktion. Diese Beweistechnik wird auch für ITL-Formeln benutzt [Bal04].

Für die Validierung von Fehlerbäumen integrieren wir Statecharts in KIV. Mit Statecharts spezifizieren wir dynamische Systeme, über denen wir die Vollständigkeitsbedinungungen der FTA nachweisen. Wir werden Statecharts, wie parallele Programme und ITL-Formeln, symbolisch ausführen und erweitern dazu den Beweis-Kalkül um entsprechende Kalkülregeln. Beim symbolischen Ausführen halten wir uns an die operationelle Semantik aus Abschnitt 4.2.2.

Datentypen, Funktionen und Prädikate, die in Statechart-Spezifikationen verwenden werden, spezifizieren wir algebraisch. Aktionen, die bei Zustandsübergängen ausgeführt werden, sind sequentielle Programme. Die Aufgabe der Vollständigkeitsprüfung von Fehlerbäumen ist, für eine algebraische Spezifikation SP, ein Statechart SC und einen Fehlerbaum FT die Frage zu beantworten, ob in allen Modellen der Spezifikation SP die Formel $SC \rightarrow FT$ gilt. Oder formal, ob $SP \models SC \rightarrow FT$. Die Antwort liefert eine Ableitung der Formel $SC \rightarrow FT$ bezüglich der Spezifikation SP, die wir mit Hilfe des interaktiven Beweisers KIV finden. Die algebraische Spezifikation SP wird in KIV-Beweisen im Hintergrund mitgeführt, während das mit Statecharts spezifizierte dynamische System eine Vorbedingung für die Gültigkeit der Fehlerbaumformeln $SC \rightarrow FT$ ist.

Durch die Verwendung algebraischer Datentypen in Statechart-Spezifikationen erhalten

wir eine integrierte Beschreibungssprache mit Beweisunterstützung, die weit über bisherige Ansätze hinaus geht. Bisherigen Ansätzen zur Integration funktionaler Spezifikationstechniken mit Statecharts steht keine Beweisunterstützung zur Verfügung. Beispielsweise kombiniert *μ-SZ* [Gei99] Z-Spezifikationen mit Statechart-Spezifikationen. Ein algebraischer Ansatz existiert mit Casl-Chart [RR00]. Auf der anderen Seite gibt es eine Reihe von Werkzeugen für die Statechart-Verifikation. Verschiedene Modellprüfer (siehe z. B. [BDW00, LMS97, PS98]) weisen Eigenschaften von Statecharts automatisch nach. Modellprüfung ist aber nur für zustandsendliche Systeme mit endlichem Datenbereich möglich. Mit der interaktiven Verifikationsmöglichkeit von Statecharts über algebraischen Spezifikationen erhalten wir eine Kombination funktionaler und dynamischer Spezifikationen mit einer Beweisunterstützung für ITL-Formeln. Wir können damit Statechart-Beweise über nicht zustandsendliche Systeme führen. Mit *nicht zustandsendlich* bezeichnen wir hier Systeme, die Datentypen mit unendlichem Wertebereich verwenden.

6.1 Das Verifikationswerkzeug KIV

KIV [BRS+00] ist ein interaktives Spezifikations- und Verifikationswerkzeug, basierend auf algebraischen Spezifikationen [Wir90]. Darauf aufbauend können sequentielle Programme [RSS95, Rei95], objektorientierte JavaCard-Programme [Ste01], abstrakte Zustandsmaschinen (ASMs, [SA97, Sch99]), parallele Programme [BDRS02] und temporallogische Spezifikationen [Bal04] erstellt und Eigenschaften dafür nachgewiesen werden. Um Spezifikationen zu strukturieren, werden die Strukturierungsoperatoren *enrichment*, *union*, *renaming*, *parameterization* und *actualisation* eingesetzt [RSSB98]. KIV basiert auf einer Logik, welche Logik höherer Stufe [And86], die Dynamische Logik (DL, [Har84]) und die Intervalltemporallogik (ITL, [Mos85]) kombiniert. Beweise werden in KIV im Sequenzenkalkül [SA91] geführt.

6.1.1 Algebraische Spezifikation

Eine algebraische Spezifikation $SP = (Sig, T, C)$ besteht aus der Signatur $Sig = (S, OP)$, den Axiomen $T \subseteq Fma_{PL}(Sig, V)$ und den Konstruktoren $C \subseteq F$. Dabei ist S eine endliche Menge von Sorten, $OP = F \cup P$ eine Menge mit den Funktionssymbolen F und den Prädikatsymbolen P. Wir nehmen an, dass die Sorte *bool* in S enthalten ist und die Operatoren true, false, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$ und $\neg$ definiert sind und ihre übliche Semantik besitzen. $Fma_{PL}(Sig, V)$ ist die Menge der Formeln erster Ordnung, die über der Signatur Sig und den Variablen V gebildet werden können. Eine Algebra $\mathcal{A}$ heißt erzeugt, $\mathcal{A} \in Gen(Sig, C)$, falls es für jedes Element a der Trägermenge von $\mathcal{A}$ einen Grundterm τ mit $a = \tau_{\mathcal{A}}$ gibt, wobei $\tau_{\mathcal{A}}$ der Wert des Terms τ in $\mathcal{A}$ ist. Wir vereinbaren, dass wir im Folgenden die Bezeichner x, y und v für Variablen und X und V für Mengen von Variablen verwenden.

Wir werten Formeln über einer Algebra $\mathcal{A}$ und Intervallen $\mathcal{I} = (\sigma_0, \sigma_1, \ldots)$, einer (un)endlichen Sequenz von Belegungen, aus. Eine Belegung $\sigma : V \rightarrow \mathcal{D}$ ist eine Zuordnung der Variablen $v \in V$ zu Werten aus ihrem Wertebereich $\mathcal{D}$. Eine prädikatenlogische Formel

$\Phi \in Fma_{PL}(Sig, V)$ ist wahr, d. h. $\mathcal{A},\mathcal{I} \models_{ITL} \Phi$, wenn für $\mathcal{A}$ und der ersten Belegung $\mathcal{I}(0)$ von $\mathcal{I}$ mit der herkömmlichen Semantikdefinition für Formeln erster Stufe $\mathcal{A},\mathcal{I}(0) \models_{PL} \Phi$ gilt. $\mathcal{A}$ und $\mathcal{I}$ sind ein Modell von T, d. h. $\mathcal{A},\mathcal{I} \models_{ITL} T$, wenn in $\mathcal{A}$ und $\mathcal{I}$ die Axiome T gelten. Die Modelle $Mod(SP) = \{(\mathcal{A},\mathcal{I}) \mid \mathcal{A} \in Gen(Sig, C)$ und $\mathcal{A},\mathcal{I} \models_{ITL} T\}$ der Spezifikation SP sind diejenigen Paare $(\mathcal{A},\mathcal{I})$, deren Algebren erzeugt sind und in denen die Axiome gelten.

Sei $Fma_{ITL}(Sig, V)$ die Menge der intervalltemporallogischen Formeln. Eine Formel $\varphi \in Fma_{ITL}(Sig, V)$ gilt in einer algebraischen Spezifikation SP, d. h. $SP \models_{ITL} \varphi$, wenn für alle Algebren und Intervalle $(\mathcal{A},\mathcal{I}) \in Mod(SP)$ gilt: $\mathcal{A},\mathcal{I} \models_{ITL} \varphi$. Wie üblich lassen wir die Algebra $\mathcal{A}$ weg, wenn sie für das Verständnis nicht wichtig ist, schreiben statt $\mathcal{I} \models_{ITL} \varphi$ bzw. $\mathcal{I}(0) \models_{PL} \varphi$ einfach $\mathcal{I} \models \varphi$ bzw. $\mathcal{I}(0) \models \varphi$, wenn die Bedeutung von $\models$ klar ist und bezeichen die Formelmengen $Fma_{PL}(Sig, V)$ und $Fma_{ITL}(Sig, V)$ mit Fma_{PL} bzw. Fma_{ITL}. Man beachte, dass eine Spezifikation SP nur prädikatenlogische Teile beschreibt ($T \subseteq Fma_{PL}$). Temporallogische Eigenschaften, die mögliche Folgen von Belegungen einschränken, werden Vorbedingungen in Formeln (siehe dazu Abschnitt 6.1.4).

6.1.2 Prädikatenlogik

Prädikatenlogische Beweise werden in KIV in einem Sequenzenkalkül geführt. Wenn $\Gamma = \{\varphi_1, \ldots, \varphi_n\}$ und $\Delta = \{\psi_1, \ldots, \psi_m\}$ Formelmengen sind, hat eine *Sequenz* $\Gamma \vdash \Delta$ hat die Bedeutung $\forall_{Cl}(\varphi_1 \land \ldots \land \varphi_n \rightarrow \psi_1 \lor \ldots \lor \psi_m)$, und wir schreiben kurz $\Gamma \rightarrow \Delta$. Wir nennen Γ den *Antezedenten* und Δ den *Sukzedenten* einer Sequenz. Eine *Sequenzenkalkülregel* hat die Form $\frac{p_1, \cdots p_n}{c}$. c ist die *Konklusio* und p_i sind die *Prämissen* der Regel. Sowohl die Konklusio c als auch die Prämissen p_i einer Regel sind Sequenzen. Beweise orientieren sich im Sequenzenkalkül an den Operatoren und „vereinfachen“ die Sequenz, bis Axiome erreicht werden, die den Beweis schließen. Durch wiederholtes Anwenden von Regeln auf die Prämissen entsteht ein Beweis- oder Ableitungsbaum. Im Allgemeinen gibt es pro Operator eine Regel für den Antezedenten und eine für den Sukzedenten. Als Beispiel sehen wir die beiden Regeln für die Konjunktion.

$$\frac{\varphi, \psi, \Gamma \vdash \Delta}{\varphi \land \psi, \Gamma \vdash \Delta}\ \textit{conj. left} \qquad \frac{\Gamma \vdash \varphi, \Delta \quad \Gamma \vdash \psi, \Delta}{\Gamma \vdash \varphi \land \psi, \Delta}\ \textit{conj. right}$$

6.1.3 Dynamische Logik

Die Dynamische Logik [Har84] ist eine Erweiterung der Prädikatenlogik um die Operatoren $[\alpha]\varphi$ (Box) und $\langle\alpha\rangle\varphi$ (Diamond), wobei α ein Pascal-artiges Programm und φ eine prädikatenlogische Formel ist. Für ein Programm α bedeutet:

$[\alpha]\varphi$: wenn α terminiert, gilt nach Ausführung von α die Bedingung φ

$\langle\alpha\rangle\varphi$: α terminiert, und es gilt nach der Ausführung von α die Bedingung φ

Ein DL-Programm α beschreibt eine Relation zwischen zwei Belegungen. Die Syntax und entsprechende I/O-Semantik findet sich im Anhang E. Zusätzlich zu den Sequenzenkalkülregeln gibt es für jedes Programmkonstrukt entsprechende DL-Kalkülregeln. Als Beispiel präsentieren wir die bedingte Verzweigung.

$$\frac{\langle\alpha\rangle\varphi, \Gamma, \varepsilon \vdash \Delta \quad \langle\beta\rangle\varphi, \Gamma, \neg\varepsilon \vdash \Delta}{\langle \textbf{if } \varepsilon \textbf{ then } \alpha \textbf{ else } \beta\rangle\varphi, \Gamma \vdash \Delta} \textit{ if left} \qquad \frac{\Gamma, \varepsilon \vdash \langle\alpha\rangle\varphi, \Delta \quad \Gamma, \neg\varepsilon \vdash \langle\beta\rangle\varphi, \Delta}{\Gamma \vdash \langle \textbf{if } \varepsilon \textbf{ then } \alpha \textbf{ else } \beta\rangle\varphi, \Delta} \textit{ if right}$$

Die Vereinfachung besteht bei DL-Beweisen darin, das Programm schrittweise auszuführen. Z. B. führen die *if*-Regeln die *if*-Anweisung auf den *then*-Zweig zurück, falls die Bedingung ε gilt, ansonsten auf den *else*-Zweig. Eine Zuweisung $x := \tau$ ersetzt in der Nachbedingung φ x durch den Wert τ.

$$\frac{\varphi_x^\tau, \Gamma \vdash \Delta}{\langle x := \tau\rangle\varphi, \Gamma \vdash \Delta} \textit{ assign left} \qquad \frac{\Gamma \vdash \varphi_x^\tau, \Delta}{\Gamma \vdash \langle x := \tau\rangle\varphi, \Delta} \textit{ assign right}$$

Bei einer *zufälligen Zuweisung* $x := ?$ wird x durch eine neue Variable, die in der Sequenz nicht vorkommt, ersetzt und erhält dadurch einen beliebigen Wert. Zuweisungsregeln reduzieren also ein Programm auf prädikatenlogische Aussagen.

6.1.4 Intervalltemporallogik

Als Grundlage für die Intervalltemporallogik in KIV [Bal04] dient eine Intervalltemporallogik [Mos85] erster Ordnung. Die ITL-Semantik basiert auf Intervallen $\mathcal{I}$ (auch Traces genannt), die eine endliche oder unendliche Sequenz von Zuständen oder Belegungen $\mathcal{I} = (\sigma_0, \ldots)$ beschreiben. Jede Belegung σ_i bildet Variablen auf Werte ihres Wertebereichs ab. Es gibt sowohl rigide als auch flexible Variablen, wohingegen Funktions- und Prädikatssymbole immer rigide sind und algebraisch spezifiziert werden. Flexible Variablen können in jeder Belegung eines Intervall einen anderen Wert haben, während rigide Variablen in allen Belegungen eines Intervalls einen festen Wert haben. Um flexible von rigiden Variablen zu unterscheiden, kennzeichnen wir, falls notwendig, flexible Variablen durch einen Punkt (z. B. $\dot{x}$).

Als Temporaloperatoren benutzen wir u. a. φ **;** ψ (chop), $\Box\varphi$ (always), $\Diamond\varphi$ (eventually), $\varphi \mathrm{~U~} \psi$ (until), $\circ\, \varphi$ (strong next) und $\bullet\, \varphi$ (weak next, im Gegensatz zu $\circ$ muss keine nächste Belegung existieren). Das Prädikat **last** gilt in der letzten Belegung eines endlichen Intervalls. Die Semantik der ITL-Operatoren in KIV entspricht der üblichen Semantik, die wir in Abschnitt 5.2 für die Formalisierung der Fehlerbäume bereits definiert haben.

Intervalle sind in KIV nicht an ein Übergangssystem gebunden. Die Übergänge zwischen Belegungen in Intervallen werden auch durch temporallogische Formeln beschrieben. Z. B. beschreibt $\Diamond\Box\varphi$ die Menge der Intervalle, in denen irgendwann immer φ gilt. Für diese Intervalle gilt, dass zu jedem Zeitpunkt irgendwann φ gilt, also $\Box\Diamond\varphi$. Im Sequenzenkalkül zeigt man dazu $\Diamond\Box\varphi \vdash \Box\Diamond\varphi$. Die Systembeschreibung steht als Voraussetzung im Antezedenten, die zu zeigende Eigenschaft im Sukzedenten. In KIV werden auch parallele Programme als Formeln zur Beschreibung eines Übergangssystems betrachtet [BDRS02].

Flexible Variablen

Für jede flexible Variable $\dot{x}$ gibt es Variablen x' und x''. Die Idee der 1-fach gestrichenen Variablen x' stammt aus der TLA [Lam94], in der x' die Belegung der Variablen $\dot{x}$ im Nachfolgezustand beschreibt. Durch Prädikate über gepunkteten und 1-fach gestrichene Variablen werden in TLA die Übergänge zur Nachfolgebelegung beschrieben. Die Relation zwischen $\dot{x}$ und x' beschreibt einen Systemübergang. Beispielsweise beschreibt $x' = \dot{x} + 1$, dass in der nachfolgenden Belegung die Variable $\dot{x}$ um 1 erhöht ist.

In KIV betrachten wir offene Systeme, deren Variablen auch vom Kontext, in dem das System betrachtet wird, beeinflusst werden können. Deshalb kann nach einem Systemübergang der Kontext Variablen noch beliebig abändern. Zur Beschreibung der Änderungen durch den Kontext verwenden wir eine Relation zwischen 1-fach und 2-fach gestrichenen Variablen. Das Systemverhalten wird weiterhin durch die Beziehung zwischen gepunkteten und 1-fach gestrichenen Variablen beschrieben. In KIV repräsentieren also die 2-fach gestrichenen Variablen den Wert der gepunkteten Variablen der nächsten Belegung. Die Formel $x' = \dot{x}+1 \wedge x' = x''$ beschreibt, dass das System $\dot{x}$ um eins erhöht und der Kontext x' nicht weiter ändert. Im Nachfolgezustand ist $\dot{x}$ um 1 erhöht.

Die Belegungen eines Intervalls $\mathcal{I} = (\sigma_0, \ldots)$ weisen also neben den gepunkteten Variablen auch den 1-fach und 2-fach gestrichenen Variablen einen Wert aus ihrem Wertebereich zu ($\sigma_i(\dot{x})$, $\sigma_i(x')$, bzw. $\sigma_i(x'')$). Für alle Variablen x gilt: $\dot{x}$, x' und x'' haben den selben Wertebereich und $\sigma_i(x'') = \sigma_{i+1}(\dot{x})$.

Kalkül

Die Idee des ITL-Kalküls besteht darin, ITL-Formeln so zu zerlegen, dass prädikatenlogische Beweisverpflichtungen für die aktuelle Belegung des Intervalls und temporallogische Beweisverpflichtungen für das Restintervall entstehen. Wir nennen dies *Abwickeln* von ITL-Operatoren, das dem Ausführen von Programmen entspricht. Für jeden ITL-Operator gibt es zwei Kalkülregeln. Exemplarisch geben wir die Regeln für den $\Box$-Operator an.

$$\frac{\varphi,\ \bullet\,\Box\varphi, \Gamma \vdash \Delta}{\Box\varphi, \Gamma \vdash \Delta}\ \textit{always left} \qquad\qquad \frac{\Gamma \vdash \varphi, \Delta \quad \Gamma \vdash\ \bullet\,\Box\varphi, \Delta}{\Gamma \vdash \Box\varphi, \Delta}\ \textit{always right}$$

Die *always left*-Regel etabliert für den aktuellen Zustand die Voraussetzung φ und, dass für das Restintervall die Voraussetzung immer gilt. Die *always right*-Regel zerlegt $\Box\varphi$ im Sukzedenten einer Sequenz in zwei Prämissen. In der linken Prämisse ist nachzuweisen, dass φ im aktuellen Zustand gilt, und in der rechten Prämisse, dass die Eigenschaft auch für das restliche Intervall gilt.

Die ITL-Operatoren werden nun solange abgewickelt, bis jede temporallogische Formel in Γ und Δ mit einem *Next*-Operator beginnt und alle anderen Formeln γ und δ prädikatenlogisch sind. Die prädikatenlogischen Teile werden mit den herkömmlichen DL-Sequenzenkalkülregeln vereinfacht. Der temporallogische Teil spricht nun nur noch über die Nachfolgebelegung. D. h. wir können im Intervall fortschreiten und die Regel *tl step*

(ähnlich zur *next*-Regel in [BDK92]) anwenden.

$$\frac{\gamma_0, \Gamma \vdash \delta_0, \Delta}{\gamma,\ \circ\,\Gamma \vdash \delta,\ \bullet\,\Delta}\ \textit{tl step}$$

Sie entfernt die führenden *Next*-Operatoren, kopiert die Werte der ungestrichenen und 1-fach gestrichenen Variablen in neue Variablen (in γ_0 und δ_0) und die der 2-fach gestrichenen Variablen in die entsprechenden ungestrichenen. Damit befindet sich der Beweis einen Schritt im Intervall weiter und die Variablen haben den Wert der 2-fach gestrichenen Variablen der vorigen Belegung.

Induktion

Durch Abwickeln der ITL-Operatoren und Ausführen von Schritten „arbeitet“ man endliche Intervalle nach und nach ab und reduziert temporallogische Beweisverpflichtungen auf prädikatenlogische. Für unendliche Intervalle benötigt man zusätzlich Induktion. Die grundsätzliche Idee der Induktion ist, dass wir das Intervall abarbeiten, bis wir eine Situation erreichen, die wir während des Beweises schon einmal betrachtet haben. Wir haben eine Schleife durchlaufen. Wenn wir eine invariante Aussage zeigen können, die vor und nach der Schleife gilt, können wir den Beweis für das gesamte Intervall mit dem Induktionsargument schließen.

Induktion erfolgt über die Länge des Intervalls. Für unendliche Intervalle benutzen wir eine Erreichbarkeitsaussage $\Diamond\varphi$, die nach endlich vielen Schritten n zum ersten Mal gilt. Wir führen noethersche Induktion über die Anzahl der Schritte n bis zum Erreichen von φ. Arbeiten wir nach Induktionsbeginn das Intervall ab und φ gilt nun in $< n$ Schritten, können wir die Induktionshypothese anwenden. Dies gilt solange, bis wir φ erreicht haben. In dieser Arbeit benötigen wir Induktion für Sicherheitsaussagen $\Gamma \vdash \Box\varphi$. Mit $\Box\varphi = \neg \Diamond\neg\, \varphi$ erhalten wir einen Erreichbarkeitsanteil im Antezedenten ($\Diamond\neg\, \varphi, \Gamma \vdash$) und können mit der Regel

$$\frac{(\ \bullet \bigwedge\Gamma \rightarrow \Box\varphi \vee \bigvee\Delta)\ \mathrm{U} \neg\, \varphi, \Gamma \vdash \Box\varphi, \Delta}{\Gamma \vdash \Box\varphi, \Delta}\ \text{ind always}$$

Induktion über die Anzahl der Schritte, bis zum ersten Mal $\neg\, \varphi$ gilt, führen. Die semantischen Grundlagen und den Kalkül für die interaktive Verifikation von ITL-Eigenschaften in KIV wurden von Balser entwickelt und für Details verweisen wir auf [Bal04].

Parallele Programme in ITL

Parallele Programme sind in KIV ebenfalls temporallogische Formeln. Semantisch sind parallele Programme in ITL dadurch integriert, dass sie, analog zu ITL-Formeln (un)endliche Folgen von Belegungen definieren. Die Beweistechnik der symbolischen Ausführung wird in KIV auch für parallele Programme angewendet. Dazu wird die erste Anweisung abgespalten und ausgeführt und in der Nachfolgebelegung das Restprogramm betrachtet. Parallele

Programme können u. a. das leere Programm **skip**, die Zuweisungen $x := \tau$, die sequentielle Komposition $\alpha;\beta$, die bedingte Verzweigung **if** ε **then** α **else** β und die parallele Ausführung $\alpha \mid\mid \beta$ sein.

Für die symbolische Ausführung berechnet die Funktion $\mathcal{T}_X$, in Abhängigkeit von den Programmvariablen X, eine Menge möglicher Anweisungen, die ausgeführt werden können. Die Anweisungen ändern die 1-fach gestrichenen Variablen, so dass nach einem Programm-Schritt noch das Kontextverhalten einfließen kann. Zu jeder Anweisung berechnet $\mathcal{T}_X$ das jeweilige Restprogramm, das nach Abarbeiten der Anweisung noch auszuführen ist.

$$\begin{array}{rcl} \mathcal{T}_X & : & \text{Prog} \rightarrow \wp((\text{Fma},\ \text{Prog})) \\ \mathcal{T}_X(x := \tau) & := & \{(x' = \tau \wedge \bigwedge_{v \in X \setminus \{x\}} v' = v, \mathbf{skip})\} \\ \mathcal{T}_X(\alpha;\beta) & := & \mathcal{T}_X(\alpha);\beta \\ \mathcal{T}_X(\mathbf{if}\ \varepsilon\ \mathbf{then}\ \alpha\ \mathbf{else}\ \beta) & := & \{(\varepsilon, \alpha), (\neg\varepsilon, \beta)\} \\ \mathcal{T}_X(\alpha \mid\mid \beta) & := & (\mathcal{T}_X(\alpha) \mid\mid \beta) \cup (\alpha \mid\mid \mathcal{T}_X(\beta))) \end{array}$$

Die *exec parprog*-Regel spaltet dann eine Anweisung φ_α eines parallelen Programms α ab und führt sie aus. Im nächsten Zustand ist noch das Restprogramm α' abzuarbeiten.

$$\frac{\bigvee\{(\varphi_\alpha \wedge \circ\, \alpha') \mid (\varphi_\alpha, \alpha') \in \mathcal{T}_X(\alpha)\}, \Gamma \vdash \Delta}{\alpha, \Gamma \vdash \Delta}\ \text{exec parprog}$$

Mit der *tl step*-Regel gehen wir nach der Ausführung des Programms in die Nachfolgebelegung über und führen dann das Restprogramm aus. Details zur Behandlung paralleler Programme in KIV finden sich in [BDRS02].

Einbettung von DL in ITL

Eine Besonderheit der Logik in KIV ist, dass auch DL-Operatoren in ITL-Formeln verwendet werden können. DL-Operatoren beschreiben komplexe Beziehungen zwischen Variablen, die durch sequentielle Programme ausgedrückt werden. Im folgenden Beispiel fordern wir, dass der Wert der Variablen x'' gleich dem Wert der Variablen x nach Programmausführung ist.

$$\langle \mathbf{if}\ y = 0\ \mathbf{then}\ x := 1\ \mathbf{else}\ x := 2; x := x^2\rangle\ x'' = x \qquad (1)$$

Das Programm beschreibt einen Übergang, der nicht mehr vom Kontext beeinflusst wird. Man beachte, dass (1) ein DL-Programm und *kein* paralleles Programm ist. Es beeinflusst nur den aktuellen Systemzustand, obwohl mehrere Zuweisungen hintereinander ausgeführt werden. Abhängig von y ist x'' entweder 1 oder 4. Wir werden später bei der Schrittberechnung von Statecharts diese Art, Variablenwerte durch DL-Programme zu berechnen und dann in entsprechend gestrichene Variablen zu kopieren, nutzen, um Nachfolgekonfigurationen zu bestimmen. Neben einfachen Zuweisungen benutzen wir in der Schrittberechnung auch parallele Zuweisungen '|'. Es gilt:

$$x = 1 \wedge y = 2 \rightarrow \langle x := y \mid y := x\rangle\ x = 2 \wedge y = 1$$

y erhält den Wert von x *vor* der parallelen Zuweisung. Ein DL-Programm ändert in ITL die erste Belegung σ_0 eines Intervalls $(\sigma_0, \sigma_1, \ldots)$ ab. Die folgenden Belegungen $(\sigma_1, \ldots)$ bleiben unverändert. Die Semantik eines DL-Programms ist:

$$(\sigma_0, \sigma_1, \ldots) \models \langle\alpha\rangle\ \varphi \quad :\Leftrightarrow \quad \text{es existiert ein } \tau \text{ mit } \sigma_0[\![\alpha]\!]\tau \text{ und } (\tau, \sigma_1, \ldots) \models \varphi.$$

wobei $\sigma_0[\![\alpha]\!]\tau$ der I/O-Semantik des Programms α mit der Eingabebelegung σ_0 und der Ausgabebelegung τ entspricht (siehe Anhang E).

6.2 Statechart-Spezifikationen in KIV

Wir möchten Statecharts analog zu parallelen Programmen behandeln. Beide beschreiben Traces oder Intervalle, und die Verifikation von Statecharts soll dem bewährten Konzept der symbolischen Ausführung folgen. Um die Statechart-Verifikation in KIV zu ermöglichen, benötigen wir erstens eine Spezifikationsmöglichkeit für Statecharts, zweitens eine semantische Integration in ITL und schließlich die Erweiterung des ITL-Kalküls um Regeln für die Statechart-Verifikation.

6.2.1 Syntax

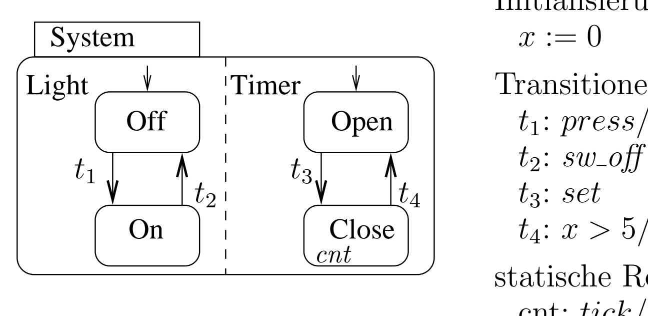
Die konkrete KIV-Syntax für die Beschreibung von Statecharts ist ein strukturierter Text mit Schlüsselworten. Die KIV-Spezifikation in Abbildung 6.1 unten beschreibt eine einfache Zeitschaltuhr für eine Lampe, die der darüber abgebildeten graphischen Repräsentation entspricht. Einmal eingeschaltet, leuchtet die Lampe für fünf Minuten/Zeiteinheiten. *tick* kennzeichnet einen Makro-Schritt und wird dazu benutzt, den Ablauf einer Zeiteinheit zu signalisieren.

Eine Statechart-Spezifikation wird in KIV durch **chart specification** eingeleitet. Danach können beliebig viele **BasicChart**s, **OrChart**s bzw. **AndChart**s spezifiziert werden.

In Abbildung 6.1 wird zuerst ein **BasicChart** *Close* spezifiziert. **variables** definiert eine lokale Variable x, die (beim Systemstart) mit 0 initialisiert wird (**initial values**). Sie wird durch die statische Reaktion **static reaction** *tick* | ***begin*** *x := x +1* ***end*** bei jedem Auftreten von *tick* um eins erhöht. *tick* ist implizit definiert und kennzeichnet einen Makro-Schritt. Allgemein hat eine statische Reaktion die Form $e \mid \alpha$ und ist aktiviert, wenn der boolesche Ausdruck e zu true evaluiert wird. Dann führt sie das Programm α aus.

Das **OrChart** *Timer* besteht aus einem einfachen Basiszustand *Open* (nach '=') und dem Unterzustand *Close*. Der Basiszustand *Open* ist implizit definiert und führt weder statische Reaktionen aus noch definiert er Variablen oder Ereignisse. Im Gegensatz dazu wird mit **subcharts** *Close*[1] die komplette Definition von *Close* in *Timer* integriert. Dies umfasst sowohl die Definition der Variablen x als auch die statische Reaktion, die in *Close* ausgeführt wird. Zusätzlich wird nach **import events** das Umgebungsereignis *set* definiert und nach **events** das lokale Ereignis *sw_off*. Mit **initial state** wird der Initialzustand *Open*

[1] Im Allgemeinen können mehrere Charts, durch '+' getrennt, angegeben werden.



Initialisierung:
$x := 0$

Transitionen:
t_1: $press/set$
t_2: sw_off
t_3: set
t_4: $x > 5/sw_off; x := 0$

statische Reaktionen:
cnt: $tick/x := x + 1$

```
chart specification
  BasicChart Close;
    variables x : nat;
    initial values x = 0;
    static reactions
      tick | begin x := x +1 end;
  OrChart Timer = Open subcharts Close;
    import events set;
    events        sw_off;
    initial state Open;
    transitions
      Open → set | → Close;
      Close → x > 5 | begin sw_off := true ; x := 0 end → Open;
  OrChart Light = Off + On;
    import events sw_off, press;
    events        set;
    initial state Off;
    transitions
      Off → press | begin set := true end → On;
      On → sw_off | → Off;
  AndChart System = Light | Timer;
    import events press;
end chart specification
```

Abbildung 6.1: Statechart Spezifikation in KIV

des *Oder*-Zustands definiert. Nach **transitions** folgt eine Liste von Übergängen. Ist der Ausgangszustand des Übergangs *Open* aktiv und das Ereignis *set* vorhanden, wird der Zustand *Close* betreten. Die allgemeine Form eines Übergangs ist $s_1 \to e \mid \alpha \to s_2$ mit s_1 als Ausgangszustand und s_2 als Endzustand.

Das **AndChart** *System* schaltet die beiden Zustände *Light* und *Timer* parallel und definiert das Umgebungsereignis *press*. Alle anderen Variablen, die in den Unterzuständen definiert wurden, sind nun lokale Variablen. Dies gilt auch für *sw_off* im Zustand *Light*. Da *System* die gesamte Definition der Untercharts importiert, wird auch die lokale Definition von *sw_off* importiert.

Zusätzlich können noch abgeleitete Ereignisse, die das Betreten und Verlassen von Zuständen s kennzeichnen, in Übergängen benutzt werden. Um sie von Prädikaten unterscheiden zu können, schreiben wir statt der STATEMATE-Notation $en(s)$ bzw. $ex(s)$ in KIV *en_s* bzw. *ex_s*. Weiterhin berücksichtigen wir Terminierungszustände, die mit *term* bezeichnet werden und den Ablauf eines Statecharts stoppen, wenn sie aktiviert werden. Beide Konstrukte werden im Beispiel in Abbildung 6.1 nicht verwendet. Die vollständige Grammatik der KIV-Syntax für Statecharts findet sich im Anhang F.

6.2.2 Timeout-Ereignisse und Schedule-Aktionen

Wir unterstützen Timeout-Ereignisse und Schedule-Aktionen (siehe Abschnitt 4.1.1, Seite 31) nicht explizit. Durch das Kennzeichnen eines Makro-Schrittes mit dem *tick*-Ereignis können wir jedoch beide Konstrukte modellieren.

Wir modellieren ein Timeout-Ereignis $tm(e, n)/\alpha$, welches n-Zeiteinheiten nach dem Auftreten des Ereignisses e die Aktion α ausführt, wie in Abbildung 6.2 zu sehen, mit zwei statischen Reaktionen (aufeinanderfolgende statische Reaktionen werden durch ; getrennt). Statt einer Variablen n können wir auch eine Funktion f mit dem Wertebereich der natürlichen Zahlen verwenden. Die natürlichen Zahlen werden dazu algebraisch spezifiziert (siehe Spezifikation *nat-basic1*, Seite 250). Zum Zählen der Zeitschritte führen wir die Variable

$tm(e,n)/\alpha$ $\rightsquigarrow$

$e/n_e := n;$ $act_e :=$ true; $tick/$if $n_e > 0$ then $n_e := n_e - 1$

$n_e = 0 \wedge act_e/$ $\alpha;\ act_e :=$ false

Abbildung 6.2: Modellierung von Timeout-Ereignissen

n_e ein, die mit 0 initialisiert wird. Sobald das Ereignis e eintritt, setzen wir act_e, n_e auf n und erniedrigen n_e in jedem Makro-Schritt um 1. Sobald $n_e = 0$ ist und act_e gilt, führen wir den Übergang aus und setzen act_e wieder false. Diese Modellierung garantiert, dass bei erneutem Eintreten von e, während n_e heruntergezählt wird, der Timer zurückgesetzt wird und n_e wieder auf n gesetzt wird. act_e benötigen wir, um nicht ohne Eintreten des Ereignisses e den Übergang auszuführen.

Schedule-Aktionen $e/sc(a,n)$ verhalten sich wie Timeout-Ereignisse, nur können sie nicht zurückgesetzt werden. Wir definieren ebenfalls eine Zählervariable n_e, die jedoch nicht durch $n_e := n$ zurückgesetzt werden darf, wenn n_e gerade heruntergezählt wird. Deshalb setzen wir n_e nur auf n, wenn $n_e \neq 0$ ist. Die statische Reaktion zum Aktivieren einer Schedule-Aktion lautet dann $e/\text{if } n_e = 0 \text{ then } n_e := n$. Ansonsten übernehmen wir die Modellierung von Timeout-Ereignissen. Die statischen Reaktionen werden jedoch nicht dem Ausgangszustand des Übergangs t, auf dem die Schedule-Aktion ausgeführt wird, zugeordnet, sondern dem Zustand $s = scope(t)$, in dessen Region der Übergang liegt.

6.2.3 Semantische Integration in ITL

Ein Statechart beschreibt eine Menge von Traces (Folgen von Belegungen), die wir als Menge von Intervallen interpretieren. Wie temporallogische Formeln oder parallele Programme werden wir Statecharts durch Ausführen der Übergänge abarbeiten. Sei $\mathcal{I} = (\sigma_0, \sigma_1 \ldots)$ ein Intervall, das einen Trace durch ein Statechart beschreibt. Nach dem Ausführen des ersten Schrittes erhalten wir das nun kürzere Intervall $\mathcal{I}' = (\sigma_1 \ldots)$. In Abschnitt 4.2.2 haben wir die Semantik von Statecharts als Menge von Intervallen definiert, die in einer initialen Konfiguration beginnen. In $\mathcal{I}'$ ist die erste Belegung σ_1 nicht mehr notwendigerweise eine initiale Konfiguration. Für das schrittweise Ausführen von Statecharts im ITL-Kalkül müssen wir deshalb die Statechart-Semantik erweitern. Wir definieren die Semantik als Menge aller maximalen Traces, die in einer konsistenten Statechart-Konfiguration (es gilt: $\downarrow(\sigma_0, SC)$) beginnen und der Übergangsrelation des Statecharts genügen.

$$\begin{aligned} traces_{\mathcal{A}}(SC) := \; & \{(\sigma_0, \ldots, \sigma_n) \mid n \in \mathbb{N}_\infty, \sigma_i \in \textstyle\sum(variables(SC)),\ \downarrow(\sigma_0, SC) \\ & \qquad \sigma_i \; \rho_{step_{SC,\mathcal{A}}} \; \sigma_{i+1} \text{ und } \sigma_n \notin dom(\rho_{step_{SC,\mathcal{A}}})\}. \end{aligned}$$

Sei V die Menge aller Variablen, so definiert die Semantik eines Statecharts nur die Belegung der Variablen $variables(SC)$.

$$\begin{aligned} [\![SC]\!]_{\mathcal{A}} := \; & \{(\sigma_0, \ldots, \sigma_n) \mid n \in \mathbb{N}_\infty, \sigma_i \in \textstyle\sum(V), \\ & \qquad (\sigma_0, \ldots, \sigma_n)\,|_{variables(SC)} \in traces_{\mathcal{A}}(SC)\} \end{aligned}$$

Die Integration von Statecharts in ITL können wir mit dieser Definition der Semantik $[\![SC]\!]_{\mathcal{A}}$ eines Statecharts SC leicht wie folgt definieren. Sei $\mathcal{I}$ ein (ITL-)Intervall, dann gilt:

$$\mathcal{I}, \mathcal{A} \models SC :\Leftrightarrow \mathcal{I} \in [\![SC]\!]_{\mathcal{A}}$$

Die Variablen eines Statecharts sind in ITL flexible Variablen. In ITL gibt es für eine flexible Variable $\dot{x}$ auch die flexiblen Variablen x' und x''. Zur besseren Lesbarkeit lassen wir im Folgenden den Punkt meist weg und schreiben x statt $\dot{x}$. Durch $\sigma_i(x'') = \sigma_{i+1}(x)$ schränkt die ITL-Semantik die Belegung der 2-fach gestrichenen Variablen ein. Um in späteren Beweisen über die Ausgabebelegung nach einem Makro-Schritt sprechen zu können, verlangen wir zusätzlich für die 1-fach gestrichenen Variablen in einem Makro-Schritt $\sigma_i \rho_{makro} \sigma_{i+1}$, dass $output(\sigma_i)$ mit

$$output(\sigma_i) :\Leftrightarrow \forall\ v \in events_{env}(SC) \cup vars_{env}(SC).\ v = v' \text{ und } \sigma_i(tick')$$

gilt. Weitere Einschränkungen sind für die Einbettung in die ITL nicht notwendig.

6.3 Statechart-Kalkül

Wir betrachten Statecharts als Formeln zur Beschreibung dynamischer Systeme. Wenn wir einen Beweis führen, der bestimmte temporallogische Eigenschaften eines Statecharts zeigen soll, steht das Statechart als Vorbedingung im Antezedenten einer Sequenz. Das Statechart beschreibt diejenigen Intervalle, die die temporallogischen Eigenschaften erfüllen sollen. Diese Intervalle werden durch symbolisches Ausführen des Statecharts bestimmt. Dabei berechnen wir sukzessive alle möglichen Folgen von Belegungen, die ein Statechart beschreiben kann.

Um ein Statechart symbolisch ausführen zu können, müssen wir zuerst die Übergangsmengen berechnen, die von der aktuellen Konfiguration aus möglich sind und dann die Übergangsmengen ausführen. Bei der Berechnung nach Damm et al. [DJHP98], die wir in Abschnitt 4.2.2 beschrieben haben, werden während der Berechnung der Übergangsmengen die Aktivierungsbedingungen von Übergängen und statischen Reaktionen ausgewertet. Wir erlauben in diesem Ansatz boolesche Ausdrücke über beliebigen algebraischen Spezifikationen als Aktivierungsbedingungen. Deshalb können die Aktivierungsbedingungen im Allgemeinen nicht automatisch ausgewertet werden. Damit die Berechnung der Übergangsmengen trotzdem automatisch erfolgen kann, modifizieren wir die Berechnung von Damm et al. so, dass die Aktivierungsbedingungen mit zurückgeliefert werden. Die Berechnung der Übergangsmengen erfolgt dann weiterhin automatisch. Der Nachweis, ob die Aktivierungsbedingungen über der aktuellen Variablenbelegung gilt, wird interaktiv geführt. Die Ausführung einer Übergangsmenge beschreiben wir durch ein DL-Programm, das den Statechart-Schritt beschreibt. Zusätzlich zur Ausführungsregel benötigen wir für Statechart-Beweise Induktion. Dazu können wir das Induktionsprinzip für temporallogische Beweise aus Abschnitt 6.1.4 verwenden.

Wir führen in Beweisen über Statecharts alle Mikro-Schritte innerhalb eines Makro-Schrittes explizit aus und fassen sie nicht zu einem Schritt zusammen. Dies ist für die interaktive Verifikation notwendig, um die Nachvollziehbarkeit der Beweise zu gewährleisten. Einen Makro-Schritt kennzeichnen wir dann durch das *tick*-Ereignis. Durch Verwendung des *tick*-Ereignisses können wir auch Aussagen φ formulieren, die nur zu Makro-Schritten gelten und so Beweise über Makro-Schritte führen ($\Box tick \to \varphi$).

6.3.1 Regelschema für den Statechart-Kalkül

In unserem Kalkül definiert eine Statechart-Formel SC die statische Übergangsrelation des Statecharts. Die aktuelle Zustandskonfiguration, Belegung der Variablen und die aktiven Ereignisse werden durch die Formelmengen Γ und Δ einer Sequenz symbolisch beschrieben. Das Abspalten und Ausführen aller konfliktfreien Übergänge eines Statecharts skizziert das Regelschema *sc unwind*.

$$\frac{\bigvee_{i=1}^{n}(cond(stp_i) \wedge step(stp_i) \wedge \circ\, nxt(stp_i)), \Gamma \vdash \Delta}{SC, \Gamma \vdash \Delta}\ sc\ unwind \tag{2}$$

Durch indeterministische Übergänge kann eine Konfiguration eines Statechart in verschiedene Nachfolgekonfigurationen übergehen. Die Regel *sc unwind* berechnet für eine Statechart-Formel SC und die Konfiguration in $\Gamma \vdash \Delta$ eine Menge potentieller Übergangsmengen $stp_1, \ldots, stp_n$. Wir werden im Folgenden in den Funktionen die Statechart-Formel SC immer implizit mitführen. Eine Übergangsmenge stp_i ist tatsächlich aktiviert, wenn die entsprechende Ausführungsbedingung $cond(stp_i)$ erfüllt ist. Ansonsten führt $cond(stp_i)$ für diesen Schritt zu einem Widerspruch im Antezedenten, dieser Fall im Beweis wird geschlossen und der entsprechende Schritt nicht ausgeführt. In der Ausführungsbedingung sind die Aktivierungsbedingungen aller Übergänge und statischen Reaktionen des auszuführenden Schrittes zusammengefasst. Die Variablenwerte, die sich nach dem Statechart-Schritt ergeben, werden durch $step(stp_i)$ berechnet. $step(stp_i)$ ist ein DL-Programm, das die Aktionen der Übergänge und statischen Reaktionen des aktuellen Schrittes ausführt und so den Statechart abwickelt. $step(stp_i)$ „implementiert“ die Schrittausführung nach der Relation ρ_{step} aus der Semantikdefinition. Schließlich wird die neue Zustandskonfiguration berechnet, die, zusammen mit der unveränderten Statechart-Formel SC, in der Nachfolgebelegung ($\circ\ nxt(stp_i)$) gilt. Die formale Definition von $cond(stp_i)$, $step(stp_i)$ und $nxt(stp_i)$ folgt in Abschnitt 6.3.3.

Nach dem Abspalten und Ausführen der Statechart-Schritte führt ein temporallogischer Schritt den Übergang zur Nachfolgebelegung aus (siehe *tl step*-Regel in Abschnitt 6.1.4). Das Statechart befindet sich in der neuen Konfiguration.

Statecharts vs. parallele Programme

Die Behandlung von Statecharts als temporallogische Formeln erfolgt analog zu parallelen Programmen. Die Berechnung der Übergangsmengen entspricht der Funktion $\mathcal{T}_X$ zur Berechnung der ersten Anweisungen eines parallelen Programms. Beide Berechnungen können verschiedene nächste Schritte liefern. Jeder Schritt besteht aus zwei Teilen. Der erste Teil beschreibt die aktuelle Belegung. Bei einem parallelen Programm ist dies das Ausführen der ersten Anweisung, bei Statecharts die Ausführungsbedingung $cond(stp_i)$ und das Ausführen der Übergänge durch $step(stp_i)$.

Der zweite Teil beschreibt die Nachfolgebelegungen. Bei parallelen Programmen beschreibt das Restprogramm die Nachfolgebelegungen. Im Gegensatz zu parallelen Programmen werden Statecharts nicht „abgearbeitet“, denn die statische Übergangsrelation wird nicht geändert. Deshalb wird die Nachfolgebelegung mit $\circ\ nxt(stp_i)$ beschrieben. $nxt(stp_i)$ berechnet die aktiven Zustände der Nachfolgekonfiguration (den neuen „Programmzeiger“ im Statechart). Die statische Übergangsrelation bleibt unverändert.

Schrittausführung

Wir beschreiben die Ausführung eines einzelnen Statechart-Schrittes mit sequentiellen Programmen. Das Programm setzt zuerst die Ereignisse zurück, d. h. es weist den booleschen Ereignisvariablen den Wert false zu, und führt dann die Aktionen der entsprechenden Übergänge aus. Schließlich werden noch die abgeleiteten Ereignisse, die das Betreten

und Verlassen von Zuständen kennzeichnen ($en_..$ bzw. $ex_..$) erzeugt. Die Ergebnisse der Schrittberechnung werden dann den 2-fach gestrichenen Variablen zugewiesen und so die Relation zur Nachfolgekonfiguration hergestellt.

Partielle Zustandskonfigurationen

Wir beschreiben die Zustände eines Statecharts mit booleschen Variablen, wobei sich eine Zustandskonfiguration aus den Werten aller Zustandsvariablen ergibt. In Beweisen über Statecharts stellt es sich manchmal als zu restriktiv heraus, wenn man die Zustandskonfiguration eindeutig beschreibt.

Sehen wir uns dazu nochmals das Beispiel der Zeitschaltuhr in Abbildung 6.1 an und versuchen, die Aussage $\Box x \leq 6$ zu zeigen. Nach Betreten des Zustands *Close* hat x den Wert 0. Für den Nachweis von $\Box x \leq 6$ ist es aber irrelevant, ob x 0, 1, ... 5 ist. Die wichtige Information ist, dass $x \leq 5$ ist. Ansonsten wird der Übergang nach *Open* ausgeführt. Generalisieren wir nach Betreten des Zustands *Close* $x = 0$ zu $x \leq 5$, müssen wir x nicht schrittweise von 1 bis 5 hochzählen. Erhöhen wir dann x um 1, ist x entweder immer noch ≤ 5, dann beenden wir den Beweis per Induktionsschluss, oder $x = 6$ und wir führen den Übergang zu *Open* aus (siehe Beweisbeispiel in Abschnitt 6.3.2, Seite 87).

Die Belegungen der Variablen und Ereignisse werden durch Gleichungen aus der Sequenz beschrieben. Durch eine Verallgemeinerung der Aussage (Abschwächen von $x = 0$ zu $x \leq 5$) beschreibt eine Sequenz dann nicht mehr genau eine Belegung, sondern eine Menge möglicher Belegungen (in denen $x = 0$, ... oder $x = 5$ ist).

Manchmal ist es auch wünschenswert, auf diese Art die konkrete Zustandskonfiguration zu generalisieren. Für die Eigenschaft $\Box x \leq 6$ ist die Information, ob der Zustand *Off* oder *On* aktiv ist, vollkommen irrelevant. Wenn wir von der konkreten Belegung von *Off* und *On* abstrahieren, erhalten wir eine allgemeinere Invariante und können z. B. den Fall *Open* $\wedge$ *Off* und *Open* $\wedge$ *On* zusammen betrachten. Dazu verwerfen wir die Information, ob *Off* bzw. *On* true oder false ist (im Sequenzenkalkül gibt es die Regel *weakening*, die Voraussetzungen im Antezedenten entfernt und damit die Aussage verallgemeinert). Die Sequenz beschreibt nun sowohl Belegungen, in denen *Off* gilt, als auch Belegungen, in denen *Off* nicht gilt. Im Folgenden betrachten wir auch Sequenzen, die eine solche *partielle* Zustandskonfiguration von Statecharts beschreiben. Den Unterschied zwischen vollständigen und partiellen Zustandskonfigurationen verdeutlichen wir in den Beweisbeispielen im folgenden Abschnitt 6.3.2.

Für die Generalisierung der Zustandskonfiguration muss die Berechnung der Übergangsmengen angepasst werden, denn bei der bisherigen Berechnung möglicher Übergänge eines *Oder*-Zustands wird eine vollständige Zustandskonfiguration vorausgesetzt. In Abschnitt 6.3.3 beschreiben wir eine allgemeinere Berechnung der Übergangsmengen, die sowohl für eine vollständige, als auch für eine partielle Zustandskonfiguration die möglichen Übergangsmengen berechnet. Vollständige Zustandskonfigurationen sind damit ein Spezialfall partieller Zustandskonfigurationen.

Initialisierung

Üblicherweise möchten wir Aussagen für einen Statechart SC zeigen, der in einer Ausgangskonfiguration startet. Um die Ausgangskonfiguration nicht für jeden Beweis explizit beschreiben zu müssen, haben wir den Statecharttyp $\widehat{SC}$ eingeführt, dessen Semantik analog der Semantik aus Abschnitt 4.2.2 ist, also in einer Ausgangskonfiguration startet. D. h. die Semantik von $\widehat{SC}$ ist $[\![\widehat{SC}]\!]^i$. Eine Initialisierungsregel führt nun Bedingungen ein, welche die Ausgangskonfiguration eines Statecharts beschreibt und wandelt für den weiteren Beweis den Typ $\widehat{SC}$ in SC um. Nun kann der Statechart SC, in der Ausgangskonfiguration startend, nach und nach abgewickelt werden. $\widehat{SC}$ selbst kann nicht abgewickelt werden, da dessen Traces immer in einer Ausgangsbelegung beginnen. Nach einem Abwicklungsschritt befindet sich das Statechart jedoch nicht mehr in einer Ausgangsbelegung (siehe dazu auch die Diskussion in Abschnitt 6.2.3).

Die Initialisierung berechnet die Ausgangsbelegungen von Variablen, die initial aktiven Zustände und setzt alle Ereignisse inaktiv. Die Funktion $\mathit{initialize}(\widehat{SC})$ berechnet die initiale Konfiguration nach Abschnitt 4.2.2, Seite 41.

$$\begin{aligned}&\mathit{initialize}(\widehat{SC}) := \\ &\quad \textstyle\bigwedge_{s \in \mathit{states}(SC)\ \cap\ \mathit{decompl}(\mathit{root}(SC))} s \wedge \bigwedge_{\overline{s} \in \mathit{states}(SC) \setminus \mathit{decompl}(\mathit{root}(SC))} \neg\ \overline{s} \wedge \\ &\quad \textstyle\bigwedge_{e \in \mathit{events}(SC)} \neg\ e \wedge \bigwedge_{v \in \mathit{vars}(SC)} v = \mathit{default}(v)\end{aligned}$$

$\mathit{default}(v)$ weist dazu den Variablen den Initialwert aus dem *initial value*-Slot des Statecharts zu, falls dieser vorhanden ist, ansonsten einen beliebigen Wert. Die Regel *sc initialize*

$$\frac{\mathit{initialize}(\widehat{SC}), SC, \Gamma \vdash \Delta}{\widehat{SC}, \Gamma \vdash \Delta}\ \mathit{sc\ initialize}$$

definiert die initiale Belegung eines Statecharts.

Beispiel 6.1 *Initialisierung*
Betrachten wir die Zeitschaltuhr aus Abbildung 6.1 und bezeichnen das Statechart SYS, *um es vom Zustand* System *unterscheiden zu können.* $\widehat{\mathrm{SYS}}$ *wird wie folgt initialisiert.*

$$\begin{aligned}&\mathrm{initialize}(\widehat{\mathrm{SYS}}) = \\ &\quad \mathrm{System} \wedge \mathrm{Timer} \wedge \mathrm{Off} \wedge \mathrm{Light} \wedge \mathrm{Open} \wedge \neg\ \mathrm{On} \wedge \neg\ \mathrm{Close} \wedge \neg\ \mathrm{tick} \wedge \\ &\quad \neg\ \mathrm{set} \wedge \neg\ \mathrm{press} \wedge \neg\ \mathrm{sw_off} \wedge \neg\ \mathrm{en_System} \wedge \neg\ \mathrm{en_Light} \wedge \neg\ \mathrm{en_Off} \wedge \\ &\quad \neg\ \mathrm{en_Timer} \wedge \neg\ \mathrm{en_Open} \wedge \neg\ \mathrm{en_On} \wedge \neg\ \mathrm{en_Close} \wedge \neg\ \mathrm{ex_System} \wedge \\ &\quad \neg\ \mathrm{ex_Light} \wedge \neg\ \mathrm{ex_Off} \wedge \neg\ \mathrm{ex_Timer} \wedge \neg\ \mathrm{ex_Open} \wedge \neg\ \mathrm{ex_Close} \wedge x = 0\end{aligned}$$

Möchten wir nun zeigen, dass für das Statechart SYS *der Wert von x immer ≤ 6 ist, beginnen wir den Beweis mit*

$$\frac{\mathrm{initialize}(\widehat{\mathrm{SYS}}), \mathrm{SYS} \vdash \Box x \leq 6}{\widehat{\mathrm{SYS}} \vdash \Box x \leq 6}\ \mathrm{sc\ initialize}$$

und wickeln anschließend das Statechart SYS *mit der Regel* sc unwind *ab.*

Um Aussagen über komplexen Statecharts beweisen zu können, benötigt man häufig Teilaussagen, die separat formuliert, bewiesen und im Gesamtbeweis als Lemma eingesetzt werden. Diese Teilaussagen beginnen meist nicht in einer Ausgangskonfiguration, sondern in einer beliebigen Konfiguration. Dazu geben wir Einschränkungen der Konfiguration explizit an. Möchte man z. B. die Ausführung in einem Zustand s beginnen und die Aussage φ zeigen, formuliert man die Beweisverpflichtung $SC, s \vdash \varphi$.

Konsistenz

Statecharts definieren Konsistenzeigenschaften für Belegungen (siehe Abschnitt 4.2.2, Seite 41: $\downarrow(\sigma, SC)$). Es ist z. B. nicht erlaubt, dass in einer Belegung zwei Unterzustände eines *Oder*-Zustands oder in einem Makro-Schritt lokale Ereignisse aktiv sind. Für Statecharts, die in einer Ausgangskonfiguration starten, erhält die Schrittberechnung die Konsistenz. Beschreibt eine Sequenz jedoch eine partielle Zustandskonfiguration, können Übergänge berechnet werden, die zu Inkonsistenzen führen.

Beispiel 6.2
Betrachten wir für die Zeitschaltuhr eine Sequenz, die eine partielle Zustandskonfiguration beschreibt:

$$\text{Off}, \text{Open}, \text{SYS} \vdash \Box x \leq 6$$

In der Sequenz ist keine Information über die Belegung von Close *enthalten und damit kann* Close *gelten, oder nicht. Unter der Vorbedingung, dass der Zustand* Close *aktiv ist, liefert die Schrittberechnung den Übergang* t_4 *von* Close *nach* Open*. Da in der obigen Sequenz keine Aussage über* Close *enthalten ist, ist die Vorbedingung* Close *nicht notwendigerweise verletzt, führt aber durch die Konsistenzbedingung des Statecharts* SYS *zur Inkonsistenz* Close *und* Open*.*

Analog kann es geschehen, dass bei der Schrittberechnung eine Aktivierungsbedingung für eine Übergangsmenge berechnet wird, die sowohl aktive lokale Ereignisse als auch aktive Umgebungsereignisse fordert (dann muss *tick* gelten, siehe Definition von $guard(t)$). In beiden Fällen erhalten wir Sequenzen, die inkonsistente Belegungen beschreiben und deshalb im Beweisverlauf nicht weiter beachtet werden müssen.

Um Inkonsistenzen zu erkennen, erzeugt die Regel *sc wellformed* im Antezedenten der Sequenz ein Prädikat, das konsistente Zustandskonfigurationen beschreibt. Beschreibt eine Sequenz nur inkonsistente Belegungen, wird dieses Prädikat zu false ausgewertet und wir erhalten einen Widerspruch im Antezedenten. Ansonsten präzisiert das Prädikat die Sequenz, so dass sie nur noch konsistente Belegungen beschreibt. Wir definieren das Konsistenzprädikat für Zustände mit Hilfe konsistenter Konfigurationen $conf(SC)$.

$$cons_{states}(SC) := \bigvee_{cnf \in conf(SC)} s_1 \wedge \ldots \wedge s_n \wedge \neg\, \bar{s}_1 \wedge \ldots \wedge \neg\, \bar{s}_m$$

für $s_i \in cnf$, $\bar{s}_j \in states(SC) \backslash cnf$. Für Ereignisse definieren wir

$$cons_{events}(SC) := \neg\ ((\bigvee_{e \in events_{loc}(SC)} e) \wedge tick).$$

Mit diesen beiden Prädikaten können wir nun die Regel *sc wellformed* folgendermaßen definieren:

$$\frac{cons_{states}(SC), cons_{events}(SC), SC, \Gamma \vdash \Delta}{SC, \Gamma \vdash \Delta}\ sc\ wellformed$$

Beispiel 6.3 *Konsistente Statecharts*
Betrachten wir nochmals die Zeitschaltuhr aus Abbildung 6.1. Eine Zustandskonfiguration ist konsistent, wenn sie

$$\begin{array}{lll} cons_{states}(\text{SYS}) := & \text{System} \wedge \text{Light} \wedge \text{Timer} \wedge & \text{On} \wedge \neg\text{Off} \wedge \text{Open} \wedge \neg\text{Close} \vee \\ & \ldots & \neg\text{On} \wedge \text{Off} \wedge \text{Open} \wedge \neg\text{Close} \vee \\ & \ldots & \text{On} \wedge \neg\text{Off} \wedge \neg\text{Open} \wedge \text{Close} \vee \\ & \ldots & \neg\text{On} \wedge \text{Off} \wedge \neg\text{Open} \wedge \text{Close} \end{array}$$

erfüllt. Gilt Open *und* Close*, widerspricht dies dem Prädikat* $cons_{states}(\text{SYS})$*. Für Ereignisse gilt*

$$cons_{events}(\text{SYS}) := \neg\ ((\text{set} \vee \text{sw_off}) \wedge \text{tick}).$$

6.3.2 Beweis-Beispiel: Zeitschaltuhr

Wir präsentieren ein Beweis-Beispiel für die Zeitschaltuhr, die das Beweisprinzip für Statecharts verdeutlicht. Für die Steuerung der Zeitschaltuhr aus Abbildung 6.1 zeigen wir, dass der Zähler nie größer sechs wird, die Lampe also nicht länger als gewünscht leuchtet. Diese Aussage zeigen wir sowohl mit dem Ansatz der vollständigen Zustandskonfiguration, als auch in einer Variante mit partiellen Zustandskonfigurationen. Um im Folgenden das Gesamtchart der Zeitschaltuhr vom parallelen Zustand *System* unterscheiden zu können, nennen wir das Gesamtchart *SYS*.

Vollständige Zustandskonfiguration

Den Beweis beginnen wir in der Ausgangskonfiguration und zeigen deshalb $\widehat{SYS} \vdash \Box x \leq 6$.

sc initialize Zuerst wird das Statechart $\widehat{SYS}$ initialisiert.

$$\frac{\begin{array}{l} System, Light, Off, Timer, Open, \neg\ On, \neg\ Close, \\ \neg\ tick, \neg\ press, \neg\ sw_off, \neg\ set, \neg\ en_System, \neg\ en_Light, \neg\ en_Off, \\ \neg\ en_Timer, \neg\ en_Open, \neg\ en_On, \neg\ en_Close, \neg\ ex_System, \neg\ ex_Light, \\ \neg\ ex_Off, \neg\ ex_Timer, \neg\ ex_Open, \neg\ ex_On, \neg\ ex_Close, x = 0, SYS \end{array} \vdash \Box x \leq 6}{\widehat{SYS} \vdash \Box x \leq 6}$$

Die Regel *sc initialize* setzt die Ausgangszustände, inaktiviert alle Ereignisse und initialisiert die Variablen. Der weitere Beweis basiert auf der Statechart-Formel *SYS*.

Zur besseren Lesbarkeit lassen wir abgeleitete Ereignisse, die das Betreten bzw. Verlassen von Zuständen kennzeichnen, beiseite. Im Folgenden werden auch die Zustände *System*, *Light* und *Timer*, die immer aktiv sind, nicht weiter aufgeführt und für die verbleibenden Zustände vereinbaren wir, dass nur die aktiven Zustände auf der Sequenz angezeigt werden. Die inaktiven werden nicht aufgeführt (dies dient nur der Lesbarkeit und beschreibt hier *keine* partiellen Zustandskonfigurationen). Die Prämisse vereinfacht sich dadurch zu $Off, Open, \neg\ tick, \neg\ press, \neg\ sw_off, \neg\ set, x = 0, SYS \vdash \Box x \leq 6$.

ind always An dieser Stelle beginnen wir eine Induktion

$$\frac{IndHyp_1, Off, Open, \neg\ tick, \neg\ press, \neg\ sw_off, \neg\ set, x = 0, SYS \vdash \Box x \leq 6}{Off, Open, \neg\ tick, \neg\ press, \neg\ sw_off, \neg\ set, x = 0, SYS \vdash \Box x \leq 6} \quad (1)$$

und erhalten die Induktionshypothese $IndHyp_1 :=$

- $(Off \wedge Open \wedge \neg tick \wedge \neg press \wedge \neg sw_off \wedge \neg set \wedge x = 0 \wedge SYS \rightarrow \Box x \leq 6)\ \mathrm{U}\ \neg x \leq 6.$

Zur Übersichtlichkeit schreiben wir die Induktionshypothese $IndHyp_1$ nicht weiter auf die Sequenz, merken uns aber, dass sie als zusätzliche Voraussetzung gilt.

sc unwind Das Abwickeln des Statecharts ergibt die möglichen Nachfolgekonfigurationen. Das Ausführen eines Statechart-Schrittes erfolgt dann durch ein DL-Programm, das die Variablenwerte für die Nachfolgekonfiguration berechnet. Die Verbindung zur Nachfolgekonfiguration stellen wir durch die 2-fach gestrichenen Variablen dar. Wir fordern nach der Programmausführung, dass die 2-fach gestrichenen Variablen die Werte enthalten, die durch die Schrittausführung berechnet wurden. Da die 2-fach gestrichenen Variablen den ungestrichenen Variablen in der Nachfolgebelegung entsprechen, haben wir so die Relation zur nächsten Belegung geschaffen. Den Übergang zur Nachfolgekonfiguration führt also nicht die *sc unwind*-Regel aus, sondern die Regel *tl step*.

$$\frac{\begin{array}{l}\langle tick := \text{false} \mid sw_off := \text{false} \mid set := \text{false}; tick := \text{true}; \\ \ press :=?\rangle(tick = tick'',\ press = press'', sw_off = sw_off'', \\ \ set = set'', x = x''),\ \circ\ (Off \wedge Open \wedge SYS), \\ Off, Open, \neg\ tick, \neg\ press, \neg\ sw_off, \neg\ set, x = 0\end{array} \vdash \Box x \leq 6}{\begin{array}{l}Off, Open, \neg\ tick, \neg\ press, \\ \neg\ sw_off, \neg\ set, x = 0, SYS\end{array} \vdash \Box x \leq 6} \quad (2)$$

Im Ausgangszustand des Statecharts sind alle Ereignisse inaktiv. Deshalb erhalten wir nur eine mögliche Nachfolgekonfiguration. Ein Makro-Schritt kann Umgebungsereignisse einlesen. Zuerst werden in einem Makro-Schritt alle Ereignisse zurückgesetzt (den Ereignissen wird false zugewiesen), dann der Makro-Schritt durch $tick$ = true gekennzeichnet und das Umgebungsereignis eingelesen ($press$:=?). Die Umgebung setzt willkürlich das

Ereignis *press* auf aktiv oder inaktiv. Die Zustandskonfiguration ändert sich nicht. Die Nachbedingung des DL-Programms fordert, dass die 2-fach gestrichenen Variablen die neu berechneten Werte erhalten.

simplify Das Ausführen des DL-Programms zur Berechnung der Nachfolgebelegung fassen wir in einem Vereinfachungsschritt zusammen.

$$\frac{\begin{array}{l} tick'', \neg\, sw_off'', \neg\, set'', x'' = 0,\ \circ\,(Off \wedge\ Open \wedge\ SYS), \\ Off, Open, \neg\, tick, \neg\, press, \neg\, sw_off, \neg\, set, x = 0 \end{array} \vdash \Box x \leq 6}{\begin{array}{l} \langle tick := \text{false} \mid sw_off := \text{false} \mid set := \text{false}; tick := \text{true}; \\ press := ?\rangle(tick = tick'',\ press = press'', sw_off = sw_off'', \\ set = set'', x = x''),\ \circ\,(Off \wedge\ Open \wedge\ SYS), \\ Off, Open, \neg\, tick, \neg\, press, \neg\, sw_off, \neg\, set, x = 0 \end{array} \vdash \Box x \leq 6} \quad (3)$$

Der Vereinfachungsschritt weist den 2-fach gestrichenen Variablen die Werte zu, die der Statechart-Schritt berechnet hat. Da $press''$ einen beliebigen Wert zugewiesen bekommt, enthält diese Zuweisung keine Information und der Simplifier entfernt sie von der Sequenz. Deshalb kommt $press''$ in der Prämisse von (3) nicht mehr vor. Damit haben wir den ersten Statechart-Schritt symbolisch ausgeführt. Den 2-fach gestrichenen Variablen der aktuellen Belegung wurden die Werte, die sich durch Ausführen des Statecharts ergeben, zugewiesen. Da die 2-fach gestrichenen Variablen den ungestrichenen Variablen der Nachfolgebelegung entsprechen, wurde damit die Konfiguration der Nachfolgebelegung bestimmt.

always right Für den Nachweis der Eigenschaft $\Box x \leq 6$ wird die temporallogische Formel abgewickelt.

$$\frac{\dfrac{}{\ldots, x = 0 \vdash x \leq 6} \qquad \begin{array}{l} tick'', \neg\, sw_off'', \neg\, set'', x'' = 0, \\ \circ\,(Off \wedge\ Open \wedge\ SYS), \\ Off, Open, \neg\, tick, \neg\, press, \neg\, sw_off, \\ \neg\, set, x = 0 \end{array} \vdash\ \bullet\,\Box x \leq 6}{\begin{array}{l} tick'', \neg\, sw_off'', \neg\, set'', x'' = 0,\ \circ\,(Off \wedge\ Open \wedge\ SYS), \\ Off, Open, \neg\, tick, \neg\, press, \neg\, sw_off, \neg\, set, \\ x = 0 \end{array} \vdash \Box x \leq 6} \quad (4)$$

Das Abwickeln des $\Box$-Operators erzeugt zwei Beweispflichten. Es muss für die aktuelle Belegung gezeigt werden, dass $x \leq 6$ ist (linke Prämisse). Die Prämisse ist trivial wahr, da $x = 0$ ist, und das Beweisziel wird geschlossen. Die zweite Beweisverpflichtung verlangt, dass die Eigenschaft auch ab dem nächsten Schritt (falls dieser vorhanden ist) gelten muss ($\bullet\,\Box x \leq 6$).

tl step Die verbleibenden TL-Formeln sprechen nun alle über die Nachfolgebelegung. Ein *tl step* setzt die Nachfolgebelegung als neue, aktuelle Belegung.

$$\frac{tick, \neg\ sw_off, \neg\ set, x = 0, Off, Open, SYS\ \vdash \Box x \leq 6}{\begin{array}{ll} tick'', \neg\ sw_off'', \neg\ set'', x'' = 0,\ \circ\ (Off \wedge\ Open \wedge\ SYS), & \\ Off, Open, \neg\ tick, \neg\ press, \neg\ sw_off, \neg\ set, & \vdash\ \bullet\ \Box x \leq 6 \\ x = 0 & \end{array}} \quad (5)$$

Die Werte der 2-fach gestrichenen Variablen der Konklusio werden den ungestrichenen Variablen der Prämisse zugewiesen und die Next-Operatoren werden entfernt. Damit spricht die Sequenz jetzt über die Nachfolgebelegung. Die Kopien der ungestrichenen und 1-fach gestrichenen Variablen sind für den weiteren Beweis nicht notwendig und werden deshalb nicht aufgeführt.

Bemerkung An dieser Stelle im Beweis befinden wir uns nicht mehr in einer Ausgangskonfiguration (es gilt *tick*), aber die Statechart-Formel *SC* steht auf der Sequenz und benötigt eine Semantik. Deshalb konnte die Semantik von Statecharts in Abschnitt 6.2.3 nicht auf initialen Traces definiert werden.

sc unwind, simplify, tl step Nun kann wieder ein Statechart-Schritt abgewickelt werden. In diesem Fall erhalten wir zwei Möglichkeiten. Entweder es gilt *press* und wir verlassen den Zustand *Off*, oder *press* gilt nicht und das Statechart verbleibt im Zustand *Off*. Wenn wir vollständige Zustandskonfigurationen betrachten, behandeln wir üblicherweise die möglichen Statechart-Schritte getrennt, da sie verschiedene Situationen beschreiben. D. h. die Disjunktion im Regelschema der *sc unwind*-Regel spalten wir in verschiedene Fälle auf.

$$\frac{\dfrac{\dfrac{apply\ induction\ (IndHyp_1)}{\begin{array}{ll} \neg\ tick, \neg\ press, \neg\ sw_off, & \\ \neg\ set, x = 0, Off, Open, & \vdash \Box x \leq 6 \\ SYS & \end{array}}}{\vdots} \qquad \dfrac{\begin{array}{ll} \neg\ tick, \neg\ press, \neg\ sw_off, & \\ set, x = 0, Open, On, & \vdash \Box x \leq 6 \\ SYS & \end{array}}{\vdots}}{tick, \neg\ sw_off, \neg\ set, x = 0, Off, Open, SYS\ \vdash \Box x \leq 6} \quad (6)$$

Abhängig davon, ob *press* gilt oder nicht, findet ein Übergang in den Zustand *On* statt und das Ereignis *set* wird erzeugt (rechte Prämisse), oder das Statechart verbleibt im Zustand *Off* (linke Prämisse). Nach einem Vereinfachungsschritt *simplify* und dem temporallogischen Schritten *always right* und *tl step*, sehen wir, dass die linke Prämisse dieselbe Form wie in der Konklusio der Ableitung (1) hat. Die Anwendung der Induktionshypothese $IndHyp_1$ schließt dieses Beweisziel.

In der rechten Prämisse wurde ein Mikro-Schritt durchgeführt und das Statechart befindet sich nun in dem Zustand *On* und das Ereignis *set* ist aktiv.

sc unwind, simplify, tl step

$$\frac{\neg\ tick, \neg\ press, \neg\ sw_off, \neg\ set, x = 0, On, Close, SYS\ \vdash \Box x \leq 6}{\neg\ tick, \neg\ press, \neg\ sw_off, set, x = 0, Open, On, SYS\ \vdash \Box x \leq 6}\ (7)$$

Ein weiterer Statechart-Schritt überführt das Statechart in den Zustand *Close*. Nun beginnt der Timer zu zählen.

generalize Damit wir nicht für jeden Zählerstand des Timers (1. . .6) einen Makro-Schritt durchführen müssen, generalisieren wir die Sequenz.

$$\frac{\neg\ tick, \neg\ press, \neg\ sw_off, \neg\ set, x \leq 5, On, Close, SYS\ \vdash \Box x \leq 6}{\neg\ tick, \neg\ press, \neg\ sw_off, \neg\ set, x = 0, On, Close, SYS\ \vdash \Box x \leq 6}\ (8)$$

Im Zustand *Close* ist die wichtige Information, dass der Zähler nicht größer 5 ist. Sonst wird der Zustand verlassen. Deshalb kann $x = 0$ zu $x \leq 5$ verallgemeinert werden. Hier beginnen wir nochmals eine Induktion mit der Induktionshypothese $IndHyp_2$:=

$$\bullet\ (\neg tick \wedge \neg press \wedge \neg sw_off \wedge \neg set \wedge x \leq 5 \wedge \\ On \wedge Close \wedge SYS \wedge IndHyp_1 \rightarrow \Box x \leq 6)\ \mathrm{U}\ \neg x \leq 6,$$

die wir uns wieder merken und nicht auf der Sequenz mitführen.

sc unwind, simplify, tl step Wir überspringen einen Makro-Schritt, der das *tick*-Ereignis erzeugt. Auf das *tick*-Ereignis reagiert der Zähler und erhöht x im nächsten Statechart-Schritt.

$$\frac{\dfrac{apply\ induction\ (IndHyp_2)}{\begin{array}{l}\neg\ tick, \neg\ press, \neg\ sw_off,\\ \neg\ set, x \leq 5, On, Close,\ \ \vdash \Box x \leq 6\\ SYS\end{array}} \qquad \begin{array}{l}\neg\ tick, \neg\ press, \neg\ sw_off,\\ \neg\ set, x = 6, On, Close,\ \ \vdash \Box x \leq 6\\ SYS\end{array}}{\dfrac{\vdots}{\neg\ tick, \neg\ press, \neg\ sw_off, \neg\ set, x \leq 5, On, Close, SYS\ \vdash \Box x \leq 6}}\ (9)$$

Es entstehen zwei Fälle. Entweder ist die Zählervariable x immer noch ≤ 5, dann kann die Beweisverpflichtung per Induktionsschluss mit $IndHyp_2$ gezeigt werden (linke Prämisse). Im zweiten Fall wird x zu 6.

sc unwind, simplify, tl step

$$\frac{apply\ induction\ (IndHyp_1)}{\neg\ tick, \neg\ press, \neg\ sw_off, \neg\ set, x = 0, Off, Open, SYS\ \vdash \Box x \leq 6}\ (12)$$

$$\frac{\vdots}{\neg\ tick, \neg\ press, sw_off, \neg\ set, x = 0, On, Open, SYS\ \vdash \Box x \leq 6}\ (11)$$

$$\frac{\vdots}{\neg\ tick, \neg\ press, \neg\ sw_off, \neg\ set, x = 6, On, Close, SYS\ \vdash \Box x \leq 6}\ (10)$$

Wenn $x = 6$ gilt, wird der Zustand *Close* verlassen, das *sw_off* Ereignis erzeugt und x auf 0 gesetzt (Ableitung (10)). Das Ereignis *sw_off* bewirkt, dass auch *On* verlassen wird (Ableitung (11)). Nun befindet sich das Statechart wieder im Ausgangszustand (vgl. mit Ableitung (1)) und der Gesamtbeweis wird mittels Induktionsschluss mit $IndHyp_1$ beendet (Ableitung (12)). Der Beweisbaum, der sich aus den Ableitungschritten ergibt, findet sich in Abbildung 6.3 a), Seite 95.

Partielle Zustandskonfiguration

Zum Vergleich führen wir den Beweis für $\Box x \leq 6$ mit einer partiellen Zustandskonfiguration durch. Die Idee von partiellen Zustandskonfigurationen ist, dass nicht nur von konkreten Variablenwerten abstrahiert wird ($x \leq 5$ statt $x = 0$), sondern auch von den konkreten Zustandskonfigurationen. Im Beweis für die Zeitschaltuhr ist die Information, ob die Lampe an ist, oder nicht, unwichtig (*Light* in *On* bzw. *Off*). Die für den Beweis wesentliche Information ergibt sich aus dem Verlassen des Zustands *Close*, wenn $x > 5$ ist. Deshalb geben wir im folgenden Beweis die Belegung der Zustände *On* und *Off* nicht explizit an.

Initialisierung Da wir in diesem Beispiel nicht in einer initialen Konfiguration starten, schränken wir die Ausgangskonfiguration explizit ein. Der Zustand *Open* soll aktiv sein und $x = 0$. Wir beginnen den Beweis also mit der Sequenz

$$SYS, Open, x = 0 \vdash \Box x \leq 6.$$

Im Ausgangszustand beginnen wir eine Induktion mit der Induktionshypothese $IndHyp_1 :=$

$$\bullet\ (SYS \wedge Open \wedge x = 0 \rightarrow \Box x \leq 6)\ \mathrm{U}\ \neg\ x \leq 6$$

sc unwind Das Abwickeln des Statecharts ergibt für die partielle Zustandskonfiguration *Open* 8 verschiedene Fälle.

$$\frac{\begin{array}{l} Off, \ldots,\ \circ\,(On \wedge Close) \vee Off, \ldots,\ \circ\,(Off \wedge Close) \vee \\ On, \ldots,\ \circ\,(Off \wedge Close) \vee On, \ldots,\ \circ\,(On \wedge Close) \vee \\ Off, \ldots,\ \circ\,(On \wedge Open) \vee Off, \ldots,\ \circ\,(Off \wedge Open) \vee \quad \vdash \Box x \leq 6 \\ On, \ldots,\ \circ\,(Off \wedge Open) \vee On, \ldots,\ \circ\,(On \wedge Open), \\ \circ\ SYS, Open, x = 0 \end{array}}{SYS, Open, x = 0 \vdash \Box x \leq 6}$$

Als Vorbedingung für einen Statechart-Schritt ist das Unterchart *Light* entweder in *On* oder *Off*. Ausgehend davon kann *Light* einen Übergang machen, oder nicht. Auch der *Timer* kann einen Übergang machen oder eben nicht. Da wir aber nicht alle Fälle einzeln betrachten wollen, suchen wir die für den Beweis wichtige Gemeinsamkeit.

cut formula Entscheidend für den weiteren Beweis ist, ob wir in *Open* bleiben oder nach *Close* gehen. Deshalb führen wir die Formel $Open'' \wedge x'' = 0 \vee Close'' \wedge x'' \leq 5$ ein. Entweder sind wir nach der Schrittausführung in *Open* und $x = 0$ oder wir wechseln nach *close*. Wenn wir nach *Close* wechseln, verallgemeinern wir, wie beim Beweis mit vollständiger Zustandskonfiguration, $x = 0$ zu $x \leq 5$.

$$\dfrac{\begin{array}{l} Open'' \wedge x'' = 0 \vee Close'' \wedge x'' \leq 5 \\ Off, \ldots, \circ (On \wedge Close) \vee \ldots, \\ \quad \circ\, SYS, Open, x = 0 \\ \vdash \Box x \leq 6 \end{array} \qquad \dfrac{\textit{simlify, tl step}}{\begin{array}{l} Off, \ldots, \circ (On \wedge Close) \\ \vee \ldots, \circ\, SYS, Open, x = 0 \\ \vdash Open'' \wedge x'' = 0 \vee Close'' \\ \wedge\, x'' \leq 5, \Box x \leq 6 \end{array}}}{\begin{array}{l} Off, \ldots, \circ (On \wedge Close) \vee Off, \ldots, \circ (Off \wedge Close) \vee \\ On, \ldots, \circ (Off \wedge Close) \vee On, \ldots, \circ (On \wedge Close) \vee \\ Off, \ldots, \circ (On \wedge Open) \vee Off, \ldots, \circ (Off \wedge Open) \vee \quad \vdash \Box x \leq 6 \\ On, \ldots, \circ (Off \wedge Open) \vee On, \ldots, \circ (On \wedge Open), \\ \circ\, SYS, Open, x = 0 \end{array}} \ (1)$$

In der rechten Prämisse ist zu zeigen, dass alle möglichen Übergänge die Eigenschaft $Open'' \wedge x'' = 0 \vee Close'' \wedge x'' \leq 5$ erfüllen. Dies geschieht mit den Regeln *simplify* und *tl step*. Sie führen die 8 möglichen Statechart-Schritte aus und zeigen, dass sie die Eigenschaft erfüllen. Nun können wir mit dem Beweis in der linken Prämisse fortfahren. Da die wichtige Information über die Nachfolgekonfiguration in der eingeführten Formel steht, entfernen wir die berechneten Übergangsmengen von der Sequenz und erhalten

$$Open'' \wedge x'' = 0 \vee Close'' \wedge x'' \leq 5,\ \circ\, SYS, Open, x = 0 \vdash \Box x \leq 6.$$

tl step Nun führen wir den Übergang zur neuen Konfiguration aus (2). Zur Vereinfachung lassen wir hier das Abwickeln des $\Box$-Operators weg. $x = 0$ erfüllt $\Box x \leq 6$ für den aktuellen Zustand und wir führen den Beweis in der Nachfolgekonfiguration fort.

$$\dfrac{\dfrac{\dfrac{\textit{apply induction } (IndHyp_1)}{Open \wedge x = 0, SYS, \vdash \Box x \leq 6} \quad Close \wedge x \leq 5, SYS \vdash \Box x \leq 6}{Open \wedge x = 0 \vee Close \wedge x \leq 5, SYS \vdash \Box x \leq 6} \ (3)}{Open'' \wedge x'' = 0 \vee Close'' \wedge x'' \leq 5,\ \circ\, SYS, Open, x = 0 \vdash \Box x \leq 6} \ (2)$$

Wenn wir eine Fallunterscheidung durchführen (3), erhalten wir in der linken Prämisse dieselbe Formel wie zu Beweisbeginn und schließen den Beweis per Induktion mit $IndHyp_1$. Für die rechte Prämisse beginnen wir erneut eine Induktion mit $IndHyp_2 :=$

$$\bullet\ (Close \wedge x \leq 5 \wedge SYS \wedge IndHyp_1 \rightarrow \Box x \leq 6)\ \mathrm{U}\ \neg x \leq 6.$$

sc unwind In der rechten Prämisse wickeln wir das Statechart erneut ab.

$$\frac{\begin{array}{ll}\ldots tick \wedge \langle \ldots x := x+1 \ldots \rangle \wedge (\circ\, Close \wedge \ldots) \vee & \\ \ldots \neg\, tick \ldots \wedge (\circ\, Close \wedge \ldots) & \vdash \Box x \leq 6 \\ Close \wedge x \leq 5,\ \circ\, SYS & \end{array}}{Close \wedge x \leq 5, SYS, \vdash \Box x \leq 6}$$

Keiner der Übergänge führt aus *Close* heraus, jedoch unterscheiden sie sich dadurch, ob x um 1 erhöht wird, oder nicht.

cut formula Für den weiteren Beweis unterscheiden wir, ob x in der Nachfolgekonfiguration ≤ 5 oder $= 6$ ist, d. h. ob $Close'' \wedge (x'' \leq 5 \vee x'' = 6)$ gilt.

$$\frac{\begin{array}{l} Close'' \wedge (x'' \leq 5 \vee x'' = 6), \\ Close \wedge x \leq 5,\ \circ\, SYS \\ \vdash \Box x \leq 6 \end{array} \quad \dfrac{simlify,\ tl\ step}{\begin{array}{l} \ldots tick \wedge \langle \ldots x := x+1 \ldots \rangle \wedge \ldots \vee \\ \ldots \neg\, tick \ldots \wedge (\circ\, Close \wedge \ldots) \\ Close \wedge x \leq 5,\ \circ\, SYS \\ \vdash \Box x \leq 6, Close'' \wedge (x'' \leq 5 \vee x'' = 6) \end{array}}}{\begin{array}{ll}\ldots tick \wedge \langle \ldots x := x+1 \ldots \rangle \wedge (\circ\, Close \wedge \ldots) \vee & \\ \ldots \neg\, tick \ldots \wedge (\circ\, Close \wedge \ldots) & \vdash \Box x \leq 6 \\ Close \wedge x \leq 5,\ \circ\, SYS & \end{array}}$$

In der linken Prämisse können wir wieder auf die Übergangsmengen verzichten. In der rechten Prämisse zeigen wir, dass alle möglichen Übergangsmengen $Close'' \wedge (x'' \leq 5 \vee x'' = 6)$ erfüllen.

tl step Wir wickeln erneut den $\Box$-Operator ab und berechnen die Nachfolgekonfiguration in Ableitung (4).

$$\frac{\dfrac{\dfrac{apply\ induction\ (IndHyp_2)}{x \leq 5, Close, SYS \vdash \Box x \leq 6} \quad x = 6, Close, SYS \vdash \Box x \leq 6}{x \leq 5 \vee x = 6, Close, SYS \vdash \Box x \leq 6}\ (5)}{Close'' \wedge (x'' \leq 5 \vee x'' = 6), Close \wedge x \leq 5,\ \circ\, SYS \vdash \Box x \leq 6}\ (4)$$

Nach der Fallunterscheidung $x \leq 5 \vee x = 6$ in Ableitung (5) können wir die linke Prämisse per Induktion mit $IndHyp_2$ schließen.

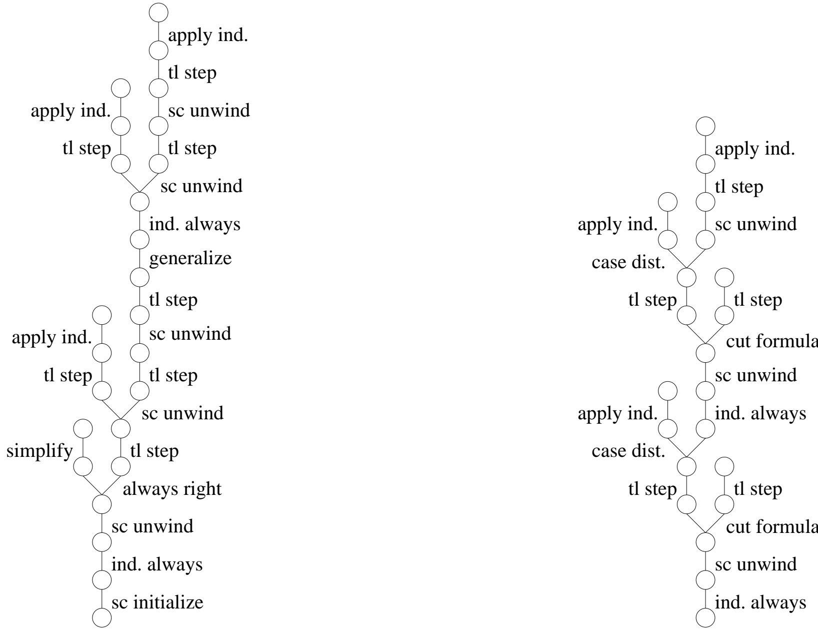
sc unwind Zum Beenden des Beweises müssen wir das Statechart noch einmal ausführen, um wieder in den Ausgangszustand zu gelangen.

$$\frac{\dfrac{simplify,\ tl\ step,\ apply\ induction\ (IndHyp_1)}{\begin{array}{ll} \ldots x'' = 0 \wedge (\circ\, Open \wedge \ldots) & \vdash \Box x \leq 6 \\ x = 6, Close,\ \circ\, SYS & \end{array}}}{x = 6, Close, SYS \vdash \Box x \leq 6}$$

Da wir in *Close* sind und $x > 5$ gilt, wird auf jeden Fall der Übergang nach *Open* ausgeführt, d. h. in allen Nachfolgekonfigurationen gilt *Open* und $x = 0$. Damit haben wir wieder die Ausgangskonfiguration erreicht und können den Beweis per Induktion mit $IndHyp_1$ schließen. Der aus den Ableitungschritten entstandene Beweisbaum ist in Abbildung 6.3 b) dargestellt.

Ergebnis

Im Vergleich zum Beweis mit einer vollständigen Zustandskonfiguration erfordert der Beweis mit partieller Zustandskonfiguration mehr Verständnis für den Beweis, denn der Benutzer bestimmt, welche Eigenschaften für den weiteren Beweis wesentlich sind und fügt diese Eigenschaft in den Beweis ein. Dies geschieht z. B. in der Ableitung (1). Jedoch erhalten wir allgemeinere Invarianten – im Beispiel sind sie unabhängig davon ob *Light* in *On* oder *Off* ist – und wir mussten deshalb nur 3 Statechart-Schritte ausführen. Beim ersten Beweis waren 5 Statechart-Schritte notwendig. Dafür sind Beweise mit vollständiger Zustandskonfiguration besser automatisierbar. So kann obiger Beweis mit vollständiger Zustandskonfiguration (ohne der Generalisierung von $x = 0$ auf $x \leq 5$) in KIV automatisch geführt werden. Zum Vergleich haben wir die Beweisbäume beider Beweise in Abbildung 6.3 gegenübergestellt. Vereinfachungsschritte wurden nicht mit aufgeführt, um die



a) vollständige Zustandskonfiguration   b) partielle Zustandskonfiguration

Abbildung 6.3: Beweisbäume

Beweisbäume übersichtlich zu halten. Diese verdeutlichen nochmals, dass der Beweis für die vollständige Zustandskonfiguration „tiefer“ ist und mehr Statechart-Schritte benötigt.

Die intuitiven Schritte beim Beweis mit partiellen Zustandskonfigurationen kommen bei der *cut*-Regel zum Tragen.

Zusammenfassend ist zu bemerken, dass für einfachere Beweise, die vollständig automatisch geführt werden, der Ansatz der vollständigen Zustandskonfiguration besser geeignet ist. Für kompliziertere Beweise mit vielen Zuständen eignet sich der Ansatz mit partiellen Zustandskonfigurationen.

6.3.3 Kalkül-Regel für Statecharts

Wir haben das Regelschema der *sc unwind*-Regel kennengelernt und die Anwendung für vollständige und partielle Zustandskonfigurationen gezeigt.

$$\frac{\bigvee_{i=1}^{n} cond(stp_i) \wedge step(stp_i) \wedge \circ\, nxt(stp_i), \Gamma \vdash \Delta}{SC, \Gamma \vdash \Delta}\ sc\ unwind$$

Dabei haben wir auf die exakte Definition der Berechnung und Ausführung der Übergangsmengen stp_i verzichtet. Dies wollen wir nun nachholen. Wie schon bemerkt, sind die Statechart-Variablen flexible Variablen, die wir durch einen Punkt $\dot{v} \in variables(SC)$ kennzeichnen. Die Hilfsvariablen $\widetilde{variables}(SC)$ werden immer nur für eine Schrittausführung benötigt und können deshalb auch rigide sein. Wir vereinbaren, dass für $\dot{v} \in variables(SC)$ $v \in \widetilde{variables}(SC)$ die entsprechende Hilfsvariable ist. Zur Definition der Kalkülregel werden ähnliche Hilfsfunktionen wie in der Semantikdefinition von Statecharts benötigt. Wir vereinbaren die Konvention, dass eine Funktion f aus der Semantikbeschreibung, die wir für die Kalkülregel entsprechend abändern müssen, mit f_K bezeichnet wird.

Ausführungsbedingungen und Berechnung der Übergangsmengen

In unserem Ansatz beschreiben die Formeln einer Sequenz mögliche Belegungen. Ob eine Ausführungsbedingung in diesen Belegungen gilt, kann häufig nicht automatisch bestimmt werden, sondern erfordert interaktive Beweisschritte. Deshalb berechnen wir neben den potentiellen Übergangsmengen auch entsprechende Ausführungsbedingungen. Nur wenn eine Ausführungsbedingung gilt, wird die entsprechende Übergangsmenge ausgeführt. Dazu erweitern wir die Berechnung der Übergangsmengen aus Abschnitt 4.2.2, Seite 42. Außerdem muss die Berechnung der Statechart-Schritte auch partielle Konfigurationen berücksichtigen.

Die Funktion $steps_K : Fma \times states(SC) \times Fma \rightarrow \wp((Fma, \wp(trans(SC))))$ zur Schrittberechnung für ein Statechart SC erhält im Vergleich zur Funktion $steps : \Sigma(variables(SC), \times states(SC) \rightarrow \wp(\wp(trans(SC)))$ aus Abschnitt 4.2.2 ein zusätzliches Argument, die (vorläufige) Aktivierungsbedingung g. Für eine Sequenz $\Gamma \vdash \Delta$ sei $\Gamma \wedge \neg\Delta := \bigwedge\Gamma \wedge \neg \bigvee\Delta$. Im Gegensatz zu $steps$ wird $steps_K$ von einer Kalkülregel aufgerufen und basiert für eine Sequenz $\Gamma \vdash \Delta$ auf der symbolischen Beschreibung $\Gamma \wedge \neg\Delta$ der Variablenbelegung, anstatt auf einer Belegung $\sigma \in \Sigma(variables(SC))$. Da aus $\Gamma \wedge \neg\Delta$ die Variablenbelegung meist

nicht eindeutig bestimmt werden kann, liefert $steps_K$ eine Menge von Tupeln aus Aktivierungsbedingung und Übergangsmenge. Nur wenn die Aktivierungbedingung erfüllt ist, kann der Übergang ausgeführt werden.

1. $mode(s) \in \{BASIC, TERM\}$:
   $steps_K(\Gamma \wedge \neg\Delta, s, g) := \{(g, \emptyset)\}$
2. $mode(s) = AND$:
   Sei $\{s_1, \ldots, s_n\} = childs(s)$, dann ist $steps_K(\Gamma \wedge \neg\Delta, s, g) :=$
   $$\{(g_1 \wedge \ldots \wedge g_n, \bigcup_{i=1}^{n} T_i) \mid (g_i, T_i) \in steps_K(\Gamma \wedge \neg\Delta, s_i, s_i \wedge g)\}$$
3. $mode(s) = OR$: Sei
   - $S' = \{\tilde{s} \mid \tilde{s} \in childs(s)$ und nicht $\Gamma \wedge \neg\, \Delta \rightarrow \neg\, \tilde{s}\}$,
   - $S^* = \{\tilde{s} \mid \tilde{s} \in childs^*(s)$ und nicht $\Gamma \wedge \neg\, \Delta \rightarrow \neg\, \tilde{s}\}$,
   - $T_{\tilde{s}} = \{t_{\tilde{s}_1}, \ldots, t_{\tilde{s}_k}\}$
     $= \{t \mid scope(t) = s,\ source(t) \in childs^*(\tilde{s})$ und $source(t) \in S^*\}$ und
   - $T = \{t_1, \ldots, t_k\} = \bigcup_{\tilde{s} \in S'} T_{\tilde{s}}$.

   Da alle Übergänge in T dieselbe Priorität haben, stehen sie in Konflikt und

$$steps_K(\Gamma \wedge \neg\Delta, s, g) :=$$
$$\{(g_t, \{t\}) | t \in T \text{ und } g_t := g \wedge guard(t) \wedge source(t)\} \cup \quad (3)$$
$$\bigcup_{s' \in S'} steps_K(\Gamma \wedge \neg\Delta, s', g \wedge s' \wedge \neg guard(t_{s'_1}) \wedge \ldots \wedge \neg guard(t_{s'_k})) \quad (4)$$

Die wesentliche Änderung im Algorithmus im Vergleich zur *steps*-Berechnung aus Abschnitt 4.2.2 befindet sich im Schritt 3 bei der Betrachtung eines *Oder*-Zustands. Betrachten wir uns zuerst einmal, von welchen Zuständen Übergänge ausgehen können. Da wir aus $\Gamma \wedge \neg\Delta$ nicht immer die aktiven Zustände folgern können – insbesondere nicht bei partiellen Zustandskonfigurationen –, können von all denjenigen Zuständen $\tilde{s}$ aus Übergänge stattfinden, die in $\Gamma \wedge \neg\Delta$ nicht zu false ausgewertet werden. Dazu versuchen wir mit einer automatischen Vereinfachungsstrategie $\Gamma \wedge \neg\Delta \rightarrow \neg\, \tilde{s}$ zu zeigen. Gelingt dies nicht, kann $\tilde{s}$ möglicherweise aktiv sein. Das sind alle Zustände in S'. Von diesen Zuständen können auch aus aktiven Unterzuständen Übergänge ausgehen, wenn die Übergänge in der aktuellen Region $scope(s)$ liegen (die Region liefert eine „Priorisierung" von Übergängen). Daraus ergibt sich für einen Zustand $\tilde{s}$ die Übergangsmenge $T_{\tilde{s}}$. $T_{\tilde{s}}$ enthält alle aus $\tilde{s}$ möglichen Übergänge. T fasst alle Übergänge zusammen, deren Ausgangszustand möglicherweise aktiv ist. Diese Übergänge stehen in Konflikt und können dann ausgeführt werden, wenn ihre Aktivierungsbedingung gilt und der entsprechende Ausgangszustand aktiv ist.

Deshalb ist die Aktivierungsbedingung g_t für einen solchen Übergang t in Gleichung (3): $g_t := g \wedge \mathit{guard}(t) \wedge \mathit{source}(t)$. Die Bedingung, dass der Ausgangszustand aktiv ist, müssen wir hinzufügen, da für die Übergänge $t \in T$ nicht folgt, dass der entsprechende Ausgangszustand $\mathit{source}(t)$ aktiv ist, sondern nur, dass er nicht inaktiv ist. Zusätzlich muss noch die Vorbedingung g, die der Berechnung übergeben wird, gelten. Die Vorbedingung g beschreibt, dass kein höher priorisierter Übergang möglich ist.

Für den Fall, dass ein Zustand $\tilde{s}$ aktiv ist, aber alle Aktivierungsbedingungen seiner Übergänge in der aktuellen Konfiguration nicht gelten, können Unterzustände weitere Übergänge aktivieren. Deshalb fügen wir für einen Zustand $\tilde{s}$, unter der Vorbedingung dass alle Übergänge aus $T_{\tilde{s}}$ nicht aktiviert sind, die möglichen Übergänge seiner Unterzustände hinzu. Dies geschieht in Gleichung (4) für alle möglicherweise aktiven Unterzustände von s.

Beispiel 6.4 *Schrittberechnung I*
Sei die Steuerung der Zeitschaltuhr SYS *aus Abbildung 6.1 in den Ausgangszuständen* Off *und* Open, $x = 0$ *und wir zeigen* $\Box x \leq 6$. *Wir erhalten die Sequenz*

$$\mathrm{SYS}, \mathrm{System}, \mathrm{Light}, \mathrm{Timer}, \mathrm{Off}, \neg\ \mathrm{On}, \mathrm{Open}, \neg\ \mathrm{Close}, x = 0 \vdash \Box x \leq 6,$$

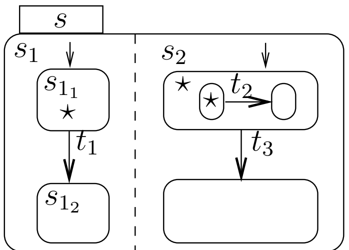
die wir mit $\mathrm{SYS}, \Gamma \vdash \varphi$ *abkürzen. In* Light *gilt* $S' = \{\mathrm{Off}\}$ *(da* $\Gamma \rightarrow \neg\mathrm{On}$ *gilt) und es können die folgenden Schritte ausgeführt werden:*

$$\mathrm{steps}_K(\Gamma \wedge \neg\varphi, \mathrm{Light}, \mathrm{Light}) = \{(\mathrm{Light} \wedge \mathrm{Off} \wedge \mathrm{press}, \{t_1\}), (\mathrm{Light} \wedge \mathrm{Off} \wedge \neg\mathrm{press}, \emptyset)\}$$

Analog ergibt sich für Timer *und* System*:*

$$\begin{aligned}
\mathrm{steps}_K(\Gamma \wedge \neg\varphi, \mathrm{Timer}, \mathrm{Timer}) = \{&(\mathrm{Timer} \wedge \mathrm{Open} \wedge \mathrm{set}, \{t_3\}), (\mathrm{Timer} \wedge \mathrm{Open} \wedge \neg\mathrm{set}, \emptyset)\} \\
\mathrm{steps}_K(\Gamma \wedge \neg\varphi, \mathrm{System}, \mathrm{true}) = \{&(\mathrm{Light} \wedge \mathrm{Off} \wedge \mathrm{press} \wedge \mathrm{Timer} \wedge \mathrm{Open} \wedge \mathrm{set}, \{t_1, t_3\}), \\
&(\mathrm{Light} \wedge \mathrm{Off} \wedge \mathrm{press} \wedge \mathrm{Timer} \wedge \mathrm{Open} \wedge \neg\mathrm{set}, \{t_1\}), \\
&(\mathrm{Light} \wedge \mathrm{Off} \wedge \neg\mathrm{press} \wedge \mathrm{Timer} \wedge \mathrm{Open} \wedge \mathrm{set}, \{t_3\}), \\
&(\mathrm{Light} \wedge \mathrm{Off} \wedge \neg\mathrm{press} \wedge \mathrm{Timer} \wedge \mathrm{Open} \wedge \neg\ \mathrm{set}, \emptyset)\}
\end{aligned}$$

Beispiel 6.5 *Schrittberechnung II*



Betrachten wir das nebenstehende Statechart SC, in dem die mit $\star$ *markierten Zustände aktiv sind. Für die Übergänge sei* $\mathrm{guard}(t_1) = g_1$, $\mathrm{guard}(t_2) = g_2$ *und* $\mathrm{guard}(t_3) = g_3$. *Aktive Zustände führen wir nicht explizit in den Aktivierungsbedingungen auf, d. h. für die Aktivierungsbedingung von* t_1 *schreiben wir einfach* g_1 *statt* $g_1 \wedge st_{1_1}$. *Dann gilt:*

$$\begin{aligned}
\mathrm{steps}_K(\Gamma \wedge \neg\Delta, s_1, \mathrm{true}) = &\quad \{(g_1, \{t_1\}), (\neg g_1, \emptyset)\} \\
\mathrm{steps}_K(\Gamma \wedge \neg\Delta, s_2, \mathrm{true}) = &\quad \{(g_3, \{t_3\}), (\neg g_3 \wedge g_2, \{t_2\}), (\neg g_3 \wedge \neg g_2, \emptyset)\} \\
\mathrm{steps}_K(\Gamma \wedge \neg\Delta, \mathrm{root}(SC), \mathrm{true}) = &\ \mathrm{steps}_K(\Gamma \wedge \neg\Delta, s, \mathrm{true}) = \\
&\{(g_1 \wedge g_3, \{t_1, t_3\}), (\neg g_1 \wedge g_3, \{t_3\}), (g_1 \wedge \neg g_3 \wedge g_2, \{t_1, t_2\}), \\
&(\neg g_1 \wedge \neg g_3 \wedge g_2, \{t_2\}), (g_1 \wedge \neg g_3 \wedge \neg g_2, \{t_1\}), (\neg g_1 \wedge \neg g_3 \wedge \neg g_2, \emptyset)\}
\end{aligned}$$

Betrachten wir nun s mit einer generalisierten Zustandskonfiguration in der von den konkreten Zuständen in s_1 abstrahiert wurde. Es kann nun entweder s_{1_1} oder s_{1_2} aktiv sein und wir erhalten:

$$\begin{aligned}
&\mathrm{steps}_K(\Gamma \wedge \neg\Delta, s_1, \mathrm{true}) = \quad \{(g_1 \wedge s_{1_1}, \{t_1\}), (\neg g_1 \wedge s_{1_1}, \emptyset), (s_{1_2}, \emptyset)\}\\
&\mathrm{steps}_K(\Gamma \wedge \neg\Delta, s_2, \mathrm{true}) = \quad \{(g_3, \{t_3\}), (\neg g_3 \wedge g_2, \{t_2\}), (\neg g_3 \wedge \neg g_2, \emptyset)\}\\
&\mathrm{steps}_K(\Gamma \wedge \neg\Delta, \mathrm{root}(SC), \mathrm{true}) = \mathrm{steps}_K(\Gamma \wedge \neg\Delta, s, \mathrm{true}) =\\
&\quad \{(g_1 \wedge s_{1_1} \wedge g_3, \{t_1, t_3\}), (\neg g_1 \wedge s_{1_1} \wedge g_3, \{t_3\}), (s_{1_2} \wedge g_3, \{t_3\}),\\
&\quad (g_1 \wedge s_{1_1} \wedge \neg g_3 \wedge g_2, \{t_1, t_2\}), (\neg g_1 \wedge s_{1_1} \wedge \neg g_3 \wedge g_2, \{t_2\}), (s_{1_2} \wedge \neg g_3 \wedge g_2, \{t_2\}),\\
&\quad (g_1 \wedge s_{1_1} \wedge \neg g_3 \wedge \neg g_2, \{t_1\}), (\neg g_1 \wedge s_{1_1} \wedge \neg g_3 \wedge \neg g_2, \emptyset), (s_{1_2} \wedge \neg g_3 \wedge \neg g_2, \emptyset)\}
\end{aligned}$$

Bemerkungen zur Berechnung der Übergangsmengen Bei der Berechnung möglicher Schritte wird nicht überprüft, ob die Aktivierungsbedingungen erfüllt sind oder nicht, sondern die Aktivierungsbedingungen werden als Vorbedingungen für einen Schritt in die Prämissen aufgenommen. Sind Aktivierungsbedingungen nicht erfüllt, erhält man widersprüchliche Prämissen, die trivial geschlossen werden. Die entsprechenden Übergänge werden nicht ausgeführt.

Statische Reaktionen Bei der Ausführung eines Schrittes müssen wir noch die statischen Reaktionen berücksichtigen. Für einen Schritt $(g, T) \in \mathit{steps}_K(\Gamma \wedge \neg\Delta, \mathit{root}, \mathrm{true})$ bestimmt die Ausführungsbedingung g und die Übergangsmenge T die geänderten, aktiven und neu betretenen Zustände mit $\mathit{changed}_K : \mathit{trans}(SC) \to \wp(\mathit{states}(SC))$, $\mathit{active}_K : \mathit{Fma} \times \mathit{trans}(SC) \to \wp(\mathit{states}(SC))$ und $\mathit{entered}_K : \mathit{Fma} \times \mathit{trans}(SC) \to \wp(\mathit{states}(SC))$.

$$\begin{aligned}
\mathit{changed}_K(T) &:= \{s \in \mathit{states}(SC) \mid \exists\, t.\ t \in T \text{ mit } \mathit{scope}(t) < s\}\\
\mathit{active}_K(g, T) &:= \{s \in \mathit{states}(SC) \mid g \to s \text{ oder } s = \mathit{root}\} \backslash \mathit{changed}_K(T)\\
\mathit{entered}_K(g, T) &:= \mathit{decompl}(\mathit{active}_K(g, T) \cup \textstyle\bigcup_{t \in T}\{\mathit{target}(t)\}) \backslash \mathit{active}_K(g, T)
\end{aligned}$$

$\mathit{changed}_K(T)$ beschreibt die Menge von Übergängen, deren Zustand sich ändert, die also entweder verlassen oder betreten werden. Die aktiven Zustände $\mathit{active}_K(g, T)$ ergeben sich nach Konstruktion in $\mathit{steps}_K(\Gamma \wedge \neg\Delta, \mathit{root}(SC), \mathrm{true})$ eindeutig aus der Aktivierungsbedingung g eines Statechart-Schrittes und die neu betretenen Zustände $\mathit{entered}_K$ berechnen sich analog zu *entered* aus der Semantikdefinition.

Beispiel 6.6 *Aktive Zustände*
Betrachten wir den zweiten Übergang $(\mathrm{Light} \wedge \mathrm{Off} \wedge \mathrm{press} \wedge \mathrm{Timer} \wedge \mathrm{Open} \wedge \neg\mathrm{set}, \{t_1\})$ *aus Beispiel 6.4 und kürzen ihn mit* $(g_2, \{t_1\})$ *ab. Es gilt:*

$$\begin{aligned}
\mathrm{changed}_K(\{t_1\}) &= \{\mathrm{Off}, \mathrm{On}\}\\
\mathrm{active}_K(g_2, \{t_1\}) &= \{\mathrm{System}, \mathrm{Light}, \mathrm{Off}, \mathrm{Timer}, \mathrm{Open}\} \backslash \{\mathrm{Off}, \mathrm{On}\}\\
&= \{\mathrm{System}, \mathrm{Light}, \mathrm{Timer}, \mathrm{Open}\}\\
\mathrm{entered}_K(g_2, \{t_1\}) &= (\{\mathrm{System}, \mathrm{Light}, \mathrm{Off}, \mathrm{Timer}, \mathrm{Open}\} \cup \{\mathrm{On}\}) \backslash\\
&\qquad \{\mathrm{System}, \mathrm{Light}, \mathrm{Off}, \mathrm{Timer}, \mathrm{Open}\}\\
&= \{\mathrm{On}\}
\end{aligned}$$

Nun können wir mit $s \in active_K(g,T)$ und $sr \in sreact(s)$ alle potentiell ausführbaren statischen Reaktionen sr bestimmen. Wie bei Übergängen wird die Aktivierungsbedingung $guard(sr)$ einer statischen Reaktion bei der Berechnung nicht ausgewertet, sondern als Vorbedingung für einen Statechart-Schritt zurückgeliefert. Ist $guard(sr)$ erfüllt, wird die Aktion $action(sr)$ ausgeführt, sonst nicht. Für eine statische Reaktion $sr \in sreact(s)$, die im Zustand s definiert ist, definieren wir

$$SR(sr) := \{(guard(sr), \{action(sr)\}), (\neg\ guard(sr), \emptyset)\}.$$

Für eine Übergangsmenge (g,T) ergeben sich mehrere mögliche Ausführungspfade, je nachdem, welche statischen Reaktionen zusätzlich zu den Übergängen aktiviert sind. Die Menge der *vervollständigten* Übergänge enthält die statischen Reaktionen. Mit $actions(T) := \bigcup_{t \in T} action(t)$ ist $action_set_K : (Fma, trans(SC)) \to \wp(Fma, \wp(Prog))$ definiert.

$$action_set_K((g,T)) := \{(g, actions(T))\} \uplus \biguplus_{sr_i \in sreact(s), s \in active_K(g,T)} SR(sr_i)^2$$

die Menge vervollständigter Übergänge. Für einen Statechart SC und eine Sequenz $\Gamma \vdash \Delta$ berechnet $Steps_K : Fma \to \wp((Fma, \wp(trans(SC)), \wp(actions(SC))))$ mit

$$Steps_K(\Gamma \wedge \neg\Delta) := \\ \{(g, T, \mathcal{ACT}) \mid (g, \mathcal{ACT}) \in action_set_K((g', T)), (g', T) \in steps_K(\Gamma \vdash \Delta, root, \mathrm{true})\}$$

die Menge der vervollständigten Übergangsmengen, die wir für einen Statechart-Schritt ausführen müssen.

Ausführungsbedingung

Für $stp = (g, T, \mathcal{ACT}) \in Steps_K(\Gamma \wedge \neg\Delta)$ ist $cond(stp) = g$ die Ausführungsbedingung für die berechnete Übergangsmenge.

Ausführen eines Statechart-Schrittes

Wir beschreiben einen Statechart-Schritt durch ein DL-Programm, das die Relation ρ_{step} aus Abschnitt 4.2.2 implementiert. Die Programm-Formeln

$$step, step_{mikro}, step_{makro} : (Fma, \wp(trans(SC)), \wp(actions(SC))) \to Fma$$

berechnet für einen vervollständigten Übergang $stp = (g, T, \mathcal{ACT}) \in Steps_K(\Gamma \wedge \neg\Delta)$ mit

$$step((g, T, \mathcal{ACT})) := \\ \begin{cases} T = \emptyset, & step_{makro}((g, T, \mathcal{ACT})) \\ \text{sonst}, & step_{mikro}((g, T, \mathcal{ACT})) \end{cases}$$

die Variablenwerte, die sich durch den Statechart-Schritt ergeben. Ist die berechnete Übergangsmenge leer, sind also keine Übergänge möglich, berechnet $step(stp)$ einen Makro-Schritt, ansonsten führt ein Mikro-Schritt die möglichen Übergänge aus.

[2] Mit $SR(sr_1) \uplus SR(sr_2) := \{(g_1 \wedge g_2, a_1 \cup a_2) \mid (g_1, a_1) \in SR(sr_1), (g_2, a_2) \in SR(sr_2)\}$.

Mikro-Schritt Wir definieren nun einen Mikro-Schritt für eine vervollständigte Übergangsmenge $stp = (g, T, \mathcal{ACT})$.

$$step_{mikro}(stp) := \bigvee_{\vec{\alpha} \in perm(\mathcal{ACT})} \langle copy_K; reset^l_K; reset^e_K; exec_K(\vec{\alpha}); upd_ex_K(T); upd_en_K(g, T)\rangle set_K$$

Analog zur Semantikdefinition in Abschnitt 4.2.2 unterteilen wir die Ausführung eines Mikro-Schrittes in Teilprogramme.

$$\begin{array}{lll}
copy_K & := & v_1 := \dot{v}_1 \mid \ldots \mid v_n := \dot{v}_n, \text{ für } \dot{v}_n \in vars(SC) \cup events(SC) \\
reset^l_K & := & e_1 := \text{false} \mid \ldots \mid e_n := \text{false}, \text{ für} \\
 & & \dot{e}_i \in events_{loc}(SC) \cup events_{imp}(SC) \cup \{tick\} \\
reset^e_K & := & \textbf{if } \dot{tick} \textbf{ then } e_1 := \text{false} \mid \ldots \mid e_n := \text{false}, \\
 & & \text{für } \dot{e}_i \in events_{env}(SC) \\
exec_K(\vec{\alpha}) & := & \alpha_1; \ldots; \alpha_n \text{ für } \vec{\alpha} =< \alpha_1, \ldots, \alpha_n > \\
upd_ex_K(T) & := & e_{s_1} := \dot{s}_1 \mid \ldots \mid e_{s_n} := \dot{s}_n \text{ für} \\
 & & \dot{e}_{s_i} \in \{e_s \mid e_s \equiv ex(s) \wedge s \in changed_K(T)\} \\
upd_en_K(g, T) & := & e_1 := \text{true} \mid \ldots \mid e_n := \text{true für} \\
 & & \dot{e}_i \in \{e \mid e \equiv en(s) \wedge s \in entered_K(g, T)\} \\
set_K & := & \bigwedge_i x_i = x''_i \text{ für } \dot{x}_i \in vars(SC) \cup events(SC)
\end{array}$$

Am Anfang eines jeden Schrittes werden die Variablenwerte kopiert, um während der Ausführung eines Statechart-Schrittes auf die Werte der Variablen *vor* Beginn der Schrittausführung zurückgreifen zu können. Wir führen die Programme dann auf Kopien der Variablen aus. Nun müssen wir die Ereignisse zurücksetzen. Das Programm $reset^l_K$ weist dazu allen lokalen Ereignissen (den Variablenkopien) eines Statecharts false zu. Umgebungsereignisse müssen dann zurückgesetzt werden, wenn *tick* den ersten Mikro-Schritt nach einem Makro-Schritt kennzeichnet. Dies geschieht durch $reset^e_K$. Danach werden die Aktionen $\vec{\alpha}$ der Übergänge und statischen Reaktionen durch $exec_K(\vec{\alpha})$ ausgeführt. Die Programme weisen nur den Hilfsvariablen neue Werte zu und überschreiben deshalb nicht die gepunkteten Statecharts-Variablen. Greifen wir auf gepunktete Variablen $\dot{v}$ zu, erhalten wir also immer den Wert der Variablen vor Beginn der Schrittausführung. Deshalb können wir die Programme α_i sequentiell ausführen und erhalten trotzdem die Berechnung der Werte nach der STATEMATE-Semantik.

Die Semantik von Statecharts erlaubt, Aktionen in beliebiger Reihenfolge auszuführen (siehe Abschnitt 4.2.2, Gleichung 1). Damit werden Schreibkonflikte zwischen Aktionen (zwei Aktionen, die in einem Schritt ausgeführt und dieselbe Variable schreiben) indeterministisch aufgelöst. Wir berücksichtigen dies in der Berechnung eines Mikro-Schrittes, indem jede Permutation der Aktionen $\vec{\alpha} \in perm(\mathcal{ACT})$ der auszuführenden Aktionen $\mathcal{ACT}$ einen separaten Schritt berechnet. Nach dem Ausführen der Aktionen werden mit $upd_ex_K(T); upd_en_K(g, T)$ abgeleitete Ereignisse, die das Verlassen und Betreten von Zuständen kennzeichnen, gesetzt. Ereignisse, die das Verlassen von Zustände kennzeichnen, müssen dann gesetzt werden, wenn ein aktiver Zustand, deren entsprechende boolesche Zustandsvariable wahr ist, verlassen wird. Dazu berechnen wir die Zustände, die sich geändert

haben ($changed_K(T)$) und setzen die Ereignisvariablen auf den Wert der entsprechenden Zustandsvariablen.

Nach dem Ausführen der Schrittberechnung durch Abarbeiten des Programms verlangen wir, dass die berechneten Variablenwerte den 2-fach gestrichenen Variablen entsprechen. Dies geschieht durch die Bedingung set_K, die so die Relation zur Nachfolgebelegung herstellt. Die Statechart-Semantik beschreibt in ρ_{step} auch das Verhalten der Umgebung. In einem Mikro-Schritt werden keine Umgebungsereignisse und -variablen verändert und die lokalen Ereignisse und Variablen können von der Umgebung nicht beeinflusst werden. Deshalb weisen wir die Werte nach der Ausführung des Statechart-Schrittes direkt den 2-fach gestrichenen Variablen zu und schließen so eine weitere Beeinflussung der Variablen durch die Umgebung aus.

Makro-Schritt Wir definieren nun einen Makro-Schritt, den wir durch $tick' = true$ und $tick'' = true$ kennzeichnen, für eine vervollständigte Übergangsmenge $stp = (g, T, \mathcal{ACT})$.

$$step_{makro}(stp) := \langle copy_K; reset^l_K \rangle set^e_K \wedge set^l_K$$

Wie in einem Mikro-Schritt werden die Variablenwerte in die Hilfsvariablen kopiert und die lokalen Ereignisse zurückgesetzt. Ein Systemschritt weist als Ausgabe des Makro-Schritts den 1-fach gestrichenen Umgebungsereignissen und -variablen die aktuellen Werte zu. Der folgende Umgebungsschritt kann nun die 2-fach gestrichenen Umgebungsvariablen beliebig beeinflussen und so beliebige Eingaben produzieren.

$$set^e_K \quad := \quad tick' = \text{true} \wedge tick'' = \text{true} \wedge \bigvee_{\dot{v} \in events_{env}(SC) \,\cup\, vars_{env}(SC)} v' = v$$

Die Werte der lokalen Ereignisse und Variablen werden wieder den 2-fach gestrichenen Variablen zugewiesen. Diese können von der Umgebung nicht geändert werden.

$$set^l_K \quad := \quad \bigvee_{\dot{v} \in events_{loc}(SC) \,\cup\, events_{imp}(SC) \,\cup\, vars_{loc}(SC)} v'' = v$$

Wie oben definiert, wird ein Makro-Schritt nur in einer stabilen Konfiguration ausgeführt, wenn also die Menge der konfliktfreien Übergänge leer ist. Umgebungsereignisse müssen im Makro-Schritt nicht zurückgesetzt werden, da ihnen neue, zufällige Werte zugewiesen werden.

Die 1-fach gestrichenen Umgebungsvariablen enthalten nun die im Makro-Schritt berechneten Ergebniswerte. Mit $\Box tick' \rightarrow \varphi'$ können wir nun Eigenschaften φ über die Berechneten Ergebnisse von Makro-Schritten nachweisen. Andererseits erlaubt uns $\Box tick \rightarrow \varphi$ Annahmen über die Umgebung zu machen. In jedem Makro-Schritt muss die Umgebung φ erfüllen.

Beispiel 6.7 *Schrittausführung*
Betrachten wir nochmals die Zeitschaltuhr SYS *aus Abbildung 6.1 und die Sequenz*

$$\text{SYS}, \text{Sys}\dot{\text{t}}\text{em}, \text{Li}\dot{\text{g}}\text{ht}, \text{Ti}\dot{\text{m}}\text{er}, \dot{\text{O}}\text{ff}, \neg\, \dot{\text{O}}\text{n}, \text{O}\dot{\text{p}}\text{en}, \neg\, \text{Cl}\dot{\text{o}}\text{se}, \dot{x} = 0 \vdash \Box \dot{x} \leq 6,$$

die wir mit SYS, $\Gamma \vdash \varphi$ *abkürzen. Man beachte, dass hier die flexiblen Variablen explizit durch* $\dot{}$ *gekennzeichnet sind. Die Berechnung der möglichen Übergangsmengen aus Beispiel 6.4 ergibt die vier Fälle* $(g_1, \{t_1, t_3\}, \{\text{set} := \text{true}\})$, $(g_2, \{t_1\}, \{\text{set} := \text{true}\})$,$(g_3, \{t_3\}, \emptyset$ *und* $(g_4, \emptyset, \emptyset)$ *und damit vier Prämissen für die Regel* sc unwind*.*

$$\frac{\begin{array}{ll}(1) & g_1 \wedge step(g_1, \{t_1, t_3\}, \{\text{set} := \text{true}\})) \wedge \circ nxt(\{t_1, t_3\}), \Gamma \vdash \varphi \\ (2) & g_2 \wedge step((g_2, \{t_1\}, \{\text{set} := \text{true}\})) \wedge \circ nxt(\{t_3\}), \Gamma \vdash \varphi \\ (3) & g_3 \wedge step((g_3, \{t_3\}, \emptyset)) \wedge \circ nxt(\{t_1\}), \Gamma \vdash \varphi \\ (4) & g_4 \wedge step((g_4, \emptyset, \emptyset)) \wedge \circ nxt(\emptyset), \Gamma \vdash \varphi\end{array}}{\text{SYS}, \Gamma \vdash \varphi}$$

Wir betrachten nur die Fälle 2 und 4 genauer. Der 2. Fall ist ein Mikro-Schritt, der 4. ein Makro-Schritt. Das Kopieren der flexiblen Variablen und das Rücksetzen der Ereignisse ist unabhängig vom konkreten Schritt.

$$\begin{array}{rl}\text{copy}_K = & tick := \dot{tick} \mid set := \dot{set} \mid \text{sw_off} = \dot{\text{sw_off}} \mid x := \dot{x} \mid \text{en_On} := \dot{\text{en_On}} \mid \\ & \text{ex_On} := \dot{\text{ex_On}} \mid \text{en_Off} := \dot{\text{en_Off}} \mid \text{ex_Off} := \dot{\text{ex_Off}} \\ \text{reset}^l_K = & tick := \text{false} \mid set := \text{false} \mid \text{sw_off} = \text{false} \mid en\text{_On} := \text{false} \mid \\ & ex\text{_On} := \text{false} \mid en\text{_Off} := \text{false} \mid ex\text{_Off} := \text{false} \\ \text{reset}^e_K = & \textbf{\textit{if}}\ \dot{\text{tick}}\ \textbf{\textit{then}}\ \text{press} := \text{false}\end{array}$$

Im Mikro-Schritt muss der Übergang t_1 *ausgeführt werden. Statische Reaktionen finden keine statt. Anschließend erzeugen wir die sich aus dem Mikro-Schritt ergebenden abgeleiteten Ereignisse (* en_.. *bzw.* en_..*). Schließlich müssen zur Definition der nächsten Belegung den 2-fach gestrichenen Variablen die Werte der Schrittberechnung zugewiesen werden.*

$$\begin{array}{rll}\text{exec}_K(< set := \text{true} >) & = & set := \text{true} \\ \text{upd_ex}_K(\{t_1\}) & = & ex\text{_Off} := \dot{\text{Off}} \mid \text{ex_On} := \dot{\text{On}} \\ \text{upd_en}_K(g_2, \{t_1\}) & = & en\text{_On} := \text{true} \\ \text{set}_K & = & tick = tick'' \wedge set = set'' \wedge press = press'' \wedge x = x'' \wedge \\ & & \text{sw_off} = \text{sw_off}'' \wedge en\text{_On} = en\text{_On}'' \wedge ex\text{_On} = \\ & & ex\text{_On}'' \wedge en\text{_Off} = en\text{_Off}'' \wedge ex\text{_Off} = ex\text{_Off}''\end{array}$$

In upd_ex_K *sieht man, dass nur das Ereignis* ex_On *erzeugt wird, denn* $\dot{\text{Off}}$ *ist* false *und damit wird* ex_Off false *zugewiesen. Der gesamte Mikro-Schritt der 2. Prämisse ist dann*

$$\begin{array}{r}\text{step}_{mikro}((g_2, \{t_1\}, \{set\})) = \langle \text{copy}_K; \text{reset}^l_K; \text{reset}^e_K; \text{exec}_K(< set := \text{true} >); \\ \text{upd_ex}_K(\{t_1\}); \text{upd_en}_K(\dot{press} \wedge \neg \dot{set}, \{t_1\}) \rangle \text{set}_K.\end{array}$$

Im Makro-Schritt der 4. Prämisse werden alle Ereignisse zurückgesetzt, das tick*-Ereignis erzeugt und dann die Umgebungsereignisse und -variablen eingelesen.*

$$\begin{array}{rl}\text{set}^e_K = & \text{tick}' := \text{true} \wedge \text{tick}'' := \text{true} \wedge \text{press}' = \text{press} \\ \text{set}^l_K = & set = set'' \wedge x = x'' \wedge \text{sw_off} = \text{sw_off}'' \wedge en\text{_On} = en\text{_On}'' \wedge \\ & ex\text{_On} = ex\text{_On}'' \wedge en\text{_Off} = en\text{_Off}'' \wedge ex\text{_Off} = ex\text{_Off}''\end{array}$$

Daraus ergibt sich der Makro-Schritt

$$\mathrm{step}_{makro}((\neg \dot{press} \wedge \neg \dot{set}, \emptyset, \emptyset)) = \langle \mathrm{copy}_K; \mathrm{reset}^l_K \rangle \mathrm{set}^e_K \wedge \mathrm{set}^l_K.$$

Neue Zustandskonfiguration Zusätzlich zur Schrittberechnung muss bei der Abwicklung eines Statechart-Schrittes die Zustandskonfiguration für die Nachfolgebelegung berechnet werden. Die Funktion $nxt : (Fma, \wp(trans(SC)), \wp(actions(SC))) \rightarrow Fma$ berechnet für ein Statechart SC und eine Übergangsmenge T die sich ergebende Zustandskonfiguration. Sei $S := entered_K(g, T) \cup active_K(g, T)$ die Menge der aktiven Zustände und $\overline{S} := states(SC) \backslash (entered_K(T) \cup active_K(T))$ die Menge der inaktiven Zustände des nächsten Schrittes, dann ist

$$nxt((g, T, \emptyset)) := \begin{cases} term(SC) \cap entered(g, T) \neq \emptyset, & \bigwedge_{s \in S} s \wedge \bigwedge_{\overline{s} \in \overline{S}} \neg\ \overline{s} \wedge \mathbf{last} \\ \text{sonst}, & \bigwedge_{s \in S} s \wedge \bigwedge_{\overline{s} \in \overline{S}} \neg\ \overline{s} \wedge SC. \end{cases}$$

und setzt damit die aktiven und inaktiven Zustände des nächsten Schrittes. Die Statechart-Formel SC bleibt unverändert, da sie die statische Zustandsübergangsrelation des Statecharts beschreibt. Wenn das Statechart einen Terminierungszustand $s \in term(SC)$ erreicht, beenden wir die Ausführung und kennzeichnen mit dem Prädikat **last**, dass der letzte Zustand des endlichen Intervalls erreicht wurde.

Beispiel 6.8 *Neue Zustandskonfiguration*
Für das Beispiel 6.7 ergibt sich für den Mikro-Schritt der 2. Prämisse mit $g_2 = \dot{\mathrm{Light}} \wedge \dot{\mathrm{Off}} \wedge \dot{\mathrm{press}} \wedge \dot{\mathrm{Timer}} \wedge \dot{\mathrm{Open}} \wedge \neg \dot{\mathrm{set}}$, $\mathrm{entered}_K(g_2, \{t_1\}) = \{\dot{\mathrm{On}}\}$ *und* $\mathrm{active}_K(g_2, \{t_1\}) = \{\dot{\mathrm{System}}, \dot{\mathrm{Light}}, \dot{\mathrm{Timer}}, \dot{\mathrm{Open}}\}$ *aus Beispiel 6.6:*

$$\begin{aligned} & S = \{\dot{\mathrm{System}}, \dot{\mathrm{Light}}, \dot{\mathrm{Timer}}, \dot{\mathrm{Open}}, \dot{\mathrm{On}}\}, \\ & \overline{S} = \{\dot{\mathrm{Close}}, \dot{\mathrm{Off}}\} \textit{ und} \\ & \mathrm{nxt}(\dot{\mathrm{Off}} \wedge \dot{press} \wedge \dot{\mathrm{Open}} \wedge \neg \dot{set}, \{t_1\}) = \\ & \quad \dot{\mathrm{System}} \wedge \dot{\mathrm{Timer}} \wedge \dot{\mathrm{On}} \wedge \dot{\mathrm{Light}} \wedge \dot{\mathrm{Open}} \wedge \neg\ \dot{\mathrm{Off}} \wedge \neg\ \dot{\mathrm{Close}} \wedge \mathrm{SYS}. \end{aligned}$$

6.4 Korrektheit der Statechart-Kalkülregeln

In diesem Abschnitt zeigen wir die Korrektheit der Statechart-Kalkülregeln.

Definition 6.1 *Korrektheit einer Kalkülregel*
Eine Regel

$$\frac{p_1 \quad \dots \quad p_n}{c}$$

ist genau dann korrekt, wenn aus

$$\models \forall_{Cl}\ p_i \textit{ für } i = 1 \ldots n \textit{ folgt: } \models \forall_{Cl}\ c.$$

Damit besagt die Korrektheit, wenn wir alle Prämissen nachgewiesen haben, dann haben wir die Konklusio gezeigt. Für die Regeln *sc initialize* und *sc wellformed* können wir auch die Invertierbarkeit zeigen.

Definition 6.2 *Invertierbarkeit einer Kalkülregel*
Eine Regel

$$\frac{p_1 \quad \dots \quad p_n}{c}$$

ist genau dann invertierbar, wenn aus

$$\models \forall_{Cl}\ c \textit{ folgt: } \models \forall_{Cl}\ p_i \textit{ für } i = 1 \dots n.$$

Invertierbar bedeutet, dass wir bei der Anwendung der Regel keine Informationen verlieren, also aus einer gültigen Konklusio gültige Prämissen erhalten.

6.4.1 Die *sc unwind*-Regel

Der Korrektheitsbeweis für die *sc unwind*-Regel folgt dem Prinzip der schrittweisen Ausführung. Wir zeigen, dass aus einer gegebenen Konfiguration die Nachfolgekonfigurationen entsprechend der operationellen Semantik aus Abschnitt 4.2.2 berechnet werden.

Da im Kalkül Variablenwerte und die Zustandskonfiguration symbolisch beschrieben werden, ergeben sich Unterschiede in der Berechnung der möglichen Statechart-Schritte. Während die Schrittberechnung *Steps* in der Semantikdefinition auf einer Belegung σ basiert, berechnen sich im Kalkül die möglichen Schritte bzw. Übergangsmengen nach Steps_K aus der aktuellen Sequenz $\Gamma \vdash \Delta$, die symbolisch die Konfiguration des Statecharts beschreibt. Neben den Übergangsmengen eines Schrittes liefert Steps_K die entsprechende Aktivierungsbedingung (die bei der Berechnung von *Steps* direkt ausgewertet werden). Nur wenn die Bedingung erfüllt, ist der entsprechende Schritt möglich.

Ein weiterer Unterschied zwischen der Semantikdefinition und dem Kalkül ergibt sich durch die Einbettung in den ITL-Rahmen. Anstatt, wie in der Semantikdefinition, die Nachfolgebelegung direkt zu beschreiben, definieren die Kalkülregeln die Belegung der 1-fach bzw. 2-fach gestrichenen Variablen und stellt mit $\sigma(x'') = \sigma'(\dot{x})$ die Relation zur Nachfolgebelegung σ' her. Die Ausführung eines Statechart-Schrittes beschreiben wir im Kalkül durch sequentielle DL-Programme. Die Programme „implementieren" die Schrittausführung.

Aus dieser Betrachtung ergibt sich die Beweisstruktur, mit der wir im Folgenden die Korrektheit der *sc unwind*-Regel zeigen. Zunächst zeigen wir, dass es für den Korrektheitsbeweis genügt, nur den ersten Übergang zur Nachfolgebelegung zu betrachten. Dann zeigen wir, dass die Berechnung der möglichen Schritte im Kalkül der Berechnung in der Semantikdefinition entspricht und schließlich, dass die sequentiellen Programme, die im Kalkül einen Schritt ausführen, einem Statechart-Schritt der Semantik entsprechen. Wie in der Semantikdefinition unterscheiden wir auch im Beweis zwischen Mikro- und Makro-Schritten.

Theorem 6.1 *Korrektheit der* sc unwind-*Regel*
Die sc unwind-*Regel*

$$\frac{\bigvee_{i=1}^{n}\text{cond}(stp_i) \wedge \text{step}(stp_i) \wedge \circ\ \text{nxt}(stp_i), \Gamma \vdash \Delta}{SC, \Gamma \vdash \Delta}\ \text{sc unwind}$$

mit $stp_i \in \text{Steps}_K(\Gamma \wedge \neg\Delta)$ *ist korrekt, d. h. wenn*

$$\models \forall_{Cl} \bigvee_{stp_i \in Steps_K(\Gamma \wedge \neg\Delta)} (cond(stp_i) \wedge step(stp_i) \wedge \circ nxt(stp_i)) \wedge \Gamma \rightarrow \Delta,$$

dann

$$\models \forall_{Cl}\ SC \wedge \Gamma \rightarrow \Delta.$$

Um das Theorem 6.1 zu beweisen, zeigen wir die Behauptung: wenn

$$\mathcal{I} \models \bigvee_{stp_i \in Steps_K(\Gamma \wedge \neg\Delta)} (cond(stp_i) \wedge step(stp_i) \wedge \circ nxt(stp_i)) \wedge \Gamma \rightarrow \Delta,$$

dann

$$\mathcal{I} \models SC \wedge \Gamma \rightarrow \Delta.$$

Entweder gilt $\mathcal{I} \not\models \Gamma \wedge \neg\ \Delta$, dann ist der Beweis trivial. Sonst zeigen wir, wenn

$$\mathcal{I} \models SC :\Leftrightarrow \mathcal{I}\ |_{variables(SC)} \in traces(SC), \tag{5}$$

dann

$$\mathcal{I} \models \bigvee_{stp_i \in Steps_K(\Gamma \wedge \neg\Delta)} (cond(stp_i) \wedge step(stp_i) \wedge \circ nxt(stp_i)) \tag{6}$$

Die Menge seiner Intervalle beschreibt ein Statechart durch

$$\begin{aligned} traces(SC) \ := \ \{(\sigma_0, \ldots, \sigma_n) \mid n \in \mathbb{N}_\infty, \sigma_i \in \sum(variables(SC)), \downarrow(\sigma_0) \\ \sigma_i\ \rho_{step}\ \sigma_{i+1} \text{ und } \sigma_n \notin dom(\rho_{step_{SC,\mathcal{A}}})\}. \end{aligned} \tag{7}$$

Bemerkung Die Semantik der Statecharts ist nur über den Statechart-Variablen definiert und schränkt die Belegung anderer Variablen nicht ein. Die Disjunktion in (6) bezieht sich ebenfalls nur auf Statechart-Variablen. Um im Folgenden nicht immer explizit die Einschränkung der Belegungen auf *variables*(SC) angeben zu müssen, meinen wir in den nachfolgenden Beweisen, dass nur die Statechart-Variablen geändert werden und schränken damit implizit die Belegungen auf *variables*(SC) ein.

Den Korrektheitsbeweis führen wir analog zur symbolischen Ausführung eines Statecharts. Da ρ_{step} nur konsistente Belegungen berechnet, ist σ_1 wieder konsistent, es gilt also $\downarrow(\sigma_1)$. Aus den Gleichungen (5) und (7) folgt damit, dass

$$\begin{aligned} (\sigma_0, \sigma_1, \ldots) \models SC \quad \Leftrightarrow \quad & \downarrow(\sigma_0),\ \sigma_0\ \rho_{step}\ \sigma_1 \text{ und} \\ & (\sigma_1, \sigma_2, \ldots) \in traces(SC). \end{aligned}$$

$(\sigma_1, \sigma_2, \ldots)$ ist in $traces(SC)$ und damit $(\sigma_1, \sigma_2, \ldots) \models SC$. Die *sc unwind*-Regel wickelt die jeweils ersten Schritte ab. Es genügt also, nur die ersten möglichen Schritte (die durch $\sigma_0\ \rho_{step}\ \sigma_1$ bzw. die *sc unwind*-Regel beschrieben werden) für den Korrektheitsbeweis zu betrachten.

Schrittberechnung

Wir zeigen nun, dass für ein Intervall $(\sigma_0, \sigma_1, \ldots)$ mit einer konsistenten Belegung $\downarrow(\sigma_0)$ die Schrittberechnung im Kalkül einen Schritt mit derselben Übergangsmenge und erfüllter Aktivierungsbedingung berechnet, wie die Schrittberechnung in der Semantikdefinition. Die Tabelle 6.1 beschreibt informell die notwendigen Funktionen für die Schrittberechnung

Semantik	Kalkül
$steps(\sigma, s)$	$steps_K(\Gamma \wedge \neg\Delta, s, g)$
berechnen maximale konfliktfreie Mengen von Übergängen, beginnend in s.	
$action_set(st, \sigma)$	$action_set_K((g, st))$
berechnen zu einem Schritt st die möglichen Aktionen der Übergänge und statischen Reaktionen.	
$action_set(Steps(\sigma), \sigma)$	$Steps_K(\Gamma \wedge \neg\Delta)$
maximale Mengen der auszuführenden Übergänge und Aktionen eines Statecharts.	

Tabelle 6.1: Funktionen: Semantik vs. Kalkül

und stellt die Funktionen aus der Semantikdefinition den Funktionen aus dem Kalkül gegenüber. Die formale Definitionen der Funktionen findet sich in Abschnitt 4.2.2 (Semantik) bzw. in Abschnitt 6.3 (Kalkül). Folgende, weitere Funktionen werden sowohl in der Semantikdefinition als auch in den Berechnungen der Kalkülregel verwendet und wurden in Abschnitt 4.2.1 (Syntax) definiert.

- $scope(t)$: ergibt die Region eines Übergangs, welche die Priorität des Übergangs bestimmt.
- $childs(s)$, $childs^*(s)$: berechnen (reflexiv transitiv) die Unterzustände von s.
- $source(t)$, $target(t)$, $guard(t)$, $action(t)$: berechnen den Ausgangs-, Endzustand, die Aktivierungsbedingung bzw. die Aktion eines Übergangs.
- $en(t)$: berechnet, ob eine Übergang aktiv ist ($en(t) := source(t) \wedge guard(t)$).

Lemma 6.1 *Schrittberechnung I*
Für ein Intervall $(\sigma_0, \sigma_1, \ldots)$ *mit konsistenter Belegung* $\downarrow(\sigma_0)$ *und* $(\sigma_0, \sigma_1, \ldots) \models \Gamma \wedge \neg\Delta$ *gilt:*

1. $Steps(\sigma_0) = \emptyset$ *und* $\mathcal{ACT} = \text{action_set}(\emptyset, \sigma_0)$ *genau dann, wenn es ein* $(g, \emptyset, \mathcal{ACT}) \in \text{Steps}_K(\Gamma \wedge \neg\Delta)$ *gibt und* $\sigma_0 \models g$.

2. *Es gibt ein* $st \in Steps(\sigma_0)$ *und* $\mathcal{ACT} = \text{action_set}(st, \sigma_0)$ *genau dann, wenn es ein* $(g, st, \mathcal{ACT}) \in \text{Steps}_K(\Gamma \wedge \neg\Delta)$ *gibt,* $st \neq \emptyset$ *und* $\sigma_0 \models g$.

In der Semantikdefinition beziehen wir uns immer auf eine konkrete Belegung. Dagegen betrachten wir im Kalkül all diejenigen Belegungen, die durch $\Gamma\wedge\neg\Delta$ und den entsprechenden Statechart SC beschrieben werden. Deshalb liefert die Schrittberechnung $Steps_K(\Gamma \wedge \neg\Delta)$ mehr potentiell mögliche Schritte mit entsprechender Aktivierungsbedingung $(g, st, \mathcal{ACT})$ als $Steps(\sigma_0)$. Der Korrektheitsbeweis geht nun über eine konkrete Belegung σ_0 und es ist zu zeigen, dass genau die Aktivierungsbedingung g in σ_0 erfüllt ist, deren entsprechender Schritt st in $Steps(\sigma_0)$ enthalten ist. Ist in der Belegung σ_0 kein Schritt möglich, berechnet $Steps_K(\Gamma \wedge \neg\Delta)$ eine Tripel $(g, \emptyset, \mathcal{ACT})$ mit erfüllter Aktivierungsbedingung g (beachte, dass für alle $st \in Steps(\sigma_0)$ gilt $st \neq \emptyset$). Für den Beweis von Lemma 6.1 verwenden wir folgende Lemmata.

Lemma 6.2 *Schrittberechnung II*
Für ein Intervall $(\sigma_0, \sigma_1, \ldots)$ *mit konsistenter Belegung* $\downarrow(\sigma_0)$ *und* $(\sigma_0, \sigma_1, \ldots) \models \Gamma \wedge \neg\Delta$ *und* $\sigma_0 \models g$ *gilt:*

1. $\text{steps}(\sigma_0, s) = \emptyset$ *genau dann, wenn* $(g', \emptyset) \in \text{steps}_K(\Gamma \wedge \neg\Delta, s, g)$ *und* $\sigma_0 \models g'$.

2. $st \in \text{steps}(\sigma_0, s)$ *genau dann, wenn* $(g', st) \in \text{steps}_K(\Gamma\wedge\neg\Delta, s, g)$, $st \neq \emptyset$ *und* $\sigma_0 \models g'$.

Wenn ein Schritt st berechnet wird, ist $st \neq \emptyset$ und die Menge $steps(\sigma_0, s) = \{st \mid (g', st) \in steps_K(\Gamma \wedge \neg\Delta, s, g),\ \sigma_0 \models g'\}$. Ansonsten ist $(g', \emptyset, \mathcal{ACT})$ das einzige Tripel in $steps_K(\Gamma \wedge \neg\Delta, s, g)$ mit erfüllter Aktivierungsbedingung. Insbesondere bedeutete dies, dass es immer ein $(g', st, \mathcal{ACT}) \in steps_K(\Gamma \wedge \neg\Delta, s, g)$ mit erfüllter Aktivierungsbedingung g' gibt.

Lemma 6.3 *Aktionenmengen*
Für ein Intervall $(\sigma_0, \sigma_1, \ldots)$ *mit konsistenter Belegung* $\downarrow(\sigma_0)$, $(\sigma_0, \sigma_1, \ldots) \models \Gamma\wedge\neg\Delta$, *einer Aktivierungsbedingung* g *und einer Menge von Übergängen* st *gilt:*

$$\mathcal{ACT} = \text{action_set}(st, \sigma_0) \text{ und } \sigma_0 \models g \text{ genau dann, wenn } (g', \mathcal{ACT}) \in \text{action_set}_K((g, st)) \text{ und } \sigma \models g'.$$

$action_set(st, \sigma_0)$ erweitert einen Schritt st um die Aktionen der statischen Reaktionen. Lemma 6.3 zeigt nun, dass für einen Schritt st und einen entsprechenden Schritt (g, st) mit erfüllter Aktivierungsbedingung, in der Semantik und im Kalkül dieselben statischen Reaktionen berücksichtigt und deren Aktionen ausgeführt werden.

Beweis 6.1 *Schrittberechnung I (Lemma 6.1)*
Sei $(\sigma_0, \sigma_1, \ldots)$ *ein Intervall mit* $\downarrow(\sigma_0)$ *und* $(\sigma_0, \sigma_1, \ldots) \models \Gamma \wedge \neg\Delta$*. Mit Lemma 6.2, Lemma 6.3 und*

$$\begin{aligned}&\mathrm{Steps}_K(\Gamma \wedge \neg\Delta) := \\ &\quad \{(g, st, \mathcal{ACT}) \mid (g, \mathcal{ACT}) \in \mathrm{action_set}_K((g', st)), (g', st) \in \mathrm{steps}_K(\Gamma \wedge \neg\Delta, \mathrm{root}, \mathrm{true})\}\end{aligned}$$

zeigen wir die Behauptung.

1. *Wenn* $\mathrm{Steps}(\sigma_0) = \emptyset$*, folgt mit* $\mathrm{Steps}(\sigma_0) := \mathrm{steps}(\sigma_0, \mathrm{root})$*,* $\sigma_0 \models \mathrm{true}$ *und Lemma 6.2, dass* $(g', \emptyset) \in \mathrm{steps}_K(\Gamma \wedge \neg\Delta, \mathrm{root}, \mathrm{true})$ *und* $\sigma_0 \models g'$*. Mit Lemma 6.3,* $\sigma_0 \models g'$ *und* $\mathcal{ACT} = \mathrm{action_set}(\emptyset, \sigma_0)$ *folgt, dass* $(g, \mathcal{ACT}) \in \mathrm{action_set}_K((g', \emptyset))$ *und* $\sigma_0 \models g$*. Damit ist* $(g, \emptyset, \mathcal{ACT}) \in \mathrm{Steps}_K(\Gamma \wedge \neg\Delta)$ *und* $\sigma_0 \models g$*.*

   Andererseits, wenn $(g, \emptyset, \mathcal{ACT}) \in \mathrm{Steps}_K(\Gamma \wedge \neg\Delta)$ *und* $\sigma_0 \models g$*, dann ist* $(g, \mathcal{ACT}) \in \mathrm{action_set}_K((g', \emptyset))$ *und mit Lemma 6.3 folgt* $\mathcal{ACT} = \mathrm{action_set}(\emptyset, \sigma_0)$ *und* $\sigma_0 \models g'$*. Desweiteren ist* $(g', \emptyset) \in \mathrm{steps}_K(\Gamma \wedge \neg\Delta, \mathrm{root}, \mathrm{true})$*,* $\mathrm{Steps}(\sigma_0) = \mathrm{steps}(\sigma_0, \mathrm{root})$ *und mit Lemma 6.2 folgt* $\mathrm{Steps}(\sigma_0) = \emptyset$*.*

2. *Wenn es ein* $st \in \mathrm{Steps}(\sigma_0)$ *gibt, folgt mit* $\mathrm{Steps}(\sigma_0) := \mathrm{steps}(\sigma_0, \mathrm{root})$*,* $\sigma_0 \models \mathrm{true}$ *und Lemma 6.2, dass* $(g', st) \in \mathrm{steps}_K(\Gamma \wedge \neg\Delta, \mathrm{root}, \mathrm{true})$*,* $st \neq \emptyset$ *und* $\sigma_0 \models g'$*. Desweiteren folgt mit* $\mathcal{ACT} = \mathrm{action_set}(st, \sigma_0)$*,* $\sigma_0 \models g'$ *und Lemma 6.3,* $(g, \mathcal{ACT}) \in \mathrm{action_set}_K((g', st))$ *und* $\sigma_0 \models g$*. Damit ist* $(g, st, \mathcal{ACT}) \in \mathrm{Steps}_K(\Gamma \wedge \neg\Delta)$*.*

   Für die Rückrichtung zeigen wir, wenn $(g, st, \mathcal{ACT}) \in \mathrm{Steps}_K(\Gamma \wedge \neg\Delta)$*,* $st \neq \emptyset$ *und* $\sigma_0 \models g$*, dann ist mit* $(g, \mathcal{ACT}) \in \mathrm{action_set}_K((g', st))$ *und Lemma 6.3 die Menge der auszuführenden Aktionen* $\mathcal{ACT} = \mathrm{action_set}(st, \sigma_0)$ *und* $\sigma_0 \models g'$*. Mit Lemma 6.2,* $(g', st) \in \mathrm{steps}_K(\Gamma \wedge \neg\Delta, \mathrm{root})$ *und* $\mathrm{Steps}(\sigma_0) = \mathrm{steps}(\sigma_0, \mathrm{root}, \mathrm{true})$ *folgt, dass* $st \in \mathrm{Steps}(\sigma_0)$*.*

□

Beweis 6.2 *Schrittberechnung II (Lemma 6.2)*
Sei $(\sigma_0, \sigma_1, \ldots)$ *ein Intervall mit* $\downarrow(\sigma_0)$ *und* $(\sigma_0, \sigma_1, \ldots) \models \Gamma \wedge \neg\Delta$*. Der Beweis folgt mit Induktion über die Struktur des Statecharts SC.*

BASIC, TERM $\mathrm{steps}(\sigma_0, s) := \emptyset$
Voraussetzung: $\sigma_0 \models g$

$$\mathrm{steps}_K(\Gamma \wedge \neg\Delta, s, g) := \{(g, \emptyset)\}$$

$(g, \emptyset)$ *ist das einzige Paar in* $\mathrm{steps}_K(\Gamma \wedge \neg\Delta, s, g)$ *und* $\sigma_0 \models g$ *gilt nach Voraussetzung. D. h.* $\mathrm{steps}_K(\Gamma \wedge \neg\Delta, s, g)$ *berechnet einen Schritt* (g, st) *mit* $\sigma_0 \models g$ *und leerer Menge von Übergängen* st *genau dann, wenn* $\mathrm{steps}(\sigma_0, s) = \emptyset$*.*

AND *für* $s_i \in \mathrm{childs}(s)$ *gilt* $\mathrm{steps}(\sigma_0, s) := \{st_1 \cup \ldots \cup st_n \mid st_i \in \mathrm{steps}(\sigma_0, s_i)\}$
Voraussetzung: $\sigma_0 \models g$

$$\mathrm{steps}_K(\Gamma \wedge \neg\Delta, s, g) := \{(g_1 \wedge \ldots \wedge g_n, st_1 \cup \ldots \cup st_n) \mid (g_i, st_i) \in \mathrm{steps}_K(\Gamma \wedge \neg\Delta, s_i, s_i \wedge g)\}$$

Nach Induktionsvoraussetzung gilt für $s_i \in \text{childs}(s)$*:* $st_i \in \text{steps}(\sigma_0, s_i)$ *genau dann, wenn* $(g_i, st_i) \in \text{steps}_K(\Gamma \wedge \neg\Delta, s_i, s_i \wedge g)$, $st_i \neq \emptyset$ *und* $\sigma_0 \models g_i$ *bzw.* $\text{steps}(\sigma_0, s_i) = \emptyset$ *genau dann, wenn* $(g_i, \emptyset) \in \text{steps}_K(\Gamma \wedge \neg\Delta, s_i, s_i \wedge g)$ *und* $\sigma_0 \models g_i$.

Ist $\text{steps}(\sigma_0, s_i) \neq \emptyset$ *gilt für* $(g_i, st_i) \in \text{steps}_K(\Gamma \wedge \neg\Delta, s_i, s_i \wedge g)$ *mit* $st_i \notin \text{steps}(\sigma_0, s_i)$ $\sigma_0 \not\models g_i$ *und für* $\text{steps}(\sigma_0, s_i) = \emptyset$ *gilt:* $(g_i, st_i) \in \text{steps}_K(\Gamma \wedge \neg\Delta, s_i, s_i \wedge g)$ *und* $\sigma_0 \not\models g_i$ *genau dann, wenn* $st_i \neq \emptyset$.

Deshalb gilt genau dann $(g', st) \in \text{steps}_K(\Gamma \wedge \neg\Delta, s, g)$, $st \neq \emptyset$ *und* $\sigma_0 \models g'$ ($st := \bigcup_{i=1\ldots n} st_i$, $g' := \bigwedge_{i=1\ldots n} g_i$)*, wenn* $st \in \text{steps}(\sigma_0, s)$ *bzw. genau dann* $(g', \emptyset) \in \text{steps}_K(\Gamma \wedge \neg\Delta, s, g)$ *und* $\sigma_0 \models g'$*, wenn* $\text{steps}(\sigma_0, s) = \emptyset$*. Nur für diese Paare gilt:* $\sigma_0 \models \bigwedge_{i=1\ldots n} g_i$.

OR *Sei im folgenden* s' *der aktive Nachfolger von* s *in* σ_0,[3] *d. h.* $\sigma_0 \models s'$ *und*

$T = \{\{t_1\}, \ldots, \{t_k\} \mid \text{scope}(t_i) = s \textit{ und } \sigma_0 \models en(t_i)\}$
die Menge der aktivierten Übergänge in σ_0.

$S' = \{\tilde{s} \mid \tilde{s} \in \text{childs}(s) \textit{ und nicht } \Gamma \wedge \neg\, \Delta \rightarrow \neg\, \tilde{s}\}$
die Menge potentiell aktiver, direkter Unterzustände von s*. Da* $\sigma_0 \models \Gamma \wedge \neg\, \Delta$ *und* $\sigma_0 \models s'$*, gilt nicht* $\Gamma \wedge \neg\Delta \rightarrow \neg\, s'$ *und damit* $s' \in S'$*. Für alle* $\tilde{s} \in S'$, $\tilde{s} \neq s'$ *gilt* $\sigma_0 \not\models \tilde{s}$.

$S^* = \{\tilde{s} \mid \tilde{s} \in \text{childs}^*(s) \textit{ und nicht } \Gamma \wedge \neg\, \Delta \rightarrow \neg\, \tilde{s}\}$
die Menge potentiell aktiver, direkter und indirekter Unterzustände von s.

$T_{\tilde{s}} = \{t_{\tilde{s}_1}, \ldots, t_{\tilde{s}_k}\} = \{t \mid \text{scope}(t) = s, \text{source}(t) \in \text{childs}^*(\tilde{s}) \cap S^*\}$
die Menge der relevanten Übergänge t *mit* $\text{scope}(t) = s$*, die* $\tilde{s}$ *verlassen.*

$T_K = \{t_1, \ldots, t_l\} = \bigcup_{\tilde{s} \in S'} T_{\tilde{s}}$
die Menge der relevanten Übergänge t*, die in* s *möglich sind, d. h. deren Ausgangszustand nicht inaktiv ist und deren* $\text{scope}(t) = s$.

Für s'*, dem einzig aktiven Nachfolger von* s *in* σ_0 *gilt in der Semantikdefinition*

$$\text{steps}(\sigma_0, s) := \begin{cases} T = \emptyset, & \text{steps}(\sigma_0, s') \\ \textit{sonst}, & \text{steps}(\sigma_0, s) := \{\{t_1\}, \ldots, \{t_k\}\} \end{cases}$$

und im Kalkül

$$\begin{aligned} &\text{steps}_K(\Gamma \wedge \neg\Delta, s, g) := \\ &\quad \{(g_t, \{t\}) | t \in T_K \textit{ und } g_t := g \wedge \text{guard}(t) \wedge \text{source}(t)\} \cup && (8) \\ &\quad \bigcup_{\tilde{s} \in S'} \text{steps}_K(\Gamma \wedge \neg\Delta, \tilde{s}, \tilde{g}) && (9) \\ &\qquad \textit{für } \tilde{g} := g \wedge \tilde{s} \wedge \neg\text{guard}(t_{\tilde{s}_1}) \wedge \ldots \wedge \neg\text{guard}(t_{\tilde{s}_k}),\ t_{\tilde{s}_i} \in T_{\tilde{s}}. \end{aligned}$$

Mit diesen Definitionen beginnen wir den Beweis.

Voraussetzung: $\sigma_0 \models g$

[3] Da σ_0 konsistent ist, ist s' der *einzige* aktive Nachfolger von s.

1. *i) Sei* $T = \emptyset$.

   Dann sind alle Übergänge t mit $\text{scope}(t) = s$ *nicht aktiviert, d. h. es gilt* $\sigma_0 \not\models en(t)$ *bzw.* $\sigma_0 \not\models \text{source}(t)$ *für* $source(t) \neq s'$ *oder* $\sigma_0 \not\models \text{guard}(t)$ *für* $source(t) = s'$. *Für alle* $t_i \in T_{s'}$ *folgt* $\sigma_0 \not\models \text{guard}(t_i)$ *und für alle* $t_i \in T_{\tilde{s}}$, $\tilde{s} \neq s'$ *folgt* $\sigma_0 \not\models \text{source}(t_i)$. *Daraus folgt für alle* $t_i \in T_K$, $\sigma_0 \not\models en(t_i)$ *und für alle Übergangsbedingung in Gleichung (8):* $\sigma_0 \not\models g_t$.

   Gleichung (9) betrachtet nun alle Übergänge, die in Unterzuständen möglich sind. Für alle Unterzustände $\tilde{s} \neq s'$*, die nicht gleich dem einzig aktiven Nachfolger sind, ist die Bedingung* $\tilde{g} := g \wedge \tilde{s} \wedge \neg\text{guard}(t_{\tilde{s}_1}) \wedge \ldots \wedge \neg\text{guard}(t_{\tilde{s}_k})$, $t_{\tilde{s}_i} \in T_{\tilde{s}}$ *nicht erfüllt, da* $\sigma_0 \not\models \tilde{s}$*, und diese Unterzustände liefern keine weiteren Schritte mit aktivierter Übergangsbedingung.*

   Betrachten wir nun noch die Übergänge des einzig aktiven Nachfolgers s'. *Wegen* $\sigma_0 \not\models g_i$ *mit* $g_i = guard(t_i)$, $t_i \in T_{s'}$ *folgt*

   $$\sigma_0 \models \tilde{g} \textit{ mit } \tilde{g} := g \wedge s' \wedge \neg\text{guard}(t_1) \wedge \ldots \wedge \neg guard(t_k)$$

   und die Aktivierungsbedingung $\tilde{g}$ *von* $\text{steps}_K(\Gamma \wedge \neg\Delta, s', \tilde{g})$ *gilt in* σ_0. *Dann liefert nach Induktionsvoraussetzung* $\text{steps}_K(\Gamma \wedge \neg\Delta, s', \tilde{g})$ *die gleichen Schritte mit erfüllter Aktivierungsbedingung wie* $\text{steps}(\sigma_0, s')$. *Insbesondere ist* $(g', \emptyset) \in \text{steps}_K(\Gamma \wedge \neg\Delta, s', \tilde{g})$, *wenn* $\text{steps}(\sigma_0, s') = \emptyset$.

   ii) Sei $T \neq \emptyset$

   Daraus folgt $\sigma_0 \models en(t_i)$ *für alle* $\{t_i\} \in T$. *Mit* $\sigma_0 \models en(t_i)$ *gilt auch* $\sigma_0 \models \text{guard}(t_i)$ *und* $\sigma_0 \models \text{source}(t_i)$ *mit* $\sigma_0 \models s'$ *und* s'*, dem einzig aktiven Nachfolger von* s*, sind in* T *nur Übergänge enthalten, die* s' *verlassen, d. h.* $\text{source}(t_i) = s'$. *In Gleichung (8) sind deshalb nur diejenigen Bedingungen* g_{t_i} *mit* $\text{source}(t_i) = s'$ *erfüllt. Nach Voraussetzung* $\sigma_0 \models g$ *und Gleichung (8) sind genau diejenigen Aktivierungsbedingungen* $g_{t_i} := g \wedge \text{source}(t_i) \wedge \text{guard}(t_i)$ *erfüllt, für die* $\sigma_0 \models en(t_i)$ *und deshalb* $\{t_i\} \in \text{steps}(\sigma_0, s)$ *ist.*

   Es bleibt also zu zeigen, dass in Gleichung (9) keine weiteren Schritte mit aktivierter Übergangsbedingung hinzugefügt werden. Dies ist trivial. Entweder wir betrachten einen Unterzustand $\tilde{s} \neq s'$. *Mit* $\sigma_0 \not\models \tilde{s}$ *kann die Aktivierungsbedingung* $\tilde{g}$ *nicht erfüllt sein. Oder wir betrachten den aktiven Nachfolger* s' *von* s. *Da* $T \neq \emptyset$ *gilt für mindestens eine Bedingung* $\text{guard}(t_{s'_i}), t_{s'_i} \in T_{s'}$, $\sigma_0 \models \text{guard}(t_{s'_i})$ *und deshalb* $\sigma_0 \not\models \neg\ \text{guard}(t_{s'_i})$.

2. *i) Sei für alle* $t \in T_K$, $g_t := g \wedge \text{guard}(t) \wedge \text{source}(t)$*:* $\sigma_0 \not\models g_t$.

   Dann ist die Menge aus Gleichung (8) leer und für alle Übergänge t in der Region von s *(*$s = \text{scope}(t)$*)* $\sigma_0 \not\models en(t)$, *d. h. auch die Menge* T *der Übergänge aus der Region von* s *ist leer. Betrachten wir nun die Unterzustände von* s. *Nach Voraussetzung ist* σ_0 *konsistent und es gibt nur ein* $s' \in S'$ *mit* $\sigma_0 \models s'$. *Dann können in Gleichung (9) nur Aktivierungsbedingungen* $\tilde{g}$ *mit* $\tilde{s} = s'$*, dem einzig aktiven Nachfolgers* s' *gelten*

(für $\tilde{s} \in S'$, $\tilde{s} \neq s'$*:* $\sigma_0 \not\models \tilde{s}$*). Nach Induktionsvoraussetzung gilt dann für* $(g'', st) \in \text{steps}_K(\Gamma \wedge \neg\Delta, s', g')$ *und* $\sigma_0 \models g$*, wenn* $st \neq \emptyset$*, dann* $st \in \text{steps}(\sigma_0, s')$ *und wenn* $st = \emptyset$*, dann* $\text{steps}(\sigma_0, s') = \emptyset$*.*

ii) Es gibt ein $t \in T_K$*,* $g_t := g \wedge \text{guard}(t) \wedge \text{source}(t)$*:* $\sigma_0 \models g_t$*.*

Da σ_0 *konsistent, gilt für alle* $t', t'' \in T_K$ *mit* $\sigma_0 \models g_{t'}$ *und* $\sigma_0 \models g_{t''}$*, dass* $\text{source}(t') = \text{source}(t'')$*, d. h es gibt einen aktiven Nachfolger von* s*, den wir wieder* s' *nennen, und für alle* t *mit* $\sigma_0 \models g_t$*, ist* $t \in T_{s'}$*. Wenn* $\sigma_0 \models g_t$*, dann gilt auch* $\sigma_0 \models en(t)$*, d. h. für alle* $(g_t, \{t\})$ *aus Gleichung (8) ist* $\{t\} \in T$ *genau dann, wenn* g_t *in* σ_0 *erfüllt ist.*

Es bleibt zu zeigen, dass Gleichung (9) keine weiteren Paare (g', st) *mit* $\sigma_0 \models s'$ *liefert. Dies ist trivial, da für alle Aktivierungsbedingungen* $\tilde{g}$ *nicht erfüllt sind. Für* $\tilde{s} \in S'$, $\tilde{s} \neq s'$ *gilt* $\sigma_0 \not\models \tilde{s}$ *und für* $\tilde{s} = s'$ *gibt es ein* $t_{s'} \in T_{s'}$ *mit* $\sigma_0 \models \text{guard}(t_{s'})$ *und deshalb* $\sigma_0 \not\models \neg\text{guard}(t_{s'_1}) \wedge \ldots \wedge \neg\text{guard}(t_{s'_k})$*.*

Damit folgt, dass steps *dieselben Übergangsmengen wie* steps_K *liefert, deren Aktivierungsbedingung in* σ *gilt. Liefert* steps *die leere Menge, dann liefert* steps_K *ein Paar* (g, st) *mit erfüllter Aktivierungsbedingung und leerer Übergangsmenge* $st = \emptyset$*.* □

Aktive Zustände Für den Beweis der Äquivalenz der berechneten Aktionenmengen benötigen wir als Hilfsaussage, dass sich die Menge der aktiven Zuständen aus der Semantikdefinition und dem Kalkül entspricht.

Lemma 6.4 *Aktive Zustände*
Für eine konsistente Belegung $\downarrow(\sigma)$ *und einen Statechart-Schritt* (g, st) *mit Aktivierungsbedingung* g*,* $\sigma \models g$ *und der Menge von Übergängen* st *gilt:*

$$\begin{array}{ll} i) & \text{active}(st) = \text{active}_K(g, st) \text{ und} \\ ii) & \text{entered}(st) = \text{entered}_K(g, st) \end{array}$$

Beweis 6.3 *Aktive Zustände (Lemma 6.4)*
Sei σ *eine konsistente Belegung* $\downarrow(\sigma)$*, und* g *eine Aktivierungsbedingung mit entsprecheder Übergangsmenge* st*,* $\sigma_0 \models g$*.*

i). Auf Semantikebene berechnet sich die Menge der aktiven Zustände mit

$$\text{active}(st, \sigma) := \{s \mid \sigma(s)\} \backslash \text{exited}(st, \sigma)$$

und

$$\text{exited}(st, \sigma) := \{s \mid \exists\, t \in st \text{ mit } \text{source}(t) < s\} \cap \{s \mid \sigma(s)\}$$

auf Kalkülebene durch

$$\text{active}_K(g, st) := \{s \mid g \to s \text{ oder } s = \text{root}(SC)\} \backslash \text{changed}_K(st)$$

mit

$$\text{changed}_K(st) := \{s \mid \exists\, t \in st \text{ mit } \text{source}(t) < s\}.$$

1. *wenn* $s \in \text{active}(st, \sigma)$*, dann auch* $s \in \text{active}_K(g, st)$
   Der Beweis folgt mit Induktion über die Tiefe von s im Statechart.
   $\underline{\text{depth}(s) = 0}$*:*

   Dann ist $s = \text{root}(SC)$*. Nach Definition ist* $\text{root}(SC)$ *auch in* $\text{active}_K(g, st)$*.*
   $\underline{\text{depth}(s) = n}$*:*

   Da σ *konsistent ist, ist* $\text{father}(s)$ *ebenfalls in* $\text{active}(st, \sigma)$*,* $\text{depth}(\text{father}(s)) = n - 1$ *und nach Induktionsvoraussetzung ist* $\text{father}(s) \in \text{active}_K(g, s)$*.*

   $\text{mode}(\text{father}(s)) = \text{BASIC}, \text{TERM}$*:* ↯

   $\text{mode}(\text{father}(s)) = \text{AND}$*:*
   Für alle Unterzustände $s_i \in \text{childs}(\text{father}(s))$ *wird in* $\text{steps}_K(\Gamma \wedge \neg\Delta, \text{father}(s), g') :=$

   $$\{(g_1 \wedge \ldots \wedge g_n, st_1 \cup \ldots \cup st_n) \mid (g_i, st_i) \in \text{steps}_K(\Gamma \wedge \neg\Delta, s_i, s_i \wedge g')\}$$

   der Zustand s_i *zur bisherigen Aktivierungsbedingung* g' *hinzugefügt. Damit ist* $g' \wedge s$ *ein Konjunktionsglied in* g*,* $g \rightarrow s$ *und* $s \in \text{active}_K(g, st)$*.*

   $\text{mode}(\text{father}(s)) = \text{OR}$*:*
   Da $s \notin \text{exited}(st, \sigma)$ *folgt für alle Übergänge mit* $\text{scope}(t) = \text{father}(s)$ *und* $\text{source}(t) = s$*,* $\sigma \not\models \text{guard}(t)$*. Deshalb werden die Übergänge des einzig aktiven Unterzustands* s *in* σ *betrachtet. Dazu fügt*

   $$\begin{aligned} &\text{steps}_K(\Gamma \wedge \neg\Delta, \text{father}(s), g') := \\ &\quad \{(g_t, \{t\}) | t \in T_K \textit{ und } g_t := g' \wedge \text{guard}(t) \wedge \text{source}(t)\} \cup \\ &\quad \bigcup_{s \in S'} \text{steps}_K(\Gamma \wedge \neg\Delta, s, \tilde{g}) \\ &\qquad \textit{für } \tilde{g} := g' \wedge s \wedge \neg\text{guard}(t_{s_1}) \wedge \ldots \wedge \neg\text{guard}(t_{s_k}),\ t_{s_i} \in T_s. \end{aligned} \tag{10}$$

   den Zustand s *in Gleichung (10) zur vorläufigen Aktivierungsbedingung* g'*,* $g' \wedge s$ *ist ein Konjunktionsglied von* g*,* $g \rightarrow s$ *und* $s \in \text{active}_K(g, st)$*.*

2. *wenn* $s \in \text{active}_K(g, st)$*, dann auch* $s \in \text{active}(st, \sigma)$
   Es gilt $\sigma \models g$ *und für* $s \in \text{active}_K(g, st)$ *auch* $g \rightarrow s$ *und deshalb* $\sigma \models s$*. Da* $s \notin \text{changed}_K(st)$ *ist* s *erst recht nicht in* $\text{exited}(st, \sigma)$*.*

ii). Nach Definition ist

$$\begin{aligned} \text{entered}(st) &:= \text{decompl}(\text{active}(st) \cup \textstyle\bigcup_{t \in st}\{\text{target}(t)\}) \backslash \text{active}(st) \textit{ und} \\ \text{entered}_K(g, T) &:= \text{decompl}(\text{active}_K(g, T) \cup \textstyle\bigcup_{t \in T}\{\text{target}(t)\}) \backslash \text{active}_K(g, T) \end{aligned}$$

und mit i) folgt die Behauptung. □

Aktionenmengen Mit obigen Hilfslemma können wir zeigen, dass sowohl in der Semantik als auch im Kalkül, für einen Schritt diegleichen Aktionenmengen berechnet werden. Der Beweis über die Äquivalenz der berechneten Aktionenmengen beruht auf den Funktionen aus Tabelle 6.2 und den folgenden Funktionen (formale Definition: siehe Abschnitt 4.2.2 (Semantik), Abschnitt 4.1.1 (Syntax) bzw. Abschnitt6.3 (Kalkül)).

- $action(t)$, $action(sr)$, $actions(st)$: berechnen die Aktionen eines Übergangs, einer statischen Reaktion bzw. einer Menge von Übergängen.
- $sreact(s)$: ergeben die statischen Reaktionen eines Zustands.
- $en(sr)$: die Aktivierungsbedingung einer statischen Reaktion (der Zustand, in der sr definiert ist, muss aktiv und die Aktivierungsbedingung $guard(sr)$ erfüllt sein).

Semantik	Kalkül
$exited(st)$	$changed_K(st)$
berechnen Zustände die verlassen (geändert) werden.	
$active(st)$	$active_K(g, st)$
berechnen aktive Zustände, die weder verlassen noch betreten werden.	
$entered(st)$	$entered_K(g, st)$
berechnen Zustände, die verlassen werden.	

Tabelle 6.2: Funktionen: Semantik vs. Kalkül

Beweis 6.4 *Aktionenmengen (Lemma 6.3)*
Sei $(\sigma_0, \sigma_1, \ldots)$ *ein Intervall mit* $\downarrow(\sigma_0)$ *und* $(\sigma_0, \sigma_1, \ldots) \models \Gamma \wedge \neg\Delta$. *Auf Semantikebene berechnet sich die Menge der Aktionen durch*

$$\text{action_set}(st, \sigma_0) := \text{actions}(st) \cup \{\text{action}(sr) \mid \sigma_0 \models \text{en}(sr)\},$$

mit

$$\sigma_0 \models \text{en}(sr) :\Leftrightarrow \exists\ s.s \in \text{active}(st) \textit{ und } sr \in \text{sreact}(s) \textit{ und } \sigma \models \text{guard}(sr)$$

und auf Kalkülebene durch

$$\text{action_set}_K((g, st)) := \{(g, \text{actions}(st))\} \uplus \biguplus_{sr_i \in sreact(s), s \in active_K(g,st)} SR(sr_i)^4$$

[4] Mit $SR(sr_1) \uplus SR(sr_2) := \{(g_1 \wedge g_2, a_1 \cup a_2) \mid (g_1, a_1) \in SR(sr_1), (g_2, a_2) \in SR(sr_2)\}$.

mit

$$SR(sr) := \{(\text{guard}(sr), \{\text{action}(sr)\}), (\neg\ \text{guard}(sr), \emptyset)\}.$$

Wir zeigen nun, dass $\text{action_set}(st, \sigma_0)$ *genau diejenigen Mengen von Aktionen berechnet, die in* $\text{action_set}_K((g, st))$ *enthalten sind und erfüllte Aktivierungsbedingungen haben.*

Aus $(g', \mathcal{ACT}) \in \text{action_set}_K((g, st))$ *und* $\sigma_0 \models g'$ *folgt außerdem* $\sigma_0 \models g$*. Den Beweis zeigen wir mit Induktion über die Anzahl statischer Reaktionen* $R = \{sr \mid sr \in \text{sreact}(s), s \in \text{active}(st)\}$ *(Semantik) bzw.* $\{sr \mid sr \in \text{sreact}(s), s \in \text{active}_K(g, st)\}$ *(Kalkül) der aktiven Zuständen. Beide Mengen sind nach Lemma 6.4 für eine Schritt* (g, st) *mit* $\sigma_0 \models g$ *gleich.*

1. *Induktionsbeginn:* $R = \emptyset$*.*

   Dann ist offensichtlich $\text{action_set}(st, \sigma_0) = \text{actions}(st)$ *und* $\text{action_set}_K((g, st)) = \{(g, \text{actions}(st))\}$*.*

2. $R = R' \cup \{sr\}$*.*

   Nach Induktionsvoraussetzung ist für R' *genau dann* $\mathcal{ACT}' \in \text{action_set}(st, \sigma_0)$*, wenn ein* $(g', \mathcal{ACT}') \in \text{action_set}_K((g, st))$ *und* $\sigma_0 \models g'$*. Betrachten wir nun die statische Reaktion* sr *mit* $sr \in \text{sreact}(s)$ *und* $s \in \text{active}(st)$*.*

   Wenn wir zu R' *die statische Reaktion* sr *hinzufügen, ist nach Konstruktion*

   $$\begin{matrix} (g' \wedge \text{guard}(sr), \mathcal{ACT}' \cup \{\text{action}(sr)\}), \\ (g' \wedge \neg\text{guard}(sr), \mathcal{ACT}' \cup \{\emptyset\}) \end{matrix} \in \text{action_set}_K((g, st)),$$

   für alle $(g', \mathcal{ACT}') \in \text{action_set}_K((g, st))$*, die bezüglich* R' *berechnet wurden. Wenn* $\sigma_0 \models \text{guard}(sr)$*, dann gilt* $g' \wedge \text{guard}(sr)$ *und* $\text{action_set}(st, \sigma_0) = \mathcal{ACT}' \cup \{sr\}$*, sonst gilt* $g' \wedge \neg\text{guard}(sr)$ *und* $\text{action_set}(st, \sigma_0) = \mathcal{ACT}'$*.*

Damit ist $(g', \mathcal{ACT}) \in \text{action_set}_K((g, st))$ *und* $\sigma_0 \models g$ *genau dann, wenn* $\mathcal{ACT}$ *in der Menge* $\text{action_set}(st, \sigma_0)$ *enthalten ist. Die beiden Funktionen berechnen also die gleichen Aktionenmengen.*

Außerdem gilt für $(g', \mathcal{ACT}) \in \text{action_set}_K((g, st))$ *mit* $\sigma_0 \models g'$ *auch* $\sigma_0 \models g$*, da die Bedingung* g *in* g' *konjunktiv verknüpft enthalten ist* ($g' := g \wedge \ldots$)*.* □

Bisher haben wir gezeigt, dass genau dann in der Semantikdefinition ein Schritt st und eine Aktionenmenge $\mathit{action_set}(st, \sigma_0)$ berechnet wird, wenn im Kalkül ein Paar (g, st) und eine Aktionenmenge $(g', \mathcal{ACT}) \in \mathit{action_set}_K((g, st))$ mit $\sigma_0 \models g'$ und $\mathcal{ACT} = \mathit{action_set}(st, \sigma_0)$ berechnet wird. D. h. in der Semantikdefinition und im Kalkül werden die gleichen auszuführenden Aktionenmengen mit erfüllter Aktivierungsbedingung berechnet. Betrachten wir uns nun nochmals die Gleichung (6). Für einen Schritt $(g, st, \mathcal{ACT}) \in \mathit{Steps}_K(\Gamma \wedge \neg\Delta)$ ist $\mathit{cond}((g, st, \mathcal{ACT})) = g$. In der Disjunktion der Gleichung (6) sind damit genau diejenigen Bedingungen $\mathit{cond}((g, st, \mathcal{ACT}))$ erfüllt, deren Übergangsmengen st auch in der Semantikdefinition mit $\mathit{Steps}(\sigma_0)$ berechnet werden. Insbesondere bedeutet dies, wenn $st = \emptyset$

wird ein Makro-, ansonsten Mikro-Schritt ausgeführt. Wegen

$$step((g, st, \mathcal{ACT})) := \begin{cases} st = \emptyset, & step_{makro}((g, st, \mathcal{ACT})) \\ \text{sonst}, & step_{mikro}((g, st, \mathcal{ACT})) \end{cases}$$

zeigen wir im Folgenden, dass die Relation ρ_{micro} dem Programm $step_{mikro}$ und die Relation ρ_{macro} dem Programm $step_{makro}$ entspricht, also die gleichen Nachfolgebelegungen ergeben. ρ_{micro} und ρ_{macro} stammen aus der Semantikdefinition, $step_{mikro}$ und $step_{makro}$ aus dem Kalkül.

Mikro-Schritt

Lemma 6.5 *Mikro-Schritt*
Für ein Intervall $(\sigma_0, \sigma_1, \ldots)$ *mit einer konsistenten Belegung* $\downarrow(\sigma_0)$*, einen Schritt* $st \in \text{Steps}(\sigma_0)$*, der Aktionenmenge* $\mathcal{ACT} = \text{action_set}(st,\sigma_0)$ *und dem entsprechenden Schritt* $(g, st, \mathcal{ACT}) \in \text{Steps}_K(\Gamma \wedge \neg\Delta)$ *mit* $\sigma_0 \models g$ *gilt:*

$$\sigma_0 \rho_{micro} \sigma_1 \textit{ und } (\sigma_1, \ldots) \models SC$$

genau dann, wenn

$$(\sigma_0, \sigma_1, \ldots) \models \text{step}_{micro}((g, st, \mathcal{ACT})) \wedge \circ\, \text{nxt}((g, st, \mathcal{ACT}))$$

Einen Mikro-Schritt ist definiert als

$$\sigma_0 \rho_{micro} \sigma_1 :\Leftrightarrow \quad \exists\, st \in Steps(\sigma_0), n \in \mathbb{N}, \alpha = \alpha_1; \ldots; \alpha_n \text{ mit } \{\alpha_1, \ldots, \alpha_n\} \in action_set(st, \sigma_0) \text{ und } \sigma_0 \rho_{micro_{\alpha,st}} \sigma_1$$

wobei

$$\rho_{micro_{\alpha,st}} := \rho_{alive} \circ \rho_{copy} \circ \rho_{reset} \circ \rho_{exec_\alpha} \circ \rho_{upd_{st}} \circ \rho_{set} \circ \rho_{states_{st}}.$$

Wir gehen im Folgenden davon aus, dass es eine Belegung σ' mit $\sigma_0 \rho_{alive} \sigma'$ gibt, das Statechart also noch nicht terminiert wurde. Ansonsten würde die Statechartformel SC nicht mehr auf der Sequenz stehen (siehe dazu nxt((g,st,$\mathcal{ACT}$)) weiter unten) und die *sc unwind*-Regel könnte nicht angewendet werden. Kann jedoch ein Mikro-Schritt angewendet werden, gilt $step_{mikro}$ mit

$$step_{mikro}((g, st, \mathcal{ACT})) := \bigvee_{\vec{\alpha} \in perm(\mathcal{ACT})} step_{micro_{\vec{\alpha}}}((g, st, \mathcal{ACT}))$$

und

$$step_{micro_{\vec{\alpha}}}((g, st, \mathcal{ACT})) := \langle copy_K; reset^l_K; reset^e_K; exec_K(\vec{\alpha}); upd_ex_K(T); upd_en_K(g, T)\rangle set_K$$

Damit existiert sowohl in der Semantikdefinition als auch im Kalkül für jede mögliche Ausführungsreihenfolge der Aktionen in $\mathcal{ACT} = \{\alpha_1, \ldots, \alpha_n\}$ eine entsprechende Relation ($perm(\mathcal{ACT})$ berechnet alle möglichen Ausführungsreihenfolgen). Deshalb genügt es im

Folgenden eine feste Ausführungsreihenfolge $\alpha := \alpha_1; \ldots; \alpha_n$ bzw. $\vec{\alpha} :=< \alpha_1, \ldots, \alpha_n >$ zu betrachten und wir zeigen, dass dafür die Relation $\rho_{micro_{\alpha,st}}$ dem DL-Programm $step_{micro_{\vec{\alpha}}}$ entspricht.

Für ein Programm α gilt: $(\sigma_0, \sigma_1, \ldots) \models \langle\alpha\rangle\ \varphi$ genau dann, wenn es ein τ mit $\sigma_0[\![\alpha]\!]\tau$ und $(\tau, \sigma_1, \ldots) \models \varphi$ gibt. Wir müssen also zeigen, dass für ein Intervall $(\sigma_0, \sigma_1, \ldots)$, $\alpha = \alpha_1; \ldots; \alpha_n$ und $\vec{\alpha} = \alpha_1, \ldots, \alpha_n$ genau dann, wenn

$$\sigma_0 \rho_{micro_{\alpha,st}} \sigma_1 \text{ und } (\sigma_1, \ldots) \models SC$$

ein τ existiert, mit

$$\begin{aligned}
&\sigma\ [\![copy_K; reset^l_K; reset^e_K; exec_K(\vec{\alpha}); upd_ex_K(st); upd_en_K(g, st)]\!]\tau,\\
&(\tau, \sigma_1, \ldots) \models set_K \text{ und}\\
&(\tau, \sigma_1, \ldots) \models\ \circ\ nxt((g, st, \mathcal{ACT})).
\end{aligned}$$

Wir zeigen dazu, dass sich die Relation ρ_{copy} und das Programm $copy_K$, sowie ρ_{reset} und $reset^l_K;reset^e_K$, ρ_{exec_α} und $exec_K(\vec{\alpha})$ und $\rho_{upd_{st}}$ und $upd_ex_K(st); upd_en_K(g, st)$ entsprechen (siehe Tabelle 6.3).

Semantik	Kalkül
ρ_{copy}	$copy_K$
Kopieren der Variablenwerte in Hilfsvariablen.	
ρ_{reset}	$reset^l_K;reset^e_K$
Zurücksetzen der Ereignisse.	
ρ_{exec_α}	$exec_K(\vec{\alpha})$
Ausführen der Aktionen eines Statechart-Schrittes.	
$\rho_{upd_{st}}$	$upd_ex_K(st); upd_en_K(g, st)$
Setzen der abgeleiteten Ereignisse.	
ρ_{set}	set_K
Setzen der Variablen des Nachfolgezustands.	
$\rho_{states_{st}}$	$nxt((g, st, \mathcal{ACT}))$
Setzen der aktiven Zustände des Nachfolgezustands.	

Tabelle 6.3: Schrittausführung: Semantik vs. Kalkül

Lemma 6.6 *copy*
Für alle Belegungen σ und die Menge der Hilfsvariablen $\widetilde{variables}(SC)$ gilt

$$\sigma \rho_{copy} \sigma' \textit{ genau dann, wenn } \sigma [\![\mathrm{copy}_K]\!] \sigma'.$$

Beweis 6.5 *copy (Lemma 6.6)*
Für $\sigma \rho_{copy} \sigma' :\Leftrightarrow \sigma' = \mathrm{copy}(\sigma)$ mit

$$\mathrm{copy}(\sigma)(v) := \begin{cases} \sigma(\dot{v}), & \textit{wenn } v \in \widetilde{vars}(SC) \cup \widetilde{events}(SC) \\ \sigma(v), & \textit{sonst} \end{cases}$$

gilt $\sigma' = \sigma[v_1/\dot{v}_1, \ldots, v_n/\dot{v}_n]$ für $\{\dot{v}_1, \ldots, \dot{v}_n\} \in \mathrm{events}(SC) \cup \mathrm{vars}(SC)$. Dies entspricht der Semantik der parallelen Zuweisung

$$\mathrm{copy}_K := v_1 := \dot{v}_1 \mid \ldots \mid v_n := \dot{v}_n, \textit{ für } \dot{v}_n \in \mathrm{vars}(SC) \cup \mathrm{events}(SC)$$

nach Anhang E. □

Lemma 6.7 *reset*
Für alle Belegungen σ und die Menge der Hilfsvariablen $\widetilde{variables}(SC)$ gilt

$$\sigma \rho_{reset} \sigma' \textit{ genau dann, wenn } \sigma [\![\mathrm{reset}^l_K ; \mathrm{reset}^e_K]\!] \sigma'.$$

Beweis 6.6 *reset (Lemma 6.7)*
Für $\sigma \rho_{reset} \sigma'$ gilt $\sigma' = \mathrm{reset}(\sigma)$ mit

$$\mathrm{reset}(\sigma)(v) := \begin{cases} \mathrm{false}, & \textit{wenn } v \in \widetilde{\mathrm{events}}_{loc}(SC) \cup \widetilde{\mathrm{events}}_{imp}(SC) \cup \{\widetilde{\mathrm{tick}}\} \\ \mathrm{false}, & \textit{wenn } v \in \widetilde{\mathrm{events}}_{env}(SC) \textit{ und } \sigma(\mathrm{tick}) \\ \sigma(v), & \textit{sonst.} \end{cases}$$

Wir unterscheiden zwei Fälle:

1. *es gilt* tick, *dann ist $\sigma' = \sigma[e_1/\mathrm{false}, \ldots, e_n/\mathrm{false}]$ für*

$$\{\dot{e}_1, \ldots \dot{e}_n\} = \{\dot{e} \mid \dot{e} \in \mathrm{events}_{loc}(SC) \cup \mathrm{events}_{imp}(SC) \cup \{\mathrm{tick}\} \cup \mathrm{events}_{env}(SC)\},$$

2. *es gilt nicht* tick, *dann ist $\sigma' = \sigma[e_1/\mathrm{false}, \ldots, e_m/\mathrm{false}]$ für*

$$\{\dot{e}_1, \ldots \dot{e}_m\} = \{\dot{e} \mid \dot{e} \in \mathrm{events}_{loc}(SC) \cup \mathrm{events}_{imp}(SC) \cup \{\mathrm{tick}\}\},$$

d. h. die lokalen Ereignisse werden immer, die Umgebungsereignisse nur wenn tick *gilt zurückgesetzt. Mit*

$$\begin{array}{lll} \mathrm{reset}^l_K & := & e_1 := \mathrm{false} \mid \ldots \mid e_n := \mathrm{false}, \textit{ für } \dot{e}_i \in \mathrm{events}_{loc}(SC) \cup \mathrm{events}_{imp} \cup \{\dot{\mathrm{tick}}\} \\ \mathrm{reset}^e_K & := & \textbf{\textit{if}}\ \dot{\mathrm{tick}}\ \textbf{\textit{then}}\ e_1 := \mathrm{false} \mid \ldots \mid e_m := \mathrm{false}, \textit{ für } \dot{e}_i \in \mathrm{events}_{env}(SC) \end{array}$$

entspricht dies der Ausführung von reset^l_K;reset^e_K. In reset^e_K werden die Umgebungsereignisse genau dann zurückgesetzt, wenn tick *gilt (siehe Semantik **if** … **then**, Anhang E).* □

Lemma 6.8 *execute*
Für alle Belegungen σ, die Menge der Hilfsvariablen $\widetilde{variables}(SC)$, $\alpha = \alpha_1;\ldots;\alpha_n$ und $\vec{\alpha} = \alpha_1,\ldots,\alpha_n$ gilt

$$\sigma\rho_{exec_\alpha}\sigma' \textit{ genau dann, wenn } \sigma[\![\mathrm{exec}_K(\vec{\alpha})]\!]\sigma'.$$

Beweis 6.7 *execute (Lemma 6.8)*
$\sigma\rho_{exec_\alpha}\sigma'$ *ist, genauso wie* $\sigma[\![\mathrm{exec}_K(\vec{\alpha})]\!]\sigma'$, *definiert als* $\sigma[\![\alpha_1;\ldots;\alpha_n]\!]\sigma'$. □

Lemma 6.9 *update*
Für alle Belegungen σ und die Menge der Hilfsvariablen $\widetilde{variables}(SC)$ gilt

$$\sigma\rho_{upd_{st}}\sigma' \textit{ genau dann, wenn } \sigma[\![\mathrm{upd_ex}_K(st);\mathrm{upd_en}_K(g,st)]\!]\sigma'.$$

Beweis 6.8 *update (Lemma 6.9)*
Für $\sigma\rho_{upd_{st}}\sigma'$ *gilt* $\sigma' = \mathrm{upd}_{impl}(st,\sigma)$ *mit*

$$\mathrm{upd}_{impl}(st,\sigma)(v)\begin{cases}\mathrm{true}, & \textit{wenn } v \equiv e,\ \dot{e} \equiv \mathrm{ex}(s) \textit{ und } s \in \mathrm{exited}(st)\\ \mathrm{true}, & \textit{wenn } v \equiv e,\ \dot{e} \equiv \mathrm{en}(s) \textit{ und } s \in \mathrm{entered}(st)\\ \sigma(v), & \textit{sonst.}\end{cases}$$

Damit ist $\sigma' = \sigma[\mathrm{e}_1/\mathrm{true},\ldots,e_n/\mathrm{true},e_{n+1}/\mathrm{true},\ldots,e_m/\mathrm{true}]$ *für*

$$\{\dot{e}_1,\ldots,\dot{e}_n\} = \{\dot{e} \mid \dot{e} \equiv en(s) \textit{ und } s \in \mathrm{exited}(st)\}$$
$$\{\dot{e}_{n+1},\ldots,\dot{e}_m\} = \{\dot{e} \mid \dot{e} \equiv ex(s) \textit{ und } s \in \mathrm{entered}(st)\}.$$

Dies gilt ebenso für $\sigma[\![\mathrm{upd_ex}_K(st);\mathrm{upd_en}_K(g,st)]\!]\sigma'$ *mit*

$$\begin{array}{ll}\mathrm{upd_ex}_K(st) \ := & e_1 := s_1 \mid \ldots \mid e_n := s_n, \dot{e}_{s_i} \in \{\dot{e}_s \mid \dot{e}_s \equiv \mathrm{ex}(s) \wedge s \in \mathrm{changed}_K(st)\}\\ \mathrm{upd_en}_K(g,st) := & e_1 := \mathrm{true} \mid \ldots \mid e_n := \mathrm{true}, \dot{e}_i \in \{\dot{e} \mid \dot{e} \equiv \mathrm{en}(s) \wedge s \in \mathrm{entered}_K(g,st)\}.\end{array}$$

In $\mathrm{changed}_K(st) := \{s \mid \exists\, t.\ t \in T \textit{ mit } \mathrm{scope}(t) < s\}$ *sind neben den Zuständen s die verlassen werden ($s = \mathrm{true}$) noch diejenigen Zustände s' enthalten, die nicht aktiv sind ($s' = \mathrm{false}$), aber betreten werden.* $\mathrm{upd_ex}_K(st)$ *setzt also genau die abgeleiteten Ereignisse auf* true, *deren Zustände in* $\mathrm{exited}(st) := \{s \mid \exists\, t.\ t \in st \textit{ mit } \mathrm{scope}(t) < s\} \cap \{s \mid \sigma(s)\}$ *sind. Die anderen abgeleiteten Ereignisse sind in σ_0 auf* false *initialisiert und werden mit* $\mathrm{upd_ex}_K(st)$ *(nochmals) auf* false *gesetzt.*

Die Menge der Zustände, die in einem Schritt betreten werden, berechnet sich durch $\mathrm{entered}(st) := \mathrm{decompl}(\mathrm{active}(st) \cup \bigcup_{t \in st}\{\mathrm{target}(t)\})\backslash\mathrm{active}(st)$ *bzw.* $\mathrm{entered}_K(g,T) := \mathrm{decompl}(\mathrm{active}_K(g,T) \cup \bigcup_{t \in T}\{\mathrm{target}(t)\})\backslash\mathrm{active}_K(g,T)$. *Nach Lemma 6.4 sind diese beiden Mengen gleich.* □

Mit Lemma 6.6 bis 6.9 folgt, dass $\sigma(\rho_{copy} \circ \rho_{reset} \circ \rho_{exec_\alpha} \circ \rho_{upd_{st}})\sigma'$ genau dann, wenn $\sigma[\![copy_K; reset^l_K; reset^e_K; exec_K(\vec{\alpha}); upd_ex_K(st); upd_en_K(g,st)]\!]\sigma'$. Damit kehren wir nun zum Beweis von Lemma 6.5 zurück.

Beweis 6.9 *Mikro-Schritt (Lemma 6.5)*
Es bleibt zu zeigen, dass genau dann, wenn für ein Intervall $(\sigma_0, \sigma_1, \ldots)$, $\alpha = \alpha_1; \ldots; \alpha_n$ *und* $\vec{\alpha} = \alpha_1, \ldots, \alpha_n$

$$\sigma_0 \rho_{micro_{\alpha,st}} \sigma_1 \textit{ und } (\sigma_1, \ldots) \models SC$$

es ein τ *gibt, mit*

$$\begin{array}{l} \sigma \; [\![\mathrm{copy}_K; \mathrm{reset}^l_K; \mathrm{reset}^e_K; \mathrm{exec}_K(\vec{\alpha}); \mathrm{upd_ex}_K(st); \mathrm{upd_en}_K(g, st)]\!] \tau, \\ (\tau, \sigma_1, \ldots) \models \mathrm{set}_K \textit{ und} \\ (\tau, \sigma_1, \ldots) \models \circ\, \mathrm{nxt}((g, st, \mathcal{ACT})) \end{array}$$

wobei $\mathrm{set}_K := V'' = V$ *mit* $\dot{V} = \dot{v}_1, \ldots, \dot{v}_n$, $\dot{v}_i \in \mathrm{events}(SC) \cup \mathrm{vars}(SC)$, V'' *die entsprechenden 2-fach gestrichenen Variablen und* V *die entsprechenden Hilfsvariablen sind. Mit*

$$\rho_{micro_{\alpha,st}} := \rho_{alive} \circ \rho_{copy} \circ \rho_{reset} \circ \rho_{exec_\alpha} \circ \rho_{upd_{st}} \circ \rho_{set} \circ \rho_{states_{st}}$$

und Lemma 6.6 bis 6.9 folgt, dass $\sigma(\rho_{alive} \circ \rho_{copy} \circ \rho_{reset} \circ \rho_{exec_\alpha} \circ \rho_{upd_{st}})\tau$ *gilt (*ρ_{alive} *ändert die Belegung nicht). Es bleibt zu zeigen, dass* $\tau(\rho_{set} \circ \rho_{upd_{st}})\sigma_1$ *genau dann, wenn* $(\tau, \sigma_1, \ldots) \models \mathrm{set}_K$ *und* $(\tau, \sigma_1, \ldots) \models \circ\, \mathrm{nxt}((g, st, \mathcal{ACT}))$.

1. *Es gilt* $\tau(\rho_{set} \circ \rho_{upd_st})\sigma_1$ *mit*

$$\begin{array}{rcl} & & \{\dot{v}_1, \ldots, \dot{v}_n\} = \mathrm{vars}(SC) \cup \mathrm{events}(SC), \\ S & = & \{\dot{s}_1, \ldots, \dot{s}_n\} = \mathrm{active}(st) \cup \mathrm{entered}(st), \\ \overline{S} & = & \{\dot{s}_{n+1}, \ldots, \dot{s}_m\} = states(SC) \backslash (\mathrm{active}(st) \cup \mathrm{entered}(st)) \\ \sigma \rho_{set} \sigma' & :\Leftrightarrow & \sigma' = \mathrm{set}(\sigma), \\ \mathrm{set}(\sigma)(v) & := & \left\{ \begin{array}{ll} \sigma(x), & \textit{wenn } v \equiv \dot{x}, \dot{x} \in \mathrm{events}(SC) \cup \mathrm{vars}(SC) \\ \sigma(v), & \textit{sonst} \end{array} \right. , \\ \sigma' \rho_{upd_st} \tau & :\Leftrightarrow & \tau = \mathrm{upd}_{states}(st, \sigma') \textit{ und} \end{array}$$

$$\mathrm{upd}_{states}(st, \sigma)(v) := \left\{ \begin{array}{ll} \mathrm{true}, & \textit{wenn } v \in S \\ \mathrm{false}, & \textit{wenn } v \in \overline{S} \\ \sigma(v), & \textit{sonst} \end{array} \right.$$

$$\sigma_1 = \tau[\dot{v}_1/\tau(v_1), \ldots, \dot{v}_n/\tau(v_n), \dot{s}_1/\mathrm{true}, \ldots \dot{s}_n/\mathrm{true}, \dot{s}_{n+1}/\mathrm{false}, \ldots \dot{s}_m/\mathrm{false}].$$

In einem Intervall $(\sigma_0, \sigma_1, \ldots)$ *gilt für alle flexiblen Variablen* $\dot{x}$, *dass* $\sigma_i(x'') = \sigma_{i+1}(\dot{x})$. *Für alle Variablen* $\dot{v} \in \mathrm{vars}(SC) \cup \mathrm{events}(SC)$ *gilt* $\tau(v) = \sigma_1(\dot{v})$ *und mit* $\sigma_0(v'') = \sigma_1(\dot{v})$ *auch* $\tau(v) = \sigma_0(v'')$. *Damit gilt für* $\dot{x}_i \in \mathrm{vars}(SC) \cup \mathrm{events}(SC)$ *in* τ $\bigwedge_i v_i = v''_i$ *bzw.* set_K *und, da* set_K *prädikatenlogisch, auch* $(\tau, \sigma_1, \ldots) \models \mathrm{set}_K$.

Weiterhin gilt, dass $(\sigma_1, \ldots) \models \bigwedge_{s \in S} s \wedge \bigwedge_{\overline{s} \in \overline{S}} \neg\, \overline{s}$ *und mit* $(\sigma_1, \ldots) \models SC$ *gilt* σ_1 *hat keinen Nachfolger genau dann, wenn ein* $s \in \mathrm{term}(SC)$ *mit* $\sigma_1(s)$ *existiert. Dann gilt nicht* $\sigma_1 \rho_{alive} \sigma'$, *für alle* σ', *das Restintervall* $(\sigma_1, \ldots) = (\sigma_1)$ *hat die Länge* 0,

es gilt **last**. *Ansonsten gilt* $(\sigma_1,\ldots) \models SC$ *und* σ_1 *hat weitere Nachfolger. So gilt für* $\mathrm{nxt}((g, st, \mathcal{ACT}))$ *mit*

$$\begin{array}{lll} \mathrm{nxt}((g, st, \mathcal{ACT})) & := & \left\{ \begin{array}{ll} \mathrm{term}(SC) \cap & \\ \quad \mathrm{entered}_K(st) \neq \emptyset, & \bigwedge_{s \in S_K} s \wedge \bigwedge_{\overline{s} \in \overline{S}_K} \neg\ \overline{s} \wedge \mathbf{last} \\ sonst, & \bigwedge_{s \in S_K} s \wedge \bigwedge_{\overline{s} \in \overline{S}_K} \neg\ \overline{s} \wedge SC \end{array} \right. , \\ S_K & := & \{\dot{s}_1, \ldots, \dot{s}_n\} = \mathrm{active}_K(g, st) \cup \mathrm{entered}_K(g, st) \ und \\ \overline{S}_K & := & \{\dot{s}_{n+1}, \ldots, \dot{s}_m\} \\ & & = \mathrm{states}(SC) \backslash (\mathrm{active}_K(g, st) \cup \mathrm{entered}_K(g, st)), \end{array}$$

dass $(\sigma_1,\ldots) \models nxt(st)$ *und damit, dass* $(\tau, \sigma_1,\ldots) \models \circ\, \mathrm{nxt}((g, st, \mathcal{ACT}))$. *Beachte, nach Lemma 6.4 sind die Mengen* $S = S_K$ *und* $\overline{S} = \overline{S}_K$.

2. *Es gilt* $(\tau, \sigma_1,\ldots) \models V'' = V$ *und* $(\tau, \sigma_1,\ldots) \models \circ\, \mathrm{nxt}((g, st\mathcal{ACT}))$.

   Das DL-Programm zur Schrittausführung ändert keine flexiblen Variablen x, x' *und* x'' *und deshalb ist* $\sigma_0(x'') = \tau(x'')$. *Für alle flexiblen Variablen* x *gilt* $\sigma_0(x'') = \sigma_1(\dot{x})$ *und für alle* $\dot{v} \in \mathrm{vars}(SC) \cup \mathrm{events}(SC)$ *gilt* $\tau \models v'' = v$. *Für die Belegungen eines Intervalls folgt, dass* $\sigma_1(\dot{v}) = \tau(v)$ *und mit* $(\sigma_1,\ldots) \models \mathrm{nxt}((g, st, \mathcal{ACT}))$, *dass* $\tau(\rho_{set} \circ \rho_{upd_st})\sigma_1$.

□

Makro-Schritt

Lemma 6.10 *Makro-Schritt*
Für ein Intervall $(\sigma_0, \sigma_1, \ldots)$ *mit einer konsistenten Belegung* $\downarrow(\sigma_0)$, $\mathrm{Steps}(\sigma) = \emptyset$ *und dem entsprechenden Schritt* $(g, st, \emptyset) \in \mathrm{Steps}_K(\Gamma \wedge \neg\Delta)$ *mit* $\sigma \models g$ *gilt:*

$$\sigma_0 \rho_{macro} \sigma_1 \ und\ (\sigma_1, \ldots) \models SC$$

genau dann, wenn

$$(\sigma_0, \sigma_1, \ldots) \models \mathrm{step}_{macro}((g, st, \mathcal{ACT})) \wedge \circ\, \mathrm{nxt}((g, st, \mathcal{ACT})).$$

Ein Makro-Schritt ist in der Semantik definiert als

$$\rho_{macro} := \rho_{stable} \circ \rho_{alive} \circ \rho_{copy} \circ \rho_{reset} \circ \rho_{set} \circ \rho_{tick} \circ \rho_{input}$$

mit

$$\begin{array}{ll} \sigma \rho_{stable} \sigma' & :\Leftrightarrow Steps(\sigma) = \emptyset \wedge \sigma = \sigma' \\ \sigma \rho_{alive} \sigma' & :\Leftrightarrow \forall\ s \in term(s).\ \sigma(s) = \mathrm{false\ und\ } \sigma = \sigma' \end{array}$$

und wird im Kalkül durch das Programm

$$step_{makro}((g, st, \mathcal{ACT})) := \langle copy_K; reset^l_K; reset^e_K \rangle set^e_K \wedge set^l_K$$

beschrieben.

Beweis 6.10 *Makro-Schritt (Lemma 6.10)*
Für ein Intervall $(\sigma_0, \sigma_1, \ldots)$ *gibt es ein* τ*, so dass*

$$\sigma_1(\rho_{stable} \circ \rho_{alive} \circ \rho_{copy} \circ \rho_{reset})\tau,\ \tau(\rho_{set} \circ \rho_{tick} \circ \rho_{input})\sigma_1 \textit{ und } (\sigma_1, \ldots) \models SC$$

genau dann, wenn

$$\sigma_1[\![\mathrm{copy}_K; \mathrm{reset}^l_K]\!]\tau,\ (\tau, \sigma_1, \ldots) \models \mathrm{set}^e_K \wedge \mathrm{set}^l_K \textit{ und } (\tau, \sigma_1, \ldots) \models \circ\,\mathrm{nxt}((g, st, \mathcal{ACT}))$$

Wegen $\mathrm{Steps}(\sigma) = \emptyset$ *gilt auch* $\sigma\rho_{stable}\sigma$*. Weiterhin gehen wir davon aus, dass ein Schritt möglich ist, das Statechart also noch nicht terminiert wurde. Damit gilt* $\sigma\rho_{alive}\sigma$*. Ansonsten wäre* $s \in \mathrm{term}(SC)$ *mit* $\sigma(s)$*. Wegen* $\mathrm{nxt}(stp')$ *des vorhergehenden Schrittes* stp' *würde* **last** *gelten, keine Statechart-Formel SC auf der Sequenz stehen und die* sc unwind-*Regel wäre nicht anwendbar gewesen. Da* ρ_{stable} *und* ρ_{alive} *die Belegung nicht ändern, folgt mit Lemma 6.6 und Lemma 6.7, dass es ein* τ *gibt, mit*

$$\sigma_1(\rho_{stable} \circ \rho_{alive} \circ \rho_{copy} \circ \rho_{reset})\tau$$

genau dann, wenn

$$\sigma_1[\![\mathrm{copy}_K; \mathrm{reset}^l_K; \mathrm{reset}^e_K]\!]\tau.$$

Es bleibt zu zeigen, dass wenn

1. $\tau(\rho_{set} \circ \rho_{tick} \circ \rho_{input})\sigma_1$ *und* $(\sigma_1, \ldots) \models SC$*, dann*

$$(\tau, \sigma_1, \ldots) \models \mathrm{set}^e_K \wedge \mathrm{set}^l_K \textit{ und } (\tau, \sigma_1, \ldots) \models \circ\,\mathrm{nxt}((g, st, \mathcal{ACT})) \textit{ und}$$

2. $(\tau, \sigma_1, \ldots) \models \mathrm{set}^e_K \wedge \mathrm{set}^l_K$ *und* $(\tau, \sigma_1, \ldots) \models \circ\,\mathrm{nxt}((g, st, \mathcal{ACT}))$*, dann*

$$\tau(\rho_{set} \circ \rho_{tick} \circ \rho_{input})\sigma_1 \textit{ und } (\sigma_1, \ldots) \models SC.$$

Sei

$$\begin{array}{rcl}
\{\dot v_1, \ldots, \dot v_n\} & = & \mathrm{vars}(SC) \cup \mathrm{events}(SC), \\
\{\dot v_{n+1}, \ldots, \dot v_m\} & = & \mathrm{vars}_{env}(SC) \cup \mathrm{events}_{env}(SC), \\
\sigma\rho_{set}\sigma' :\Leftrightarrow \sigma' & = & \mathrm{set}(\sigma), \\
\mathrm{set}(\sigma)(v) & := & \left\{ \begin{array}{ll} \sigma(v), & \textit{wenn } v \equiv \dot x, \dot x \in \mathrm{events}(SC) \cup \mathrm{vars}(SC) \\ \sigma(v), & \textit{sonst} \end{array} \right. , \\
\sigma\rho_{tick}\sigma' & :\Leftrightarrow & \sigma' = \sigma[\mathrm{tick}/\mathrm{true}], \\
\sigma\rho_{input}\sigma' & :\Leftrightarrow & \forall\ v.\ v \in \mathrm{variables}(SC)/(\mathrm{vars}_{env}(SC) \cup \mathrm{events}_{env}(SC)) \\
 & & \quad \Rightarrow \sigma(v) = \sigma'(v), \\
S & = & \{\dot s_1, \ldots, \dot s_n\} = \mathrm{active}_K(g, st) \cup \mathrm{entered}_K(g, st), \\
\overline{S} & = & \{\dot s_{n+1}, \ldots, \dot s_m\} = \mathit{states}(SC) \backslash (\mathrm{active}_K(g, st) \cup \mathrm{entered}_K(g, st))
\end{array}$$

$$
\begin{aligned}
\text{nxt}((g, st, \mathcal{ACT})) &:= \begin{cases} \text{term}(SC) \cap \text{entered}(st) \neq \emptyset, & \bigwedge_{s \in S} s \wedge \bigwedge_{\overline{s} \in \overline{S}} \neg\, \overline{s} \wedge \textbf{last} \\ \textit{sonst,} & \bigwedge_{s \in S} s \wedge \bigwedge_{\overline{s} \in \overline{S}} \neg\, \overline{s} \wedge SC \end{cases}, \\
\text{set}^e_K &:= \text{tick}' = \text{true} \wedge \textit{tick}'' = \text{true} \wedge \bigvee_{\dot{v} \in events_{env} \,\cup\, vars_{env}} v' = v, \\
\text{set}^l_K &:= \bigvee_{\dot{v} \in events_{loc} \,\cup\, events_{imp} \,\cup\, vars_{loc}} v'' = v \;\; \textit{und} \\
\text{output}(\sigma_i) &:\Leftrightarrow \forall\, \dot{v} \in \text{events}_{env}(SC) \cup \text{vars}_{env}(SC).\ \dot{v} = v' \;\textit{und}\; \sigma_i(\textit{tick}').
\end{aligned}
$$

1. *Mit* $\tau(\rho_{set} \circ \rho_{tick} \circ \rho_{input})\sigma_1$ *ist*

$$\sigma_1 = \tau[\dot{v}_1/\tau(v_1), \ldots, \dot{v}_n/\tau(v_n), \text{tick}/\text{true}, \dot{v}_{n+1}/x_{n+1}, \ldots, \dot{v}_m/x_m]$$

für neue Variablen $x_{n+1} \ldots x_m$. *D. h. die Variablen* $\dot{v}_{n+1}, \ldots, \dot{v}_m$ *haben einen beliebigen Wert in* σ_1. *In einem Intervall* $(\sigma_0, \sigma_1, \ldots)$ *gilt für alle flexiblen Variablen* $\dot{x}$, *dass* $\sigma_i(x'') = \sigma_{i+1}(\dot{x})$. *Deshalb gilt* $\sigma_0(v'') = \sigma_1(\dot{v})$ *und da* $\tau(v) = \sigma_1(\dot{v})$, *dass* $\tau(v) = \sigma_0(v'')$, *für alle Variablen* v'' *mit* $\dot{v} \in \text{variables}(SC) \backslash (\text{vars}_{env}(SC) \cup \text{events}_{env}(SC))$. *Damit gilt* $\tau \models \text{set}^l_K$ *und da* set^l_K *prädikatenlogisch auch* $(\tau, \sigma_1, \ldots) \models \text{set}^l_K$.

Ebenso wegen $\sigma_0(x'') = \sigma_1(\dot{x})$ *gilt* $\tau(\text{tick}'') = \text{true}$ *und mit* $\text{output}(\sigma_0) :\Leftrightarrow \forall\, \dot{v} \in \text{events}_{env}(SC) \cup \text{vars}_{env}(SC).\ \dot{v} = v'$ *und* $\sigma_0(\dot{tick})$ *gilt* $\tau \models \text{set}^e_K$. *Da* set^e_K *prädikatenlogisch gilt auch* $(\tau, \sigma_1, \ldots) \models \text{set}^e_K$.

In einem Makro-Schritt ist $st = \emptyset$ *und für* $\sigma_0 \rho_{makro} \sigma_1$ *gilt* $\sigma_0 \mid_{states(SC)} = \sigma_1 \mid_{states(SC)}$. *Es folgt, dass* $\text{entered}(st) = \text{exited}(st) = \emptyset$, *d. h. die aktiven Zustände ändern sich nicht. Damit gilt* $(\sigma_1, \ldots) \models \text{nxt}((g, st, \mathcal{ACT}))$ *und* $(\sigma_0, \sigma_1, \ldots) \models \circ\,\text{nxt}((g, st\mathcal{ACT}))$.

2. *Wenn* $(\tau, \sigma_1, \ldots) \models \text{set}^e_K \wedge \text{set}^l_K$ *und* $(\tau, \sigma_1, \ldots) \models \circ\text{nxt}((g, st, \mathcal{ACT}))$ *gilt mit* $\sigma_0(x'') = \sigma_1(\dot{x})$, *dass die lokalen Variablen* $\dot{v}_1, \ldots, \dot{v}_n$ *in* σ_1 *den Wert von* $v_1, \ldots, v_n$ *in* τ *haben, wegen* $\tau \models \text{tick}'' = \text{true}$ *gilt* $\sigma_1(\text{tick}) = \text{true}$ *und wegen nxt(st) sind in* σ_1 *dieselben Zustände aktiv, wie in* σ_0. *Weiterhin erlaubt ein Umgebungsschritt die Umgebungsvariablen und -ereignisse beliebig zu setzen und damit folgt* $\tau(\rho_{set} \circ \rho_{tick} \circ \rho_{input})\sigma_1$ (set^e_K *setzt die 1-fach gestrichenen Variablen, so dass die Umgebung die 2-fach gestrichenen Variablen noch beliebig ändern kann). Schließlich folgt aus* $(\tau, \sigma_1, \ldots) \models \text{set}^e_K$ $\text{output}(\tau)$ *und, da das DL-Programm* step_{makro} *keine flexiblen Variablen ändert, gilt auch* $\text{output}(\sigma_0)$.

□

Damit haben wir gezeigt, dass die *sc unwind*-Regel korrekt ist. Im Wesentlichen haben wir sogar die Äquivalenz zwischen Konklusio und Prämisse der Regel gezeigt. Jedoch benötigen wir in Lemma 6.5 die Voraussetzung $\downarrow(\sigma_0)$, dass die Ausgangsbelegung konsistent ist. Diese Voraussetzung wird von der Statechart-Semantik erfüllt, aber nicht notwendigerweise von der Prämisse der *sc unwind*-Regel. Wenden wir die Regel auf eine partielle Zustandskonfiguration und ein Statechart mit den Zuständen s_1 und s_2 als Unterzustände eines *Oder*-Zustands s an und beschreiben in der Sequenz $\Gamma \vdash \Delta$, dass s_1 aktiv ist, machen aber

keine Angaben über s_2. Dann kann in einer Prämisse der *sc unwind*-Regel die Aktivierungsbedingung für einen Statechart-Schritt verlangen, dass auch s_2 aktiv ist und wir erhalten eine inkonsistente Konfiguration, in der sowohl s_1 als auch s_2 aktiv sind (siehe auch Motivation für die *sc wellformed*-Regel, Seite 86). Folgen von Belegungen, die inkonsistente Belegungen enthalten, sind aber nicht in der Statechart-Semantik enthalten. Deshalb ist die *sc unwind*-Regel nicht invertierbar.

6.4.2 Die *sc wellformed*-Regel

Theorem 6.2 *Die* sc wellformed-*Regel ist invertierbar und korrekt*
Die sc wellformed-*Regel ist invertierbar und korrekt, d. h.*

$$\models \forall_{Cl}\ cons_{states}(SC), cons_{events}(SC), SC, \Gamma \to \Delta$$

genau dann, wenn

$$\models \forall_{Cl}\ SC \wedge \Gamma \to \Delta.$$

Für die Korrektheit ist zu zeigen, dass wenn

$$\models \forall_{Cl}\ cons_{states}(SC), cons_{events}(SC), SC, \Gamma \to \Delta,$$

dann

$$\models \forall_{Cl}\ SC \wedge \Gamma \to \Delta.$$

Wir zeigen wieder die Behauptung, dass wenn

$$\mathcal{I} \models cons_{states}(SC), cons_{events}(SC), SC, \Gamma \to \Delta,$$

dann

$$\mathcal{I} \models SC \wedge \Gamma \to \Delta.$$

Entweder es gilt $\mathcal{I} \not\models \Gamma \wedge \neg\, \Delta$, dann ist der Beweis trivial. Sonst zeigen wir, dass wenn

$$\mathcal{I} \models SC :\Leftrightarrow \mathcal{I}\ |_{variables(SC)} \in traces(SC),$$

dann

$$\mathcal{I} \models cons_{states}(SC) \wedge cons_{events}(SC) \wedge SC.$$

Es gilt

$$cons_{states}(SC) := \bigvee_{cnf \in conf(SC)} s_1 \wedge \ldots \wedge s_n \wedge \neg\, \overline{s}_1 \wedge \ldots \wedge \neg\, \overline{s}_m$$

für $s_i \in cnf$, $\overline{s}_j \in states(SC) \backslash cnf$, $conf(SC) := \{S \mid S \subseteq states(SC), \downarrow(S)$ und $S = decompl(S)\}$ und

$$cons_{events}(SC) := \neg\, ((\bigvee_{e \in events_{loc}(SC)} e) \wedge tick).$$

$conf(SC)$ berechnet für ein Statechart SC alle möglichen, konsistenten Zustandskonfigurationen (siehe Abschnitt 4.2.1, Syntax).

Beweis 6.11 *Beweis Korrektheit der* sc wellformed-*Regel*
$cons_{states}(SC)$ *und* $cons_{events}(SC)$ *sind prädikatenlogische Formeln und für ein Intervall* $(\sigma_0, \sigma_1, \ldots) \in \text{traces}(SC)$ *bleibt zu zeigen, dass* $\sigma_0 \models cons_{states}(SC) \wedge cons_{events}(SC)$.

Für alle Intervalle eines Statecharts gilt, dass die Ausgangsbelegung σ_0 *konsistent ist, d. h. es gilt* $\downarrow(\sigma_0)$ *(siehe Gleichung 7) mit*

$$\begin{array}{ll} \downarrow(\sigma_0) :\Leftrightarrow & \\ \quad \downarrow(S) \textit{ und } S = \text{decompl}(S) \textit{ und} & i) \\ \quad \{e \mid e \in \text{events}_{loc}(SC) \cup \text{events}_{imp}(SC) \textit{ und } \sigma_0(e)\} \neq \emptyset \textit{ und } \sigma_0(\text{tick}) & ii) \end{array}$$

Mit i) gilt $\sigma_0 \models cons_{states}(SC)$ *und mit ii) gilt* $\sigma_0 \models cons_{events}(SC)$. □

Für die *sc wellformed*-Regel können wir auch die Invertierbarkeit zeigen.

Beweis 6.12 *Beweis Invertierbarkeit der* sc wellformed-*Regel*
Es ist zu zeigen, dass wenn

$$\models \forall_{Cl}\ SC \wedge \Gamma \to \Delta,$$

dann

$$\models \forall_{Cl}\ cons_{states}(SC), cons_{events}(SC), SC, \Gamma \to \Delta.$$

Dies ist trivial wahr, da die Prämisse der sc wellformed-*Regeln nur zusätzliche Voraussetzungen hat.* □

6.4.3 Die *sc initialize*-Regel

Theorem 6.3 *Die* sc initialize-*Regel ist invertierbar und korrekt*
Die sc initialize-*Regel ist invertierbar und korrekt, d. h.*

$$\models \forall_{Cl}\ \text{initialize}(\widehat{SC}), SC, \Gamma \to \Delta$$

genau dann, wenn

$$\models \forall_{Cl}\ \widehat{SC} \wedge \Gamma \to \Delta.$$

Für die Korrektheit ist zu zeigen, dass wenn

$$\models \forall_{Cl}\ \textit{initialize}(\widehat{SC}), SC, \Gamma \to \Delta,$$

dann

$$\models \forall_{Cl}\ \widehat{SC} \wedge \Gamma \to \Delta.$$

Wir zeigen wieder die Behauptung, dass wenn

$$\mathcal{I} \models \textit{initialize}(\widehat{SC}), SC, \Gamma \to \Delta,$$

dann

$$\mathcal{I} \models \widehat{SC} \wedge \Gamma \to \Delta.$$

Entweder es gilt $\mathcal{I} \not\models \Gamma \wedge \neg\, \Delta$, dann ist der Beweis trivial. Sonst zeigen wir, dass wenn

$$\mathcal{I} \models \widehat{SC} :\Leftrightarrow \mathcal{I}\,|_{variables(SC) \in traces^i(SC)},$$

dann

$$\mathcal{I} \models initialize(\widehat{SC}) \wedge SC$$

mit

$$\begin{aligned}&initialize(\widehat{SC}) :=\\ &\quad \textstyle\bigwedge_{s \in states(SC)\, \cap\, decompl(root(SC))} s \wedge \bigwedge_{\overline{s} \in states(SC) \setminus decompl(root(SC))} \neg\, \overline{s} \wedge\\ &\quad \textstyle\bigwedge_{e \in events(SC)} \neg\, e \wedge \bigwedge_{v \in vars(SC)} v = default(v).\end{aligned}$$

Beweis 6.13 *Korrektheit der* sc initialize-*Regel*
Wegen

$$traces^i_{\mathcal{A}}(SC) := \{(\sigma_0, \ldots, \sigma_n) \mid n \in I\!N_\infty, \sigma_j\ \rho_{step_{SC,\mathcal{A}}}\ \sigma_{j+1}\ \mathit{und}\ \sigma_0 \in \mathrm{init}(SC)\}.$$

beginnen alle Intervalle von $\widehat{SC}$ *in einem Ausgangszustand* σ_0. *Nach Definition sind dies all diejenigen* σ_0 *für die*

$$\sigma_0(v) = \begin{cases} \text{true}, & \mathit{wenn}\ v \in \text{states}(SC) \cap \text{decompl}(\text{root}(SC)) \\ \text{false}, & \mathit{wenn}\ v \in \text{states}(SC) \setminus \text{decompl}(\text{root}(SC)) \\ \text{false}, & \mathit{wenn}\ v \in \text{events}(SC) \\ \text{default}(v), & \mathit{wenn}\ v \in \text{vars}(SC) \end{cases}$$

gilt. Damit ist σ_0 *konsistent, es gilt* $\downarrow(\sigma_0)$ *und* $\mathcal{I} \models SC$. *Da* initialize($\widehat{SC}$) *prädikatenlogisch ist, bleibt zu zeigen, dass* $\sigma_0 \models$ initialize($\widehat{SC}$). *Dies gilt trivial mit* $\sigma_0 \in init(SC)$. □

Auch für die *sc initialize*-Regel können wir die Invertierbarkeit zeigen.

Beweis 6.14 *Invertierbarkeit der* sc initialize-*Regel*
Es ist zu zeigen, dass wenn

$$\models \forall_{Cl}\ \widehat{SC} \wedge \Gamma \rightarrow \Delta,$$

dann

$$\models \forall_{Cl}\ \text{initialize}(\widehat{SC}), SC, \Gamma \rightarrow \Delta.$$

Dies ist trivial wahr, da die Prämisse der sc initialize-*Regeln nur zusätzliche Voraussetzungen hat.* □

6.5 Beweistechnik und -automatisierung

6.5.1 Beweistechnik

Durch die Technik der schrittweisen Ausführung folgt der Beweis einer Statechart-Eigenschaft den Durchläufen durch das Statechart. Nach einer möglichen Initialisierung, die z. B. Ausgangszustände setzt, wird das Statechart schrittweise abgewickelt. Das Abwickeln (*sc unwind*) berechnet, ausgehend von der aktuellen Belegung, die Nachfolgebelegungen. Parallel dazu werden auch die temporallogischen Eigenschaften abgewickelt. Dies resultiert in jeweils einer Beweisverpflichtung für die aktuelle Belegung und in einer zweiten für die Nachfolgebelegung. Nachdem die Beweisverpflichtung für die aktuelle Belegung gezeigt wurde, setzt die Regel *tl step* die Nachfolgebelegung als aktuelle Belegung. Der Beweis ist nun einen Schritt auf dem Intervall fortgeschritten, den das Statechart beschreibt. Diese Beweistechnik hilft dem Benutzer, die Beweisschritte nachvollziehen zu können, da er während des Beweises immer weiß, in welcher Zustandskonfiguration sich das Statechart befindet. Dies wird durch die aktiven Zustände des Statecharts gekennzeichnet.

Bei Beweisen mit partiellen Zustandskonfigurationen folgt die Beweisstruktur demselben Prinzip, jedoch muss der Benutzer nach einem Ausführungsschritt entscheiden, welche Information über aktive Zustände, Ereignisse und Variablenbelegung der Nachfolgebelegung für den weiteren Beweis notwendig ist. Diese Information führt der Benutzer selbst ein und führt dann den Beweis mit Abwickeln der Temporaloperatoren fort. Die Beweistechniken für vollständige und partielle Zustandskonfigurationen haben wir in Abschnitt 6.3.2 demonstriert.

6.5.2 Beweisheuristiken

Werden Statecharts durch eine vollständige Zustandskonfiguration beschrieben, ist die Abfolge der Regelanwendungen kanonisch. Zuerst wird ein Statechart-Schritt abgewickelt, ausgeführt und dann vereinfacht. Anschließend wird die temporallogische Aussage abgewickelt und ein temporallogischer Schritt durchgeführt. Nun folgt wieder ein *sc unwind*-Schritt (vgl. Beweise in Abbildung 6.3). So werden nach und nach alle möglichen Pfade durch das Statechart durchlaufen. Zum Nachweis von Sicherheitseigenschaften beginnen wir vor jedem *sc unwind*-Schritt eine Induktion. Erreichen wir denselben Zustand wieder, können wir durch Anwenden der Induktionshypothese den Beweis schließen. Diese Abfolge von Befehlsanwendungen ist bei Statechart-Beweisen mit vollständiger Zustandskonfiguration leicht zu automatisieren.

Das KIV System benutzt sogenannte *Heuristiken*, um Beweise zu automatisieren. Sie wenden z. B. Regeln wie *simplify* automatisch an, um Formeln zu vereinfachen. Im Allgemeinen wendet eine Heuristik unter bestimmten Vorbedingungen eine Kalkülregel an. Für temporallogische Beweise gibt es bereits sehr mächtige Heuristiken, die auch Induktion über endliche und unendliche Intervalle automatisch durchführen. Um Beweise über Statecharts automatisieren zu können, haben wir die Heuristiken für temporallogische Beweise erweitert. Die Statechart-Heuristiken wenden zum Initialisieren des Statecharts die

sc initialize Regel an. Die Heuristiken wenden nun abwechselnd die *sc unwind*- und die *tl step*-Regel an und wickeln so nach und nach das Statechart ab. Nach der *sc unwind*-Regeln führen Heuristiken für DL-Programme die Aktionen der Statecharts automatisch aus. Die weiteren temporallogischen Heuristiken wickeln parallel dazu die temporalen Operatoren ab, versuchen in jedem Schritt den Beweis per Induktion zu schließen und beginnen, falls dies nicht gelingt, erneut eine Induktion. Da die *sc unwind*-Regel aus vollständigen Zustandskonfigurationen wiederum vollständige Nachfolgekonfigurationen berechnet, können wir den Beweis automatisch fortführen.

Diese beiden Heuristiken und die Heuristiken der Temporallogik genügen, um in KIV Beweise über Statecharts mit vollständiger Zustandskonfiguration automatisch zu führen. Die Heuristiken können das Beispiel der Zeitschaltuhr aus obigen Abschnitt ohne Benutzerinteraktion nachweisen. Natürlich wird dabei im Zustand *Close* die Variable x schrittweise bis 6 hochgezählt. Die Generalisierung auf $x \leq 6$ erfolgt nicht automatisch.

Bei Aussagen mit partiellen Zustandskonfigurationen ist diese Technik leider nicht anwendbar. Nach einem Statechart-Schritt ist unklar, welche Generalisierung für den weiteren Beweis notwendig ist, und welche nicht. Deshalb erfolgt bei Beweisen mit partiellen Zustandskonfigurationen die Statechart-Schrittausführung manuell, um danach eine Generalisierung der Zustandskonfiguration durchführen zu können. Die temporallogischen Heuristiken zur Anwendung der Induktion und Schrittabwicklung können auch für Beweise mit partiellen Zustandskonfigurationen verwendet werden. Entscheidet sich der Benutzer, den Beweis mit vollständigen Zustandskonfigurationen fortzuführen, so führt er nach der *sc unwind*-Regel eine Fallunterscheidung über alle möglichen Nachfolgekonfigurationen durch, erhält durch Anwendung der *sc wellformed*-Regel konsistente und vollständige Zustandskonfigurationen und kann nun den Beweis automatisiert weiterführen.

6.6 Zusammenfassung

In diesem Kapitel haben wir einen Ansatz zur interaktiven Verifikation von Statecharts vorgestellt. Unser Ziel ist der Nachweis von Eigenschaften über STATEMATE-Statecharts, die mit der Makro-Schritt-Semantik ausgeführt werden. Dazu unterstützen wir die wesentlichen Konstrukte der STATEMATE-Statecharts. Hervorheben möchten wir jedoch vier Besonderheiten unseres Ansatzes. Erstens führen wir Statecharts in Mikro-Schritten aus und können trotzdem Beweise über Makro-Schritte führen. Zweitens formulieren wir Statecharts über algebraischen Spezifikationen und können beliebige Datentypen in den Aktivierungsbedingungen und Aktionen der Statecharts verwenden. Drittens erreichen wir durch das Beweisprinzip der symbolischen Ausführung, dass Beweise dem Ablauf des Statecharts folgen und erhalten damit verständliche Beweise. Und schließlich erlaubt die Generalisierung von Statechart-Zustandsmengen allgemeinere Invarianten und führt zu kürzeren Beweisen.

STATEMATE-Statecharts besitzen viele Konstrukte und reichhaltige Spezifikationsvarianten, die nicht alle in KIV berücksichtigt sind. So werden syntaktische Vereinfachungen, wie *entry*- und *exit*-Aktionen in Zuständen und das Zusammenfassen von Übergängen durch sogenannte Konnektoren, nicht unterstützt. Dies schränkt jedoch den Sprachumfang

nicht wesentlich ein. Ebenfalls unterstützen wir Timeout-Ereignisse und Schedule-Aktionen nicht direkt. Beide Konstrukte können wir jedoch durch zusätzliche Zählervariablen simulieren, die in einem Makro-Schritt – durch das *tick*-Ereignis ausgelöst – hochgezählt werden (Abschnitt 6.2.2).

Einschränkungen im Sprachumfang gegenüber STATEMATE-Statecharts ergeben sich durch die fehlende Unterstützung von sog. *History*-Konnektoren. History-Konnektoren erlauben es, einen Zustand mit der Konfiguration von Unterzuständen zu betreten, mit der er verlassen wurde (und nicht mit der Konfiguration, die sich durch initiale Zustände ergibt). Die Nutzung von History-Konnektoren scheint nur für die Verwendung vollständiger Zustandskonfigurationen sinnvoll zu sein. Bei partiellen Zustandskonfigurationen ist unklar, welche früheren (partiellen) Konfigurationen von Bedeutung sind. Um History-Konnektoren verwirklichen zu können, müssen die aktiven Unterzustände eines Zustandes, der verlassen wird, gespeichert werden. Wenn wir dazu zusätzliche Zustandsvariablen einführen, ist es leicht möglich, History-Konnektoren für Statecharts in unseren Kalkül zu integrieren.

Von diesen Einschränkungen abgesehen, wurde ein Kalkül zur interaktiven Verifikation von STATEMATE-Statecharts entwickelt. Die Semantik entspricht der operationellen STATEMATE-Semantik von Statecharts. Parallele Übergänge werden semantikerhaltend sequentialisiert, damit die Schrittausführung von Statechart in sequentielle Programme übersetzt werden kann. Sequentielle Programme sind auch ausdrucksstark genug, die noch fehlenden STATEMATE-Konstrukte in KIV zu integrieren.

Grundlage unserer Statechart-Semantik ist die operationelle Semantik von Damm et al. [DJHP98]. Ein wesentlicher Unterschied unserer Semantik ist, dass wir Ereignisse und Variablen auch zu Mikro-Schritten beobachten können. Gerade für die interaktive Verifikation ist dies von großem Vorteil, um den Überblick über den Ablauf der Schritte und die Folge der Zustandswechsel zu behalten. So führt z. B. auch der STATEMATE-Simulator [HLN+90, Sim87] Statecharts in Mikro-Schritten aus. Trotzdem können wir Eigenschaften über Makro-Schritten nachweisen. Dazu benutzen wir das *tick*-Ereignis, das einen Makro-Schritt markiert. Weisen wir eine Eigenschaft φ eines Statecharts nach, die zu Makro-Schritten gilt, so zeigen wir $\Box(tick' \rightarrow \varphi(x'))$. $\varphi(x')$ formuliert die Eigenschaft $\varphi(x)$ über den 1-fach gestrichenen Variablen. Die Aussage $\Box(tick' \rightarrow \varphi(x'))$ spricht also über das Ausgabeverhalten eines Statecharts, d. h. über die Variablenbelegungen am Ende eines Makro-Schrittes. Andererseits formulieren wir Umgebungsannahmen $\psi(x)$, die nur in einem Makro-Schritt berücksichtigt werden, als $\Box(tick \rightarrow \psi(x))$. Umgebungsannahmen schränken das Umgebungsverhalten ein und werden als zusätzliche Voraussetzung in den Antezedenten einer Beweisverpflichtung mit aufgenommen.

Als zweites möchten wir hervorheben, dass durch die Verwendung algebraisch spezifizierter Datentypen, Funktionen und Prädikate, die in KIV-Statecharts verwendet werden, eine Beweisunterstützung für Statecharts entstanden ist, die nicht auf zustandsendliche Systeme beschränkt ist. Es können beliebige Funktionen, Prädikate und symbolische Konstanten spezifiziert und in den Aktivierungsbedingungen verwendet werden. So kann für die Zeitschaltuhr die Aktivierungsbedingung des Übergangs $t_3 :\ x > 5/sw_off; x := 0$ statt mit dem konkreten Wert 5 auch mit einer Variablen k beschriftet werden. Für dieses State-

chart können wir mit dem selben Beweisaufwand die Sicherheitseigenschaft $\Box x \leq k + 1$ nachweisen (statt der Generalisierung von $x = 0$ zu $x < 5$ generalisieren wir zu $x < k$, siehe Abschnitt 6.3.2, Seite 91). Die Verwendung von Variablen mit beliebigen Werten anstatt einer konkreten Belegung ist mit den bisherigen Ansätzen zur Statechart-Verifikation mit Modellprüfern nicht möglich. Beim automatischen Auswerten der Aktivierungsbedingungen helfen auch Entscheidungsprozeduren nicht weiter, wenn die Bedingungen über komplexen, algebraisch spezifizierten Prädikaten formuliert sind. In Abschnitt 10 behandeln wir eine Statechart-Fallstudie, in der Multiplikation und Division zur Auswertung der Aktivierungsbedingungen notwendig sind. STeP ist ein interaktiver temporallogischer Beweiser [BBC+00], der ebenfalls nicht auf einen endlichen Datenbereich eingeschränkt ist, er unterstützt jedoch keine Statecharts.

Zusätzlich zu den allgemeinen Aktivierungsbedingungen erhalten wir durch die Verwendung sequentieller Programme als Aktionen in Übergängen und statischen Reaktionen eine sehr mächtige Beschreibungssprache für Aktionen, die sogar über die Möglichkeiten von STATEMATE hinausgehen.

Ein wesentlicher Punkt in unserem Ansatz ist, dass Statecharts Formeln sind. Sie werden direkt durch die entwickelten Kalkülregeln unterstützt und nicht in einen anderen Formalismus kodiert. In STeP und TLA (Temporal Logic of Actions, [Lam94]) werden parallele Programme in flach Zustandsübergangssysteme bzw. temporallogische Formeln übersetzt, wodurch der Kontrollfluss der Programme verloren geht. Dies vermeiden wir und erhalten in den Beweisen die Struktur des Statecharts. Durch die schrittweise Abarbeitung sind die Beweise leicht verständlich, denn sie folgen immer dem Ablauf des Statecharts.

Schließlich ermöglicht es die Generalisierung von Statecharts auf partielle Zustandskonfigurationen, von gewissen Eigenschaften der Statecharts zu abstrahieren. Das Konzept partieller Zustandskonfiguration und die korrekte Berechnung möglicher Nachfolgebelegungen ist bisher in der Literatur noch nicht bekannt. Für interaktive Statechart-Beweise kann auf partielle Zustandskonfigurationen nicht verzichtet werden. Wir erhalten dadurch allgemeinere Invarianten für die temporallogische Verifikation und können kürzere Beweise führen. Außerdem können allgemeinere Aussagen bewiesen werden, die in anderen Beweisen als Lemmata verwendet werden.

In KIV haben wir Kalkülregeln für Beweise mit vollständigen und partiellen Zustandskonfigurationen implementiert und die Anwendbarkeit des Kalküls mit verschiedenen Statechart-Beweisen gezeigt. Zur Demonstration präsentieren wir in Kapitel 10 die Fallstudie des funkbasierten Bahnübergangs, den wir in KIV mit Statecharts spezifiziert, einen Fehlerbaum erstellt und dessen Vollständigkeit nachgewiesen haben.

KAPITEL 7

Modellprüfung von Fehlerbäumen

Für den Einsatz der formalen FTA im industriellen Umfeld ist es erstrebenswert, den Nachweis der Beweisverpflichtungen zu automatisieren. Der Ingenieur kann sich dann weiterhin vollständig auf die sicherheitstechnische Analyse des Systems konzentrieren und erhält zusätzlich die formale Validation seiner Analyse.

Eine bekannte Methode für den automatischen Nachweis von Systemeigenschaften ist die Modellprüfung. Sie prüft, ob ein Systemmodell *Sys* eine geforderte Eigenschaft φ erfüllt, d. h. ob $Sys \models \varphi$. Dabei ist das Systemmodell *Sys* meist als Zustandsübergangssystem und die nachzuweisende Eigenschaft φ in einer Temporallogik beschrieben. Die *Computation Tree Logic* (CTL, [CE81]) ist eine Temporallogik, für die es effiziente Algorithmen zur Modellprüfung gibt [CES86]. Verschiedene Modellprüfer basieren auf diesen Algorithmen (z. B. RAVEN [Ruf00], SMV [McM90] und SVE [FSS$^+$94]) und werden im industriellen Umfeld eingesetzt. Deshalb geben wir in diesem Kapitel Vollständigkeitsbedingungen für die FTA in CTL an und ermöglichen damit eine effiziente Modellprüfung der formalen FTA.

Die Vollständigkeitsbedingungen in CTL sind ebenfalls an die Fehlerbaum-Gatter geknüpft und sollen der ITL-Semantik aus Kapitel 5 entsprechen. Die Logiken CTL und ITL besitzen jedoch eine unterschiedliche Ausdrucksstärke, und so gelingt dies nur für Fehlerbäume mit punktuellen Ereignissen (Ereignisse ohne Dauer) direkt in CTL. Für allgemeine Fehlerbäume mit temporalen Ereignissen geben wir eine Konstruktion mit sogenannten Akzeptor-Automaten an. Ein Aktzeptor-Automat beobachtet das temporale Ereignis, „akzeptiert“ es in einem Endzustand und bildet es so auf ein punktuelles Ereignis (das Erreichen des Endzustands) ab. Mit dieser Konstruktion gelingt es dann auch, allgemeine Fehlerbäume mit CTL-Modellprüfern auf Vollständigkeit zu prüfen. In Kapitel 8 zeigen wir formal, dass die CTL-Bedingungen der ITL-Semantik entsprechen.

Unser Vorgehen demonstrieren wie am Beispiel des Drucktanks aus Abschnitt 3.1. Die Vollständigkeit des Fehlerbaums für diese Fallstudie haben wir in RAVEN verifiziert. RAVEN benutzt zeitbehaftete Automaten, sogenannte E/A-Intervallautomaten zur Spezifikation des Modells und eine Erweiterung der CTL, die *clocked* CTL, die auch für CTL-Operatoren Zeitannotationen erlaubt. Nachdem wir die Semantik der E/A-Intervallauto-

maten und der CCTL präsentiert haben, definieren wir die Vollständigkeitsbedingungen für die FTA. Diese benötigen keine Zeitannotationen und können deshalb auch in reiner CTL definiert werden. Mit den geschaffenen Voraussetzungen bearbeiten wir das Beispiel des Drucktanks. Wir spezifizieren den Drucktank mit E/A-Intervallautomaten, formalisieren dann den informellen Fehlerbaum aus Abschnitt 3.1.1 und zeigen die Vollständigkeitsbedingungen, die sich aus den CTL-Bedingungen der einzelnen Gatter und den formalisierten Fehlerbaumereignissen ergeben. Schließlich präsentieren wir verwandte Arbeiten und fassen die erarbeiteten Resultate zusammen.

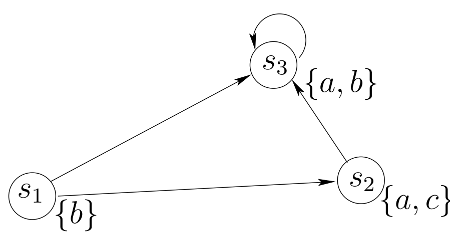
7.1 Clocked Computation Tree Logic (CCTL)

Die Semantik der CCTL basiert auf einer Abfolge von Belegungen, die wir im Folgenden durch Zustandsübergangssysteme beschreiben. Zur Modellierung reaktiver, zeitbehafteter Systeme benutzen wir Ein-/Ausgabe-Intervallautomaten (E/A-Intervallautomaten, [RK99]).

7.1.1 E/A-Intervallautomaten

Einfache Zustandsübergangssysteme bestehen aus Zuständen, einer Übergangsrelation zwischen den Zuständen und einer Beschriftung der Zustände mit booleschen Variablen. Die Übergangsrelation beschreibt, welche Zustandsfolgen erzeugt werden können und die Beschriftung der Zustände, welche Variablen in welchem Zustand wahr sind. Das Zustandsübergangssystem beschreibt dann ein Systemverhalten als Folge von Variablenbelegungen.

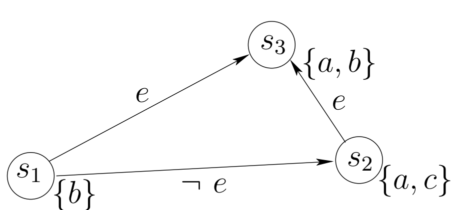
Beispiel 7.1 *Zustandsübergangssystem*



Dieses Zustandsübergangssystem hat 3 Zustände. In s_1 ist die aussagenlogische Variable b wahr (a und c aber nicht), in s_2 ist a und c wahr und in s_3 ist a und b wahr. Der Nachfolgezustand von s_1 ist s_2 oder s_3, von s_2 ist es s_3 und s_3 hat als einzigen Nachfolger sich selbst. Ein mögliches Systemverhalten ist $(\{b\}, \{a, c\}, \{a, b\}, \{a, b\}, \ldots)$.

Um Reaktionen auf andere Systemkomponenten oder die Umgebung zu modellieren, knüpft man Zustandsübergänge an Bedingungen. Sobald bestimmte Eingaben vorliegen, wird der Zustandsübergang ausgeführt. Man erhält endliche Automaten.

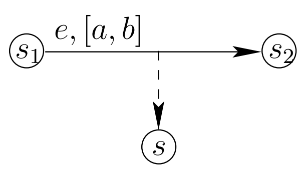
Beispiel 7.2 *endlicher Automat*



Sei s_1 der Startzustand. Wenn die Bedingung e gilt, geht der Automat im ersten Schritt in den Zustand s_3 über, ansonsten in den Zustand s_2. Dort verbleibt der Automat, bis e eintritt und geht dann in s_3. Ein mögliches Systemverhalten ist hier ($\{b\} \rightarrow \{a,c\} \xrightarrow{e} \{a,b\} \ldots$).

In zeitbehafteten Systemen möchte man zusätzlich den Übergang an zeitliche Bedingungen knüpfen. Eingaben sollen eine bestimmte Zeit vorhanden sein, bevor ein Zustandsübergang ausgeführt wird. Solche Modelle beschreiben wir mit E/A-Intervallautomaten, die zusätzlich zu den Eingabevariablen e ein Intervall $[a,b]$ mit $a \in \mathbb{N}, b \in \mathbb{N}_\infty, a \leq b$, als Beschriftung eines Übergangs erlauben. Ein Übergang wird indeterministisch nach t Zeitschritten ausgeführt ($a \leq t \leq b$), wenn das Ereignis e t Schritte gilt. Damit das System auch reagieren kann, wenn das Ereignis e keine t Zeitschritte vorhanden ist, erhält der Übergang eine (gestrichelte) Zusatzkante in den Zustand, der in diesem Fall eingenommen wird.

Definition 7.1 *E/A-Intervallautomaten, graphische Notation*



Wird das Ereignis e für t, $a \leq t \leq b$, Zeitschritte beobachtet, wird der Zustand s_2 eingenommen. Wird e für $0 < t < a$ Zeitschritte beobachtet, gelangen wir in den Zustand s. Falls e überhaupt nicht gilt, wird in s_1 verblieben (d. h. der Zustand s_0 besitzt einen impliziten Übergang nach s_1, der mit $\neg\, e$ beschriftet ist).

Sind bei einem Übergang $e,[a,b]$ die Intervallgrenzen $a = b$ schreibt man $e,[a]$ und ist $a = b = 1$ nur e. Im letzten Fall kann s nicht erreicht werden und die Zusatzkante ist nicht erforderlich. Der Übergang erhält die Bedeutung wie in endliche Automaten. Ist zusätzlich $e = \text{true}$ kann die Kantenbeschriftung weggelassen werden und wir erhalten die Semantik herkömmlicher Zustandsübergangssysteme.

Definition 7.2 *Semantik von E/A-Intervallautomaten*

Eine E/A-Intervallstruktur ist ein Tupel $\Im = (\mathcal{P}, \mathcal{E}, \mathcal{S}, \mathcal{I}, \mathcal{T}, \mathcal{L})$ mit:

- *$\mathcal{P}$: einer endlichen Menge boolescher Variablen.*
- *$\mathcal{E}$: einer endlichen Menge boolescher Eingabevariablen.*
- *$\mathcal{S}$: einer endlichen Menge von Zuständen.*
- *$\mathcal{I}$: einer endlichen Menge von Ausgangszuständen $\mathcal{I} \subseteq \mathcal{S}$.*
- *$\mathcal{T}$: einer Übergangsrelation $\mathcal{T} \subset \mathcal{S} \times \wp(\mathcal{E}) \times I\!N \times I\!N_\infty \times \mathcal{S} \times S$.*

  Die Übergangsrelation beschreibt den Ausgangszustand, das aktivierende Ereignis als maximal einelementige Menge, die untere und obere Zeitschranke für den Übergang,

den Endzustand und den zusätzlichen Ausstiegszustand. Für $t = (s_1, E, a, b, s_2, s)$ *sei* $t[i]$, $1 \leq i \leq 6$, *das i-te Element, d. h.* $t[1] = s_1$, $t[2] = E$, ... *Wir verlangen: für alle* $(s_1, E, a, b, s_2, s) \in \mathcal{T}$ *gilt* $|E| \leq 1$, $1 \leq a \leq b$ *und wenn* $t_1, t_2 \in T$ *mit* $t_1[1] = t_2[1]$, $t_1[4] > 1$ *und* $t_2[4] > 1$, *dann* $t_1 = t_2$. *D. h. es darf maximal ein zeitbehafteter Übergang für einen Ausgangszustand existieren.*

- $\mathcal{L}$: *einer Beschriftungsfunktion* $\mathcal{L} : \mathcal{S} \to \wp(\mathcal{P})$.

  $\mathcal{L}$ *weist jedem Zustand s eine Menge boolescher Variablen aus* $\mathcal{P}$ *zu, die in s zu* true *evaluieren.*

Wir betrachten hier eine einfache Variante der E/A-Intervallstruktur, bei der die Kanten mit maximal einer Eingabevariablen beschriftet werden kann. Für einen unbeschrifteten Übergang t ist $t[2] = \emptyset$. Diese Variante genügt für die Modellierung des Beispiels und vereinfacht die Definition der E/A-Intervallstrukturen. In [RK99] sind E/A-Intervallstrukturen definiert, die auch boolesche Ausdrücke über Eingabevariablen als Beschriftung von Übergängen erlauben. Die Einschränkung auf maximal einen zeitbehafteten Übergang mit ein und demselben Ausgangszustand besteht auch dort.

Für die Definition möglicher Pfade benötigen wir eine erweiterte Zustandskonfiguration, die für jeden Zustand eine Uhr und die aktuellen Eingaben enthält. Diese Uhr beschreibt, wie lange der Zustand aktiv ist. Die maximal aktive Zeit eines Zustandes hängt von den aktuellen Eingaben ab.

Definition 7.3 *erweiterte Zustandskonfiguration*

Für einen Zustand s einer E/A-Intervallstruktur $\Im = (\mathcal{P}, \mathcal{E}, \mathcal{S}, \mathcal{I}, \mathcal{T}, \mathcal{L})$ *ist eine erweiterte Zustandskonfiguration* $g = (s, i, n)$ *mit* $s \in \mathcal{S}$, $i \in \wp(\mathcal{E})$ *und* $0 \leq n < max\{b | \exists\, t \in \mathcal{T}, t[1] = s, t[4] = b\}$. *Die Menge aller möglichen erweiterten Zustandskonfigurationen für* $\Im$ *ist*

$$G_\Im = \{(s, i, n) \mid s \in \mathcal{S}, i \in \wp(\mathcal{E}) \text{ und } 0 \leq n < max\{b | \exists\, t \in \mathcal{T}, t[1] = s, b = t[4]\}\}.$$

Wir definieren eine Pfad über erweiterten Zustandskonfigurationen.

Definition 7.4 *Pfad*

Sei $\Im = (\mathcal{P}, \mathcal{E}, \mathcal{S}, \mathcal{I}, \mathcal{T}, \mathcal{L})$ *eine E/A-Intervallstruktur. Ein Pfad ist eine Folge erweiterter Zustandskonfigurationen* $\sigma = (g_0, g_1, \ldots)$ *mit* $g_j \in G_\Im$ *und für alle* $g_j = (s_j, i_j, n_j)$ *gilt entweder*

1. $g_{j+1} = (s_j, i_{j+1}, n_j + 1)$ *und es existierte ein* $(s_j, E, a, b, s_{j+1}, s) \in \mathcal{T}$ *mit:*
   $E \subseteq i_j$ *und* $n_j + 1 < b$,

2. $g_{j+1} = (s_{j+1}, i_{j+1}, 0)$ *und es existierte ein* $(s_j, E, a, b, s_{j+1}, s) \in \mathcal{T}$ *mit:*
   $E \subseteq i_j$ *und* $a \leq n_j + 1 \leq b$,

3. $g_{j+1} = (s, i_{j+1}, 0)$ *und es existierte ein* $(s_j, E, a, b, s_{j+1}, s) \in \mathcal{T}$ *mit:*
   $b > 1$, $n_j > 0$ *und* $E \not\subseteq i_j$ *oder*

4. $g_{j+1} = (s_j, i_{j+1}, 0)$*:*
wenn $n_j = 0$ *und für alle* $t \in \mathcal{T}$ *entweder* $s_j \neq t[1]$ *oder* $t[2] \not\subseteq i_{j+1}$.

Im ersten Fall führen wir einen Zeitschritt aus, d. h. der Zustand wird nicht verlassen, die interne Uhr aber um eins erhöht. Im zweiten Fall wird der Endzustand eines (zeitbehafteten) Übergangs betreten und dabei die Uhr zurückgesetzt. Betrachten wir einen nicht zeitbehafteten Übergang ($a = b = 1$), führt der zweite Fall auch diesen Übergang aus. Der dritte Fall beschreibt die Situation, in der ein zeitbehafteter Übergang ($b > 0$) betreten wurde ($n_j > 0$), nun aber das Eingabesignal nicht mehr anliegt (der Übergang t mit $t[1] = s_j$ und $t[4] > 1$ ist der einzige zeitbehaftete Übergang aus s_j). Dann wird der Ausstiegszustand betreten. Im vierten Fall befinden wir uns in einem Zustand mit Uhrenwert 0, und kein Übergang hat eine Aktivierungsbedingung, die von der aktuellen Eingabe erfüllt wird. Dann verbleiben wir in diesem Zustand.

Komposition Für größere Modelle ist es wichtig, das Systemmodell modular zu erstellen. Die einzelnen Systemkomponenten werden in verschiedenen Modulen durch jeweils einen endlichen Automaten beschrieben. Die verschiedenen Module bzw. deren Automaten kommunizieren dann über die Eingabevariablen. Das Verhalten des Gesamtsystems ergibt sich aus der Parallelschaltung der einzelnen Module (Komposition). Die Semantik der Komposition ist, vereinfacht, die Pfadinklusion. Jeder Pfad, der im komponierten System möglich ist, muss auch in einem einzelnen Modul möglich sein (wenn der Pfad auf die atomaren Formeln des Moduls projiziert wird). Für Details verweisen wir auf [Ruf99].

7.1.2 Semantik von CCTL

Die CCTL ist eine Erweiterung der CTL um Zeitannotationen [RK97]. Sie erlaubt Aussagen über Zeitspannen einfacher zu formulieren und wird in dem Modellprüfer RAVEN verwendet. In CCTL werden temporale Operatoren mit Intervallen $[a, b]$ beschriftet. Die Intervalle kennzeichnen ausgehend vom aktuellen Zustand Zeitpunkte a und b, zwischen denen die temporalen Operatoren gelten sollen.

Definition 7.5 *temporale CCTL-Operatoren*

$$\begin{aligned}
X_{[a]}\varphi \quad &:\Leftrightarrow \quad \textit{in } a \textit{ Zeitschritten gilt } \varphi \\
F_{[a,b]\varphi} \quad &:\Leftrightarrow \quad \varphi \textit{ gilt irgendwann zwischen } a \textit{ und } b \\
G_{[a,b]}\varphi \quad &:\Leftrightarrow \quad \varphi \textit{ gilt immer zwischen } a \textit{ und } b \\
\varphi \mathrm{U}_{[a,b]}\psi \quad &:\Leftrightarrow \quad \psi \textit{ gilt innerhalb } [a,b] \textit{ und zuvor gilt immer } \varphi
\end{aligned}$$

Für den Next-Operator gilt $X\varphi := X_{[1]}\varphi$. Ist in einem Intervall $a = 0$ schreibt man statt $[a, b]$ nur $[b]$. Ist $a = 0$ und $b = \infty$ wird der Intervallausdruck weggelassen. Wir definieren die Semantik von CCTL-Formeln über den Pfaden von E/A-Intervallautomaten.

Definition 7.6 *Semantik von CCTL*

Sei $\Im = (\mathcal{P}, \mathcal{E}, \mathcal{S}, \mathcal{I}, \mathcal{T}, \mathcal{L})$, a, $b \in I\!N_\infty$, $g_0 \in G_\Im$ *und* φ *und* ψ *CCTL-Formeln. Wir schreiben* $g_0 \models \varphi$ *für* $\Im, g_0 \models \varphi$. *Für eine atomare Formeln* $a \in \mathcal{P}$ *und* $g_0 = (s, i, n)$ *gilt* $g_0 \models a :\Leftrightarrow a \in \mathcal{L}(s)$. *Die Operatoren* $\wedge$, $\vee$, $\neg$, ... *sind wie üblich definiert.*

1. *für die existenzquantifizierten Operatoren gilt:*

$$
\begin{array}{lll}
g_0 \models EX_{[a]}\ \varphi & :\Leftrightarrow & \text{es existiert ein Pfad } (g_0, g_1, \ldots) \text{ mit} \\
 & & g_a \models \varphi \\
g_0 \models EG_{[a,b]}\ \varphi & :\Leftrightarrow & \text{es existierte ein Pfad } (g_0, g_1, \ldots), \\
 & & \text{so dass für alle } k,\ a \leq k \leq b, \text{ gilt } g_k \models \varphi \\
g_0 \models EF_{[a,b]}\ \varphi & :\Leftrightarrow & \text{es existierte ein Pfad } (g_0, g_1, \ldots) \\
 & & \text{und ein } k,\ a \leq k \leq b, \text{ mit } g_k \models \varphi \\
g_0 \models E(\varphi \mathrm{U}_{[a,b]} \psi) & :\Leftrightarrow & \text{es existiert ein Pfad } (g_0, g_1, \ldots) \text{ und} \\
 & & \text{ein } k,\ a \leq k \leq b, \text{ so dass } g_k \models \psi, \\
 & & \text{und für alle } i < k \text{ gilt } g_i \models \varphi
\end{array}
$$

2. *für die allquantifizierten Operatoren gilt:*

$$
\begin{array}{lll}
g_0 \models AX_{[a]}\ \varphi & :\Leftrightarrow & \text{für alle Pfade } (g_0, g_1, \ldots) \text{ gilt} \\
 & & g_a \models \varphi \\
g_0 \models AG_{[a,b]}\ \varphi & :\Leftrightarrow & \text{für alle Pfade } (g_0, g_1, \ldots) \\
 & & \text{und } k,\ a \leq k \leq b, \text{ gilt } g_k \models \varphi \\
g_0 \models AF_{[a,b]}\ \varphi & :\Leftrightarrow & \text{für alle Pfade } (g_0, g_1, \ldots) \\
 & & \text{existiert ein } k,\ a \leq k \leq b, \text{ mit } g_k \models \varphi \\
g_0 \models A(\varphi \mathrm{U}_{[a,b]} \psi) & :\Leftrightarrow & \text{für alle Pfade } (g_0, g_1, \ldots) \text{ gibt es} \\
 & & \text{ein } k,\ a \leq k \leq b, \text{ so dass } g_k \models \psi, \\
 & & \text{und für alle } i < k \text{ gilt } g_i \models \varphi
\end{array}
$$

Sind die Zeitannotationen $a = 0$ und $b = \infty$, so erhalten die CCTL-Operatoren die Semantik herkömmlicher CTL-Operatoren.

Ein System, das durch eine E/A-Intervallstruktur $\Im = (\mathcal{P}, \mathcal{E}, \mathcal{S}, \mathcal{I}, \mathcal{T}, \mathcal{L})$ beschrieben wird, erfüllt eine (C)CTL-Formel φ, d. h. es gilt $\Im \models \varphi$, genau dann, wenn für alle Ausgangszustände $g \in \{(s, i, 0) \mid (s, i, 0) \in G_\Im,\ s \in \mathcal{I} \text{ und } i \in \wp(\mathcal{E})\}$ der E/A-Intervallstruktur $\Im, g \models \varphi$ gilt. Eine (C)CTL-Formel φ ist gültig, wenn für alle E/A-Intervallstruktur $\Im$ gilt: $\Im \models \varphi$.

Wir haben hier die Semantik von CCTL angegeben, da der Modellprüfer RAVEN CCTL verwendet und wir die Fallstudie in RAVEN modelliert und verifiziert haben. Für die Formalisierung der Vollständigkeitsbedingungen für die Fehlerbaumgatter können wir auf die Zeitannotationen verzichten und sie in reiner CTL formalisieren (CCTL ohne Zeitannotationen). Zusätzlich definieren wir für die Vollständigkeitsprüfung der FTA den temporalen Operator P (*precedes*).

Definition 7.7 *precedes*

$$g_0 \models A(\varphi \mathrm{P}_{[a,b]} \psi) \quad :\Leftrightarrow \quad \begin{array}{l} \textit{für alle Pfade } (g_0, g_1, \ldots): \\ \quad \textit{für jedes } k,\ a \leq k \leq b,\ \textit{mit } g_k \models \psi \\ \quad \textit{existiert ein } i \leq k \textit{ mit } g_i \models \varphi \end{array}$$

Informell bedeutet $A(\varphi \mathrm{P}_{[a,b]} \psi)$ für alle Pfade, wenn ψ zu einem Zeitpunkt zwischen a und b gilt, dann muss φ vor oder gleichzeitig mit ψ gelten. Wenn ψ auf einem Pfad nicht gilt, muss auch φ nicht gelten. Wir benötigen im Folgenden die Form $A(\varphi \mathrm{P}_{[0,\infty]} \psi)$ mit unterer Intervallgrenze 0 und oberer Intervallgrenze ∞, die wir auch in reiner CTL formalisieren können. $A(\varphi \text{ P } \psi)$ ist äquivalent zu $\neg E(\neg\varphi \text{ U } (\psi \wedge \neg\varphi))$ und kann deshalb in jedem CTL Modellprüfer – und erst recht im CCTL-Modellprüfer RAVEN – spezifiziert werden.

7.2 Vollständigkeitsprüfung in (C)CTL

Auf den Grundlagen des vorigen Abschnitts definieren wir die Vollständigkeitsbedingungen der FTA. Für die Bedingungen werden keine Zeitannotationen benötigt und wir formulieren sie in CTL (als Teilsprache von CCTL), so dass sie auch CTL-Modellprüfer nachwgewiesen werden können. Analog zur ITL-Semantik sind die Bedingungen an die einzelnen Fehlerbaumgatter gebunden und die Vollständigkeit der FTA folgt aus dem Nachweis aller Gatterbedingungen.

7.2.1 Formalisierung der Fehlerbaumgatter

Wir definieren nun die Vollständigkeitsbedingungen für die 7 Fehlerbaumgatter aus Abschnitt 5.1. Die Formalisierung beschreibt Gatter, die eine Wirkung auf zwei Unterereignisse zurückführen. Die Verallgemeinerung auf $n \geq 1$ Unterereignisse ist offensichtlich. Im Folgenden bezeichnen φ_1 und φ_2 formalisierte Ursachen und ψ eine formalisierte Wirkung.

Zerlegungsgatter Die Zerlegungsgatter beschreiben informell: immer wenn die Wirkung eintritt, müssen auch die Ursachen vorliegen. Diesen Sachverhalt formalisieren wir durch die Implikation.

Definition 7.8 *Vollständigkeit der Zerlegungsgatter*

- Z-Und-*Gatter:*

  ψ / z / φ_1 φ_2 AG $(\psi \rightarrow \varphi_1 \wedge \varphi_2)$

- Z-Oder-*Gatter:*

  ψ / z / φ_1 φ_2 AG $(\psi \rightarrow \varphi_1 \vee \varphi_2)$

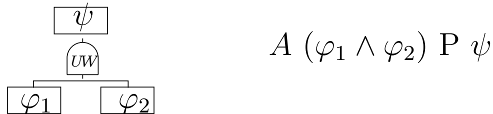
- Z-Block-*Gatter:*

Die Vollständigkeitsbedingungen der Zerlegungsgatter garantieren, dass nie eine Wirkung ohne den entsprechenden Ursachen eintritt. Beim *Und*-Gatter müssen beide, beim *Oder*-Gatter nur eine der Ursachen vorliegen. Dies muss natürlich für alle Zustände im System gelten. Deshalb ist der Implikation ein *AG* vorgestellt. Bei einem *Z-Block*-Gatter ist die Nebenbedingung χ kein Fehler, muss aber ebenso eintreten, wie die einzige Ursache ψ.

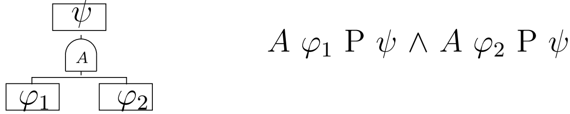
Ursache/Wirkungs-Gatter Die Ursache/Wirkungs-Gatter beschreiben eine Situation, in der die Ursachen φ_1 bzw. φ_2 vor der Wirkung ψ eintreten. Für die Definition der Vollständigkeit benutzen wir den P-Operator.

Definition 7.9 *Vollständigkeit der Ursache/Wirkungs-Gatter*

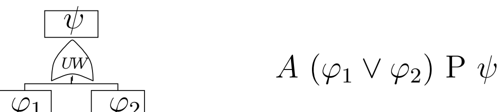
- *synchrones* UW-Und-*Gatter:*



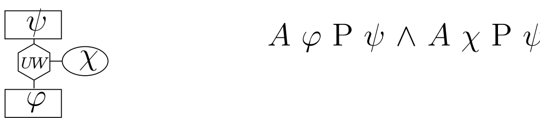
- *asynchrones* UW-Und-*Gatter:*



- UW-Oder-*Gatter:*



- UW-Block-*Gatter:*



Die Vollständigkeitsbedingungen der Ursache/Wirkungs-Gatter garantieren: wenn die Wirkung eintritt, musste(n) zuvor eine/alle Ursache(n) eintreten. Dies gilt für alle Pfade im System. Analog zur ITL-Semantik ist auch erlaubt, dass Ursachen und Wirkung gleichzeitig auftreten können (siehe Definition 7.7 von P). Beim synchronen *UW-Und*-Gatter müssen die Ursachen zusammen auftreten. Dies wird beim asynchronen *UW-Und*-Gatter nicht verlangt. Beim *UW-Block*-Gatter muss, wie beim asynchronen *UW-Und*-Gatter, die Ursache φ und die Nebenbedingung χ auftreten.

Punktuelle vs. temporale Ereignisse

Für punktuelle Ereignisse beschreibt die Formalisierung der Vollständigkeit eines Gatters das gewünschte Verhalten. Sie verlangt, dass die Ursachen vor oder gleichzeitig mit der Wirkung eintreten.

Betrachtet man temporale Ursachen (Ursachen mit Dauern, die in Temporallogik formalisiert werden), beschreibt die obige Formalisierung für Ursache/Wirkungs-Gatter nicht immer das gewünschte Verhalten. Temporale Ereignisse müssen als CTL-Formel beschrieben werden, damit wir nach der Instantiierung der Gatterbedingungen wieder CTL-Formeln für die Modellprüfung erhalten. Betrachten wir uns dazu die Formalisierung der Vollständigkeit $A\ \varphi\ \mathrm{P}\ \psi$ für eine einzige Ursache φ (dann fällt die Formalisierung der *Und*- bzw. *Oder*-Gatter zusammen) und der Wirkung ψ. Ist φ eine temporale Formel, muss sie in CTL durch $\varphi := QO\varphi'$ beschrieben werden, wobei $Q \in \{A, E\}$ ein Pfadquantor und $O \in \{F,\ G,$ $\mathrm{U},\ \mathrm{P}\}$ ein linearer Temporaloperator ist. $A\ \varphi\ \mathrm{P}\ \psi$ bedeutet dann: wenn es einen Zustand g_i mit $g_i \models \psi$ gibt, dann gibt es einen Zustand $g_j, j \leq i$ mit $g_j \models QO\ \varphi'$. D. h. ausgehend vom Zustand g_j muss für alle/einen Pfad $O\ \varphi'$ gelten. Wählt man den Pfadquantor A, muss die Wirkung für alle g_j-Pfade gelten. Diese Forderung ist aber zu stark, denn eine Ursache muss nur für den Pfad vorliegen, auf dem die Wirkung folgt. Im umgekehrten Fall, wenn der Pfadquantor E gewählt wird, genügt, dass ein g_j-Pfad existiert, in dem die Ursache gilt. Dieser muss aber nicht der Pfad sein, auf dem die Wirkung folgt. Abbildung 7.1 verdeutlicht diesen Sachverhalt. Links ist der A-Quantor dargestellt, der auch für den rechten

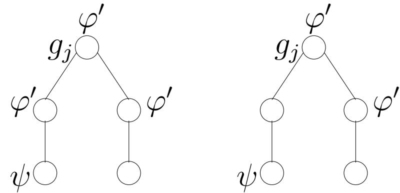


Abbildung 7.1: $AG_{[2]}\ \varphi'$ bzw. $EG_{[2]}\ \varphi'$

Pfad verlangt, dass die Ursache φ' in den nächsten beiden Zuständen gilt. Der E-Quantor (rechts) ist zu schwach, da er nicht notwendigerweise für den Pfad, auf dem die Wirkung folgt (im Beispiel der linke Pfad), die Ursache verlangt.

Die CTL-Semantik beschreibt deshalb nur für punktuelle Ereignisse die Bedeutung der Ursache/Wirkungs-Gatter adäquat. Dort muss die Ursache φ in g_j gelten und g_j liegt vor g_i. Um auch temporale Ereignisse korrekt zu behandeln, führen wir im nächsten Abschnitt sogenannte *Akzeptor-Automaten* ein, mit deren Hilfe temporale Ereignisse auf punktuelle Ereignisse zurückgeführt werden.

7.2.2 Akzeptor-Automaten

Wie in Abschnitt 5.1 erwähnt, treten temporale Ereignisse, wie z. B. "10 Sekunden bremsen", in einer FTA üblicherweise auf. Wir wiederholen hier nochmals die von Górski [GW95]

identifizierten, typischen Ereignismuster.

1. ein Ereignis tritt (für eine bestimmte Zeit) ein, z. B. „(10 Sekunden) bremsen“
2. ein Ereignis tritt dauerhaft ein, z. B. „Kollision“, sie kann nicht mehr rückgängig gemacht werden
3. eine Sequenz von Ereignissen (mit Zeitbedingungen) wird durchlaufen, z. B. „Ampel ist rot, dann gelb und schließlich grün“

Jedes dieser Muster kann einen zeitlichen Verlauf beschreiben. Wie im vorigen Abschnitt erläutert, beschreiben die CTL-Vollständigkeitsbedingungen die Bedeutung der Ursache/-Wirkungs-Gatter für temporale Ereignisse nicht adäquat, wenn die temporale Ursache in CTL beschrieben wird. Der Formalisierung von der Ursache „10 Sekunden bremsen“ vor einer bestimmten Wirkung gelingt in reiner CTL nicht, da man einem linearen Temporaloperator $G_{[10]}brake$ immer einen Pfadquantor voranstellen muss. Deshalb formalisieren wir die Ereignisse mit den quantifizierenden Operatoren aus Abschnitt 5.2 und betten diese ohne Pfadquantoren in CTL ein.

1. irgendwann gilt Ereignis φ c Schritte:
   $\diamondsuit \lceil\lceil \varphi \rceil \wedge l = c$
2. wenn φ gilt, dann für immer (und irgendwann gilt φ):
   $\boxtimes \neg\ (\lceil \varphi \rceil^0 \ ;\ \text{true}\ ;\ \lceil \neg\ \varphi \rceil^0) \wedge \diamondsuit \lceil \varphi \rceil^0$
3. Sequenz von Ereignissen $\varphi_1 \ldots \varphi_n$ mit Dauern $c_1 \ldots c_n$:
   $(\diamondsuit \lceil\lceil \varphi_1 \rceil \wedge l = c_1) \ ;\ \ldots\ ;\ (\diamondsuit \lceil\lceil \varphi_n \rceil \wedge l = c_n)$

Pandya hat in [Pan01a] einen Ansatz beschrieben, der CTL* und ihre Teilsprache CTL um die Möglichkeit erweitert, temporale Eigenschaften mit linear temporallogischen Formeln [Pan01b] zu beschreiben. Diese Formeln entsprechen den ITL-Formeln mit quantifizierenden Operatoren aus Abschnitt 5.2 über endlichen Intervallen (ITL-Formeln wurden über endlichen und unendlichen Intervallen definiert), die Pandya *Quantified Discrete-Time Duration Calculus* Formeln (QDDC-Formeln) nennt. Wir bezeichnen die Menge aller QDDC-Formeln mit Fma_{QDDC}. In einem Zustand g gilt eine QDDC-Formel $\varphi \in Fma_{QDDC}$, wenn auf dem Pfad vom Ausgangszustand des Systems zu g φ gilt (wir geben die Einbettung von QDDC-Formeln in CTL nach Pandya in Abschnitt 8.4.2 formal an). Für die um QDDC-Formeln erweiterte CTL können die resultierenden Formeln mit gewöhnlichen CTL-Modellprüfern automatisch nachgewiesen werden. Dazu wird für die eingebettete QDDC-Formeln ein sogenannter Akzeptor-Automate konstruiert, der parallel zum Systemmodel läuft. Nun wird geprüft, ob ein Endzustand im Akzeptor-Automaten erreicht wird. Der Endzustand eines Akzeptor-Automaten kennzeichnet, dass die QDDC-Formeln erfüllt ist, die der Automat repräsentiert. Das Erreichen des Endzustandes eines Akzeptor-Automaten ist ein punktuelles Ereignis, für das die CTL-Bedingungen die Bedeutung der Ursache/Wirkungs-Gatter adäquat beschreiben.

Mit dieser Einbettung können wir QDDC-Formeln auch in Fehlerbäumen erlauben, die mit CTL-Modellprüfung validiert werden. $CTL(\varphi)$ beschreibt, dass eine QDDC-Formel $\varphi \in Fma_{QDDC}$ in CTL eingebettet ist.

Lemma 7.1
Für eine Formel $\varphi \in \mathrm{Fma}_{QDDC}$, die atomar in einer CTL-Formel vorkommt, gilt:

$$\Im \models CTL(\varphi) \Leftrightarrow \Im \,||\, a_\varphi \models CTL(\varphi'),$$

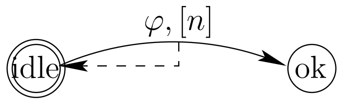
wobei a_φ der Akzeptor-Automat für die QDDC-Formel φ ist und φ' eine boolscher Ausdruck über Variablen, der das Erreichen der Endzustände des Akzeptor-Automaten beschreibt.

Für die Korrektheit dieses Lemmas und die Konstruktion der Akzeptor-Automaten verweisen wir auf [Pan01a]. Im Folgenden geben wir die Akzeptor-Automaten a_φ und die akzeptierende Formel φ' für die typischen Ereignistypen der Fehlerbäume nach Górski an.

Ereignis mit bestimmter Dauer

Soll beim ersten Ereignismuster von Górski ein Ereignis für eine bestimmte Zeit beobachtet werden, schalten wir den folgende Akzeptor-Automat parallel zum Systemmodell.

Definition 7.10 *Ereignis mit bestimmter Dauer ($\Diamond[\lceil\varphi\rceil \wedge l = n$)*



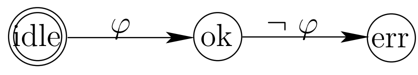
Sei φ ein Ereignis, das n Zeitschritte gilt. Der folgende Automat akzeptiert diese Aussage. Der Endzustand ok ist mit $L(ok) := ok_\varphi$ markiert.

ok_φ gilt erst, wenn φ n Zeitschritte lang gegolten hat und markiert damit das Ende des Ereignisses mit Dauer. Für die Modellprüfung der Vollständigkeit weisen wir dann $A\ ok_\varphi$ P ψ nach. Bevor die Wirkung eintritt, muss damit die Ursache (vollständig) auf dem Pfad aufgetreten sein, auf dem die Wirkung eintritt.

Dauerhaftes Ereignis

Soll ein dauerhaftes Ereignis φ als Ursache betrachtet werden, gibt es die Möglichkeit, dass das Eintreten des Ereignisses die Ursache ist. Dies ist ein punktuelles Ereignis und kann in CTL durch $A\ \varphi$ P ψ beschrieben werden. Eine zweite Möglichkeit ist, dass die Ursache gelten muss, bis die Wirkung eintritt.

Definition 7.11 *dauerhaftes Ereignis ($\boxtimes\neg$ ($\lceil\varphi\rceil^0$; true ; $\lceil\neg\ \varphi\rceil^0$) $\wedge \Diamond\lceil\varphi\rceil^0$)*



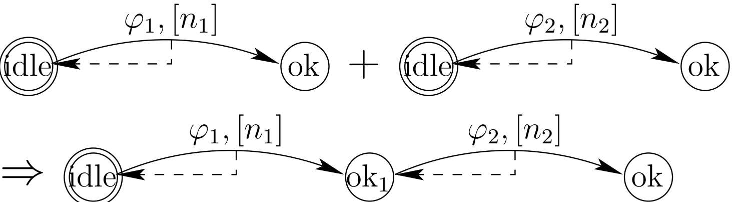
Sei φ ein Ereignis, das dauerhaft gilt. Dieses Aussage wird durch den folgenden Automaten akzeptiert. Der Zustand ist ok mit $L(ok) := ok_\varphi$, err mit $L(err) := ko_\varphi$ markiert.

ok_φ gilt, wenn φ einmal eingetreten ist und immer noch gilt. Da bei einem dauerhaften Ereignis φ bis zur Wirkung gelten soll, muss immer wenn die Wirkung gilt, auch ok_φ gelten. Wir zeigen also $AG\ \psi \rightarrow ok_\varphi$.

Sequenz von Ereignissen

Das dritte Ereignismuster beschreibt eine Sequenz temporaler Ereignisse als Ursache. Um eine Sequenz temporaler Ereignisse zu beschreiben, komponieren wir Akzeptor-Automaten für temporale Ereignisse aus Definition 7.10.

Definition 7.12 *Sequenz von Ereignissen* ($\diamond [\lceil \varphi_1 \rceil \wedge l = n_1 \,;\, \diamond [\lceil \varphi_2 \rceil \wedge l = n_2$)
Die Ereignisse φ_1 und φ_2 sollen nacheinander n_1 bzw. n_2 Zeiteinheiten gelten. Dann komponieren wir die einzelnen Akzeptor-Automaten wie folgt:



Der Zustand ok ist mit $L(ok) := ok_\varphi$ markiert und akzeptiert den sequentiellen Ablauf beider Ereignisse.

Erst wenn die gesamte Sequenz von Ereignissen eingetreten ist, wird der Endzustand betreten. Die angegebene Konstruktion kann leicht auf $n > 2$ Akzeptor-Automaten verallgemeinert werden.

Für die Vollständigkeitsbedingungen zeigen wir in Abschnitt 8.4.1 die Äquivalenz der ITL- und CTL-Formalisierung für punktuelle Ereignisse. Wir zeigen auch, dass die Akzeptor-Automaten-Konstruktion mit dem Nachweis der resultierenden CTL-Bedingungen der ITL-Semantik entspricht. Für diesen Nachweis formalisieren wir die Einbettung von ITL-Formeln in CTL und zeigen das entsprechende Theorem.

7.3 Beispiel: Drucktank

Mit den erarbeiteten Grundlagen für die Modellprüfung von Fehlerbäumen führen wir nun die formale FTA für den Drucktank aus Abschnitt 3.1 durch. Da in diesem Beispiel Zeiten eine wichtige Rolle spielen, beschreiben wir die Spezifikation des Modells mit zeitbehafteten Automaten, die Vollständigkeitsbedingungen in CCTL und führen die Modellprüfung mit RAVEN [Ruf00] durch.

Spezifikation des Drucktanks

Wir beschreiben das Systemmodell des Drucktanks samt Steuerung mit zeitbehafteten Automaten. Das Systemmodell in Abbildung 7.2 besteht aus acht Modulen, die den Komponenten des Drucktanks entsprechen. RAVEN benutzt qualifizierte Bezeichner, um auf Variablen eines anderen Moduls zu verweisen. Z. B. bedeutete *power := k2.close* im Modul *pump*, dass die Variable *power* gilt, wenn im Modul *k2 close* wahr ist. Dies ist dann der

Fall, wenn ein Zustand aktiv ist, der mit *close* beschriftet ist. Ein Zustand c, in dem *close* gilt, wird mit $c := \{close\}$ beschriftet. Bedingungen auf Übergängen können mit '!' negiert werden, und Ausgangszustände sind durch doppeltes Einkreisen gekennzeichnet.

Das Modul *env* beschreibt die Umgebung, die indeterministisch den Eingangsschalter s_1 betätigt (in Zustand *env.c* gilt *env.press*). Darauf reagiert der Schalter s_1, indem er schließt und in den Zustand $s_1.c$ übergeht. Dort verbleibt er solange *press* gilt. Dann kann er indeterministisch in den Zustand geöffnet ($s_1.o$) übergehen. Ist der Schalter jedoch defekt ('fails to open'), bleibt er im Zustand $s_1.c$ 'hängen', obwohl *press* nicht mehr gilt (siehe Übergang von c nach c in s_1, der mit !*press* beschriftet ist). Mit dieser indeterministischen Auswahl haben wir in diesem Projekt die Möglichkeit von Ausfällen modelliert (siehe auch Abschnitt 4.3). Die Module für das Relais k_1, das Relais k_2, den Drucksensor s und die Pumpe *pump* haben ein analoges Verhalten wie der Schalter und sind entsprechend modelliert. Der Timer ist ebenfalls wie ein Schalter modelliert, jedoch ist er nur maximal *timeout* Zeitschritte geschlossen (solange er korrekt funktioniert).

Der eigentliche Drucktank *tank* besteht aus den Zuständen 'leer' (e), 'füllend' (f), 'voll' (u) und 'explodiert' (r). Sobald er gefüllt (*flow*) wird, geht er in den Zustand 'füllend' über. Wird er dann *fill_t* Zeiteinheiten gefüllt, ist er 'voll'. Wird der Füllvorgang einmal unterbrochen, muss erneut *fill_t* Zeiteinheiten gefüllt werden. Der Drucksensor erkennt den Zustand 'Tank voll' (*tank.u*) und ist so angebracht, dass das System noch eine Reaktionszeit hat und bei weiterem Füllen der Tank nicht sofort 'explodiert'. Die Reaktionszeit beträgt *full_t* Zeiteinheiten. Wird der Füllvorgang unterbrochen, dauert es *drain_t* Zeiteinheiten, bis der Tank geleert ist.

Wir messen die Zeit in Sekunden, indem wir einen Zeitschritt im Modell als eine Sekunde definieren. Entsprechend der Fallstudienbeschreibung setzen wir die Zeitkonstanten *fill_t* und *drain_t* auf 54, *full_t* auf 6 und *timeout* auf 59 (Abbildung 7.2 oben). Damit ist die maximale Füllzeit $fill_t + full_t = 60$ Sekunden.

Modellprüfung der FTA-Bedingungen

Den Ausgangspunkt für die formale FTA des Drucktanks bildet der Fehlerbaum aus Abbildung 3.2, Abschnitt 3.1.1. Für die Formalisierung des Fehlerbaums benutzen wir die verschiedenen Typen von *Und*- bzw. *Oder*-Gatter aus Abschnitt 5.1. Die Analyse des Fehlerbaums, der ursprünglich aus dem *Fault Tree Handbook* [VGRH81] stammt, hat ergeben, dass ein Gatter tatsächlich einer Ursache/Wirkungs-Beziehung entspricht, obwohl im *Fault Tree Handbook* die Gatter informell als Zerlegungsgatter beschrieben sind. In Abbildung 7.3 sehen wir den formalen Fehlerbaum, dessen oberstes Gatter g_1 eine Ursache/Wirkungs-Beziehung beschreibt. Die Ursache hat eine Dauer und kann deshalb nicht gleichzeitig mit der punktuellen Wirkung eintreten (Betreten des Zustands *tank.r*). Alle weiteren Übergänge im Fehlerbaum sind Zerlegungen, wobei g_2, g_7 und g_8 nur eine Ursache haben und das Ereignis umbenennen.

Für die Modellprüfung werden die einzelnen Ereignisse, wie in Abbildung 7.3 zu sehen, formalisiert. Dazu bekommen die informellen Ereignisse eine formale Entsprechung über den Variablen im Automatenmodell des Drucktanks. Aus den Gatter-Typen berechnet sich

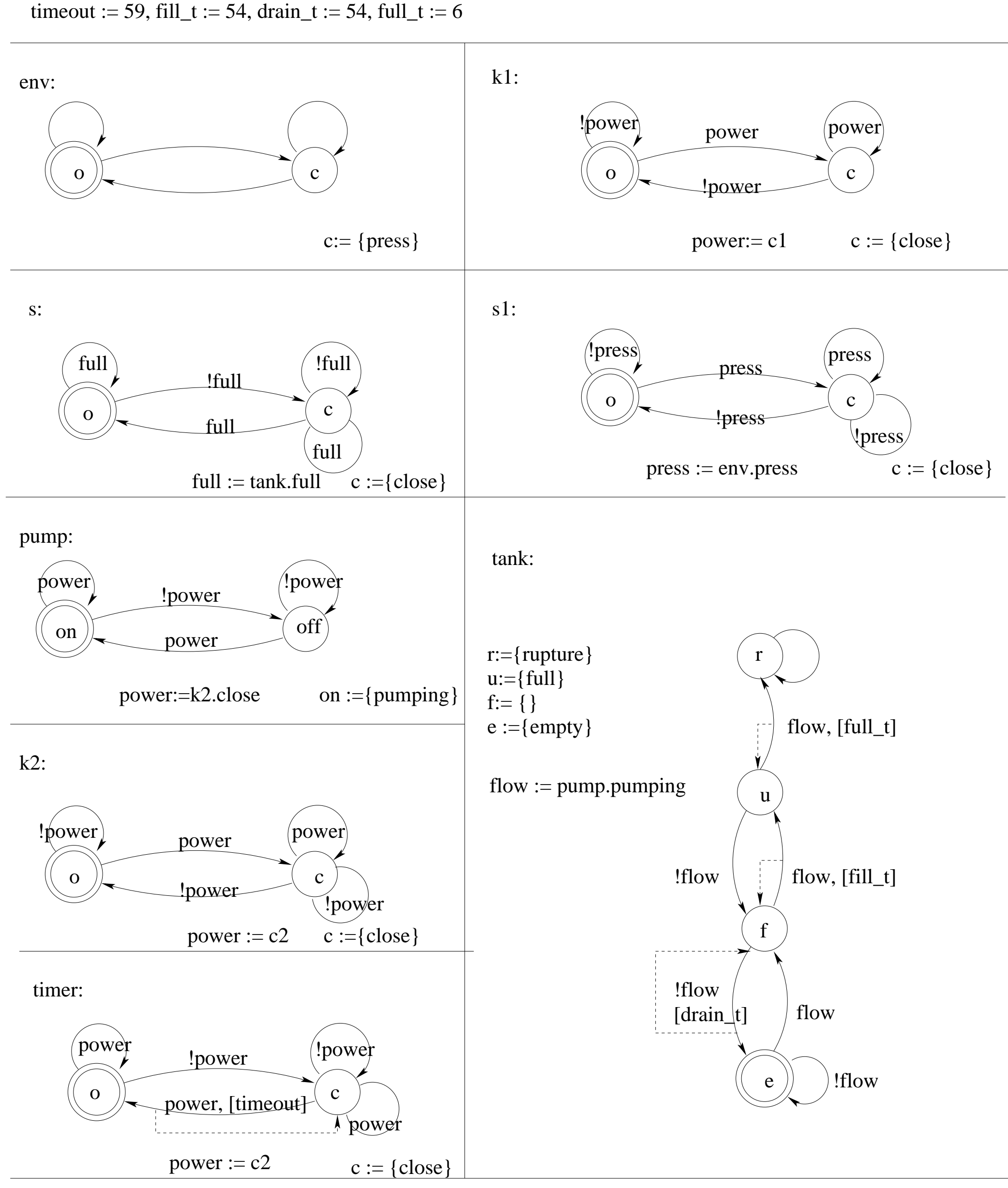
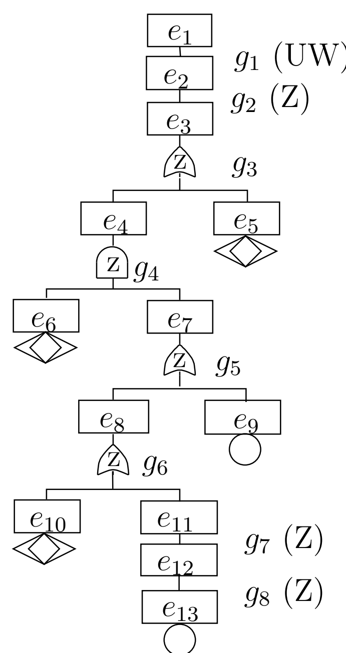


Abbildung 7.2: Modell des Drucktanks mit Steuerung



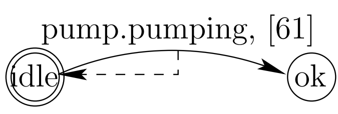
e_1	Tank platzt	$tank.rupture$
e_2	länger als 60 sek füllen	$\lceil\lceil pump.pumping \rceil \wedge \ell = 61$
e_3	K_2 länger als 60 sek geschlossen	$\lceil\lceil k_2.close \rceil \wedge \ell = 61$
e_4	Strom länger als 60 sek an K_2	$\lceil\lceil c_2 \rceil \wedge \ell = 61$
e_5	K_2 öffnet nicht	$\neg k_2.power \rightarrow EX\ K_2.close$
e_6	S defekt	$tank.full \rightarrow EX\ S.close$
e_7	Strom länger als 60 sek an S	$\lceil\lceil c_{in} \rceil \wedge \ell = 61$
e_8	Strom länger als 60 sek durch K_1, während S geschlossen	$\lceil\lceil k_1.close \rceil \wedge \ell = 61$
e_9	Strom länger als 60 sek durch S_1, während S geschlossen	$\lceil\lceil s_1.close \rceil \wedge \ell = 61$
e_{10}	K_1 öffnet nicht	$\neg k_1.power \rightarrow EX\ K_1.close$
e_{11}	Strom länger als 60 sek an K_1 während S geschlossen	$\lceil\lceil c_1 \rceil \wedge \ell = 61$
e_{12}	Timer öffnet nach 60 sek nicht, während S geschlossen	$(\lceil\lceil timer.power \rceil \wedge \ell = 61) \rightarrow \text{timer.close}$
e_{13}	Timer defekt	

Abbildung 7.3: Formaler Fehlerbaum für den Drucktank

dann die entsprechende Bedingung, deren Nachweis die Vollständigkeit des Fehlerbaums garantiert.

Für den Nachweis der Vollständigkeit des Gatters g_1 benötigen wir einen Akzeptor-Automaten, der das temporale Ereignis e_2 auf ein punktuelles Ereignis abbildet.

Beispiel 7.3 *Akzeptor-Automat* a_{e_2}



Der Automat akzeptiert ohne Unterbrechung irgendwann 61 sek pump.pumping *(> 60 sek). Sobald die 61 sek vorüber sind, signalisiert dies der Zustand* ok.

Die Vollständigkeit des Gatters g_1 zeigen wir mit der Bedingung $A\ a_{e_2}.ok$ P $tank.rupture$.

Theorem 7.1 *Vollständigkeit des Gatters* g_1
Erweitern wir das Modell des Druckbehälters aus Abbildung 7.2 mit dem Akzeptor-Automaten a_{e_2}*, so gilt*

$$A\ a_{e_2}.ok \ \mathrm{P}\ tank.rupture.$$

Auf diese Art und Weise kann nun für jedes Ereignis mit Dauer ein Akzeptor-Automat angegeben werden. Wir haben die Akzeptor-Automaten in Abbildung 7.4 zusammengefasst. Mit diesen Automaten lassen sich nun die Gatterbedingungen des Fehlerbaums formulieren.

Theorem 7.2 *Vollständigkeit des Drucktank Fehlerbaums*
Die Vollständigkeit des Fehlerbaums zeigen wir durch den Nachweis der einzelnen Gatter. Für das Modell aus Abbildung 7.2 gilt

$$\begin{aligned}
g_2 :&\quad AG\ (EX\ a_{e_2}.ok) \rightarrow a_{e_3}.ok,\\
g_3 :&\quad AG\ (EX\ a_{e_3}.ok) \rightarrow (a_{e_4}.ok \vee (\neg\ k_2.power \rightarrow (EX\ k_2.close))),\\
g_4 :&\quad AG\ a_{e_4}.ok \rightarrow (a_{e_7}.ok \wedge (tank.full \rightarrow (EX\ s.close))),\\
g_5 :&\quad AG\ s.close \rightarrow ((EX\ a_{e_7}.ok) \rightarrow a_{e_8}.ok \vee a_{e_9}.ok),\\
g_6 :&\quad AG\ s.close \rightarrow ((EX\ a_{e_8}.ok)\\
&\qquad\qquad \rightarrow (a_{e_{11}}.ok \vee (\neg\ k_1.power \rightarrow (EX\ k_1.close)))\ \textit{und}\\
g_7 :&\quad AG\ s.close \rightarrow ((EX\ a_{e_{11}}.ok) \rightarrow (a_{e_{12}}.ok \rightarrow (EX\ timer.close))).
\end{aligned}$$

Die Bedingung g_8 *wird für den Vollständigkeitsnachweis nicht benötigt, da* e_{12} *und* e_{13} *die selben Ereignisse sind, informell jedoch umformuliert wurden.*

Bei der Formulierung der Bedingungen fällt auf, dass die Wirkungen meist einen Zeitschritt später als die Ursachen eintreten. Dies liegt am Ausführungsmodell der Intervallautomaten. Ist z. B. der Schalter s_1 im Zustand $s_1.o$ und wird durch die Umgebung aktiviert (*env.press*), ist er erst im nächsten Zeitschritt geschlossen, denn der Zustandsübergang nach $s_1.c$ benötigt einen Zeitschritt. Diese Verzögerung tritt immer in den Situationen auf,

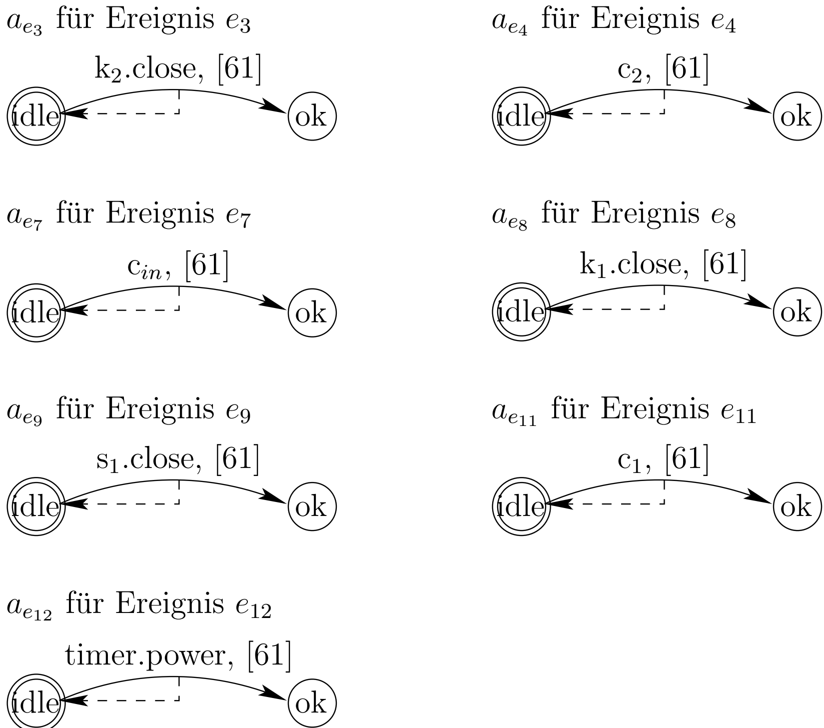


Abbildung 7.4: Akzeptor-Automaten für den Drucktank

in denen das System auf Ereignisse reagiert. Beim Nachweis von g_4 ist der Zeitschritt nicht notwendig, da dort Bedingungen nur umformuliert werden.

Bei den Beweisen für die Gatter g_5, g_6 und g_7 ist zu bemerken, dass die Vorbedingung *s.close* der informellen Nebenbedingung „während s geschlossen ist“ entspricht. Die Wirkung ist für g_5 $a_{e_7}.ok$, für g_6 $a_{e_8}.ok$ und für g_7 $a_{e_{11}}.ok$. Mit dem Nachweis aller Gatterbedingungen, den wir in RAVEN geführt haben, ist die formale FTA für den Drucktank abgeschlossen.

Bemerkung Mit einer Meta-Überlegung können wir in diesem Beispiel sogar auf die Konstruktion der Akzeptor-Automaten aus Abbildung 7.4 verzichten. In diesem Beispiel sind alle Zeitbedingungen gleich, d. h. alle Ereignisse müssen länger als 60 Sekunden eintreten. Weiterhin treten die Ereignisse „gleichzeitig“ ein. Wenn wir nun zeigen, dass immer wenn *pump.pumping* gilt (Wirkung) auch das Relais K_2 geschlossen ist ($k_2.close$, Ursache), haben wir erst recht gezeigt, dass wenn *pump.pumping* 60 Sekunden gilt, auch das Relais K_2 60 Sekunden geschlossen sein muss. Allerdings enthält obige Beweisverpflichtung nun noch punktuelle Ereignisse. Mit dieser Überlegung können wir die Vollständigkeit des Drucktanks auch folgendermaßen zeigen.

Theorem 7.3 *Vollständigkeit des Drucktank-Fehlerbaums(II)*
Die Vollständigkeit des Fehlerbaums zeigen wir durch den Nachweis der einzelnen Gatter.

Für das Modell aus Abbildung 7.2 gilt

$$
\begin{aligned}
g_2: &\quad AG\ (EX\ pump.open) \rightarrow k_2.close,\\
g_3: &\quad AG\ (EX\ k_2.close) \rightarrow (c_2 \vee (\neg\ k_2.power \wedge (EX\ k_2.close))),\\
g_4: &\quad AG\ c_2 \rightarrow ((c_in) \wedge (tank.full \rightarrow (EX\ s.close))),\\
g_5: &\quad AG\ s.close \rightarrow ((EX\ c_in) \rightarrow s_1.close \vee k_1.close),\\
g_6: &\quad AG\ s.close \rightarrow ((EX\ k_1.close)\\
&\qquad \rightarrow (c_1 \vee (\neg\ k_1.power \wedge (EX\ k_1.close)))\ \mathit{und}\\
g_7: &\quad AG\ s.close \rightarrow ((EX\ c_1) \rightarrow (timer.power \rightarrow (EX\ timer.close))).
\end{aligned}
$$

Wir haben die Gatterbedingungen sowohl mit dieser Formalisierung, als auch mit der Akzeptor-Automaten-Konstruktion nachgewiesen. Die Reduktion auf punktuelle Ereignisse ohne Akzeptor-Automaten lässt sich immer dann anwenden, wenn die Zeitdauern der temporalen Ereignisse gleich sind und die Ereignisse gleichzeitig eintreten.

Nutzen der Vollständigkeitsprüfung

Um den Nutzen der Vollständigkeitsprüfung von Fehlerbäumen zu demonstrieren, spielen wir das „Vergessen" einer Ursache im Fehlerbaum nach. Dazu lassen wir für den Nachweis von g_3 die Ursache „k_2 öffnet nicht" ($\neg\ k_2.power \wedge (EX\ k_2.close)$) weg. Die Vollständigkeitsbedingung kann nicht mehr gezeigt werden und der Modellprüfer RAVEN erzeugt ein Gegenbeispiel. Der Pfad zum Gegenbeispiel hilft, die fehlende Ursache zu identifizieren. Die zusätzliche Ursache wird dann in den Fehlerbaum aufgenommen. Wurden bei der FTA mehrere Ursachen übersehen, kann so nach und nach der Fehlerbaum vervollständigt werden.

Beispiel 7.4 *Fehlende Ursachen „k_2 öffnet nicht"*

Das Gegenbeispiel zeigt welche Zustände im System (links notiert) wann aktiv sind (waagrechter Balken, recht) und wann nicht (dünne Linie). Der Fehlerzustand ist mit dem senk-

rechten Balken markiert. Man erkennt, dass das Relais k_2 geschlossen ist (k2.k2#0), obwohl s geöffnet ist (s entspricht hier pr_switch.pr_switch#0) und damit kein Strom an k_2 anliegt. Dies beschreibt die Ursache „k_2 öffnet nicht“.

Ergebnisse

Die Drucktank-Fallstudie zeigt, dass die formale FTA für zustandsendliche Systeme mit Modellprüfung mit angemessenem Aufwand durchführbar ist. Das Beispiel aus dem *Fault Tree Handbook* [VGRH81] ist schon sehr genau untersucht und deshalb konnte, wie erwartet, die Vollständigkeit des Fehlerbaums gezeigt werden. Es hat sich aber auch gezeigt, dass schon für dieses Beispiel die Zerlegungsgatter der herkömmlichen FTA nicht ausreichend sind, sondern Ursache/Wirkungs-Gatter zur korrekten Beschreibung notwendig sind.

Obwohl in diesem Beispiel alle Gatter vollständig sind, zeigt das Beispiel 7.4, welchen Nutzen die formale FTA bringen kann. Sind Ursachen nicht vollständig aufgezählt, konstruiert der Modellprüfer ein Gegenbeispiel, das die fehlende Ursache beschreibt.

7.4 Verwandte Arbeiten

In der Diplomarbeit [Sch01, Sch03] von Schäfer (Universität Oldenburg, Prof. Dr. E.-R. Olderog, Abteilung Semantik) wird die Duration Calculus (DC) Modellprüfung von Fehlerbäumen vorgestellt. Der DC ist eine Erweiterung der ITL für kontinuierliche Systeme. Die ITL-Formalisierung der Fehlerbäume aus Abschnitt 5.2 wurde in der Arbeit [Sch01] zur DC-Modellprüfung der FTA-Vollständigkeitsbedingungen übernommen. Die DC-Modellprüfung mit Moby/DC [Tap01] basiert auf sogenannten Phasenautomaten, mit denen eine Untermenge des DC dargestellt werden kann. Phasenautomaten beschreiben Abläufe in einem System. Mit ihnen werden sowohl die Modelle als auch die nachzuweisenden Aussagen beschrieben. Verifiziert wird dann die Negation der nachzuweisenden Aussage, d. h. solch ein Ablauf kann niemals auftreten. Mit den Phasenautomaten können Formeln der folgenden Grammatik beschrieben werden (P ist ein Zustandsausdruck und $expr(l)$ ein Intervallausdruck über l, der Länge des Intervalls).

$$\begin{aligned} F \;\; ::= \;\; & (F1 \vee F2) \mid (\lceil P \rceil \wedge expr(l)); F \mid ((\lceil P \rceil \wedge expr(l)) \vee l = 0); F \mid \\ & (\lceil P \rceil \wedge expr(l)) \mid ((\lceil P \rceil \wedge expr(l)) \vee l = 0) \end{aligned}$$

Für die Verifikation zeigt man dann $\neg\, F$.

Die FTA-Bedingungen sind aber nicht in dieser Grammatik enthalten, da innerhalb der Vollständigkeitsbedingungen für die Ursache/Wirkungs-Gatter Negation auftritt. Phasenautomaten sind aber nicht gegen Komplementbildung abgeschlossen, d. h. die Vollständigkeitsbedingungen können nicht direkt mit Phasenautomaten beschrieben werden (darin unterscheidet sich der kontinuierliche DC vom diskreten DC bzw. der ITL, die gegen Komplementbildung abgeschlossen ist [Pan01b]).

Trotzdem ist es in der Arbeit von Schäfer gelungen, für alle FTA-Bedingungen spezielle Phasenautomaten abzuleiten. Auch Phasenautomaten für die Ereignistypen aus Górski

[GW95] wurden angegeben (analog zu den in Abschnitt 7.2.2 präsentierten Akzeptor-Automaten). Damit können Fehlerbäume, die aus Zerlegungs- und Ursache/Wirkungs-Gatter aufgebaut sind und Ereignisse *mit Dauer*, *dauerhafte* Ereignisse oder *Sequenzen* von Ereignissen enthalten, mit dem Modellprüfer Moby/DC auf Vollständigkeit geprüft werden. Zur Demonstration des Ansatzes wurde auch die Drucktank-Fallstudie in [Sch01] erfolgreich durchgeführt.

7.5 Resultate

In diesem Kapitel wurden Vollständigkeitsbedingungen der FTA in CTL definiert. Obwohl diese CTL-Bedingungen nur für punktuelle Fehlerbaumereignisse der ITL-Semantik entsprechen, gelingt es mit Hilfe sogenannter Akzeptor-Automaten, auch temporale Ereignisse korrekt zu behandeln. Die Akzeptor-Automaten werden parallel zum Systemmodell ausgeführt und 'akzeptieren' ein Ereignis. Damit lassen sich alle Ereignisse behandeln, die von endlichen Automaten akzeptiert werden. Insbesondere lassen sich Formeln des QDDC beschreiben, die die Ereignistypen nach Górski formalisieren. Damit kann die formale FTA für zustandsendliche Systeme durchgeführt, und die Vollständigkeitsbedingungen können automatisch mittels Modellprüfung nachgewiesen werden.

Im folgenden Kapitel zeigen wir für punktuelle Ereignisse formal, dass die CTL-Vollständigkeitsbedingungen äquivalent zur ITL-Semantik sind. Das minimale Schnittmengen-Theorem 5.1 gilt damit auch für die Vollständigkeitsprüfung in CTL. Außerdem können wir statt punktueller Ereignisse auch QDDC-Formeln (und damit temporale Ereignisse) in die CTL-Formalisierung einbetten. Wir zeigen, dass diese Einbettung zusammen mit den CTL-Vollständigkeitsbedingungen äquivalent zur ITL-Semantik von Fehlerbäumen ist. Mit dem Theorem 7.1 von Pandya [Pan01a] erreichen wir, dass auch die CTL-Formalisierung mit Akzeptor-Automaten für temporale Ereignisse der ITL-Semantik entspricht.

Zur Evaluierung der CTL-Formalisierung wurde die Fallstudie „Drucktank“ durchgeführt. Es hat sich herausgestellt, dass bei dieser Fallstudie die herkömmlichen Zerlegungsgatter nicht genügen, um den Fehlerbaum korrekt zu beschreiben, sondern die von uns beschriebenen Ursache/Wirkungs-Gatter benötigt werden. Nachdem die Ereignisse formalisiert und die entsprechenden Gatterbedingungen erzeugt wurden, konnte die Vollständigkeit des Fehlerbaums mit CTL-Modellprüfung nachgewiesen werden.

KAPITEL 8

Formaler Vergleich von FTA-Semantiken

In der bisherigen Arbeit haben wir verschiedene Semantiken für die FTA kennengelernt. In Abschnitt 5.5 wurde kurz erwähnt, dass die bekannten Ansätze aus der Literatur (Hansen et al. [HRS94] und Bruns und Anderson [BA93]) Schwächen bei temporalen Ereignissen haben. Ähnlich verhält es sich bei der in Kapitel 7 beschriebenen Formalisierung in CTL. In CTL müssen zusätzlich Akzeptor-Automaten zum Systemmodell hinzugefügt werden, die temporale Ereignisse akzeptieren. Nur die ITL-Formalisierung aus Kapitel 5 behebt diese Schwächen. In diesem Kapitel untersuchen wir diese Schwächen genauer. Bei der Untersuchung fällt auf, dass sie *nur* bei temporalen Ereignissen auftreten. Es kann sogar formal nachgewiesen werden, dass bei punktuellen Ereignissen alle besprochenen Semantiken äquivalent sind. Diesen Nachweis haben wir im Spezifikations- und Verifikationswerkzeug KIV [BRS+00] durchgeführt. Es wurde ein semantisches Rahmenwerk geschaffen, in dem die verschiedenen Formalisierungen ausgedrückt werden können. Grundlage dafür bildete eine Erweiterung der in Abschnitt 5.2 vorgestellten ITL.

Die Erweiterungen der ITL definieren wir in Abschnitt 8.1. Die erweiterte ITL enthält zusätzlich zu den üblichen ITL-Operatoren Vergangenheitsoperatoren für den μ-Kalkül mit Vergangenheit und CTL-Operatoren. Anschließend gehen wir in Abschnitt 8.2 detailliert auf die Schwächen der einzelnen FTA-Formalisierungen ein. Für die formalen Beweise spezifizieren wir die erweiterte ITL algebraisch in KIV. Einige Besonderheiten und wichtige Aspekte dieser Spezifikation präsentieren wir in Abschnitt 8.3, die gesamte algebraische Spezifikation findet sich im Anhang C. Schließlich weisen wir in Abschnitt 8.4.1 nach, dass für punktuelle Ereignisse alle Semantiken äquivalent sind und in Abschnitt 8.4.2, dass mit der Akzeptor-Automaten-Konstruktion auch temporale Ereignisse in CTL korrekt behandelt werden.

8.1 Erweiterte ITL

Für den formalen Vergleich der verschiedenen Semantikdefinitionen müssen die Logiken ITL, DC [CHR91], μ-Kalkül [Koz83] mit Vergangenheit und CTL in einem gemeinsamen logischen Rahmen betrachtet werden. Kontinuierliche Aspekte sind für die Vollständigkeits-

bedingungen der FTA nicht relevant, deshalb interpretieren wir die DC-Bedingungen über dem diskreten Zeitmodell der ITL. Die CTL-Bedingungen für den Vollständigkeitsnachweis einer FTA wurden in Abschnitt 7.2 zwar über einer erweiterten CTL, der clocked CTL (CCTL), definiert. Dies war aber nur für den verwendeten Modellprüfer notwendig, der CCTL-Formeln prüft. Die Definition der Vollständigkeitsbedingungen benutzt keine Operatoren der CCTL-Erweiterung. Deshalb betrachten wir in unserem Rahmenwerk nur CTL-Operatoren. Als Grundlage für den logischen Rahmen dient die ITL-Semantikdefinition aus Abschnitt 5.2, die wir um Vergangenheitsoperatoren des μ-Kalküls mit Vergangenheit und Pfadoperatoren der CTL erweitern. Dabei halten wir uns an die Definitionen von Stirling [Sti92]. Die Syntax und Semantik dieser zusätzlichen Operatoren werden in diesem Abschnitt kurz beschrieben. Die erweiterten Intervalltemporallogik ITL nennen wir ITL$^+$.

Wir definieren die Semantik der ITL$^+$ über Intervallen aus $\mathbb{Z}$, die sowohl negativ unendliche untere Grenzen $-\infty$ als auch unendliche obere Grenzen ∞ haben können. Wir sehen davon ab, dass die Intervallgrenzen bei $-\infty$ und ∞ offen sind und vereinbaren, dass wir für $[a,b]$, $(-\infty,b]$, $[a,\infty)$ bzw. $(-\infty,\infty)$ mit $a,b \in \mathbb{Z}$ einfach $[a,b]$ mit $a \in \mathbb{Z} \cup \{-\infty\}$, $b \in \mathbb{Z} \cup \{\infty\}$ schreiben. Die Menge aller Intervalle IV aus $\mathbb{Z}$ ist dann IV $:= \{[a,b] \mid a \in \mathbb{Z} \cup \{-\infty\},\ b \in \mathbb{Z} \cup \{\infty\}$ und $a \leq b\}$. Eine Interpretation $\mathcal{J} : [a,b] \to (V \to \mathcal{D})$ ist eine Abbildung von Intervallen nach Belegungen. Eine Belegung $\sigma : V \to \mathcal{D}$ ordnet Variablen einen Wert aus Ihrem Wertebereich $\mathcal{D}$ zu. Der Domain $dom(\mathcal{J})$ einer Interpretation ist $[a,b]$, und wir definieren für eine Interpretation $\mathcal{J}$ mit $dom(\mathcal{J}) = [a,b]$, dass $min(\mathcal{J}) := a$, $max(\mathcal{J}) := b$ und $t \in \mathcal{J} :\Leftrightarrow min(\mathcal{J}) \leq t \leq max(\mathcal{J})$. Die Menge aller Interpretationen über einer Menge von Intervallen $I \subseteq$ IV nennen wir $J(I)$. Für eine Interpretation $\mathcal{J}$ mit Domain $dom(\mathcal{J}) = [a,b]$ beschreibt, für $a \leq c \leq d \leq b$, $\mathcal{J}\mid_{[c,d]}$ die auf $dom(\mathcal{J}\mid_{[c,d]}) = [c,d]$ eingeschränkte Interpretation $\mathcal{J}$. Die Menge aller Formeln der ITL$^+$ nennen wir Fma_{ITL^+}.

Ein Trace (oder Intervall) $\mathcal{I} = (\sigma_0, \sigma_1, \ldots)$, den wir in Abschnitt 5.2 als Folge von Belegungen definiert haben, entspricht in dem hier definierten temporallogischen Rahmen einer Interpretation $\mathcal{J}$ mit $dom(\mathcal{J}) = [0, \mathit{length}(\mathcal{I})]$, wobei für alle $0 \leq i \leq \mathit{length}(\mathcal{I})$ gilt: $\mathcal{J}(i) = \sigma_i$.

Um Vergangenheitsoperatoren behandeln zu können, dürfen wir bei der Auswertung von Zukunfts-Operatoren keine Präfixe von Intervallen abschneiden, wie es z. B. bei dem $\otimes$-Operator in Definition 5.2, Seite 58, geschieht. Ansonsten würden wir die Vergangenheit „vergessen“. Deshalb werten wir Formeln zu einem Zeitpunkt t aus, der nicht dem Anfang eines Intervalls entsprechen muss. Um auch die CTL-Operatoren, die über Ausführungspfade von Systemen quantifizieren, betrachten zu können, benötigen wir für die Auswertung von Formeln eine Menge von Interpretationen $\mathcal{IS} \subseteq J(\text{IV})$, die alle Ausführungspfade eines Systems beschreiben. Schließlich benötigen wir für die Auswertung der prädikatenlogischen Anteile eine Algebra $\mathcal{A}$. Eine temporallogische Formel $\varphi \in Fma_{ITL^+}$ gilt, wenn für $\mathcal{J} \in \mathcal{IS}$ und $t \in \mathcal{J}$ $\mathcal{A}, \mathcal{IS}, \mathcal{J}, t \models_{ITL^+} \varphi$. Für eine prädikatenlogische Formel $\Phi \in Fma_{PL}$ gilt $\mathcal{A}, \mathcal{IS}, \mathcal{J}, t \models_{ITL^+} \Phi$ genau dann, wenn $\mathcal{A}, \mathcal{J}(t) \models_{PL} \Phi$. Wie üblich lassen wir die Algebra $\mathcal{A}$ weg, wenn sie für das Verständnis nicht wichtig ist und schreiben $\models$ statt $\models_{ITL^+}$, wenn die Bedeutung aus dem Kontext klar ist. Statt $\mathcal{A}, \mathcal{IS}, \mathcal{J}, t \models_{ITL^+} \varphi$ schreiben wir einfach $\mathcal{IS}, \mathcal{J}, t \models \varphi$.

Wir definieren nun die temporalen Operatoren, die wir für die folgenden Äquivalenz-

beweise benötigen und vereinbaren, dass $a = min(\mathcal{J})$ und $b = max(\mathcal{J})$ sei.

Definition 8.1 *Chop-Operator*

Das Intervall der Interpretation kann so geteilt werden, dass φ über dem ersten und ψ über dem zweiten Teilintervall der Interpretation $\mathcal{J}$ gilt.

$$\mathcal{IS}, \mathcal{J}, t \models \varphi \ ; \ \psi \quad :\Leftrightarrow \quad \textit{es gibt ein ein } c \neq \infty \textit{ mit } t \leq c \leq b: \\ \mathcal{J}|_{[a,c]}, t \models \varphi \textit{ und } \mathcal{J}, c \models \psi$$

Die Semantik des Chop-Operators ist analog zu Definition 5.1. Für die Auswertung von ψ werten wir jedoch $\mathcal{J}$ zum Zeitpunkt c aus, anstatt die Einschränkung $\mathcal{J}|_{[c,b]}$ zu betrachten (diese Einschränkung würde dem Abschneiden von Präfixen nach Definition 5.1 entsprechen). So können wir mit entsprechenden Vergangenheitsoperatoren wieder auf Zeitpunkte $a \leq t \leq c$ zurückblicken. Im Gegensatz dazu schränken wir $\mathcal{J}$ für die Auswertung von φ explizit auf $\mathcal{J}|_{[a,c]}$ ein. Für die Auswertung von φ können also keine Zeitpunkte $c \leq t \leq b$ betrachtet werden, der Chop-Operator teilt die Interpretation.

Definition 8.2 *CTL und Vergangenheitsoperatoren*

- ψ U φ, *Until-Operator: es gilt solange ψ, bis φ gilt (und φ muss irgendwann gelten)*

$$\mathcal{IS}, \mathcal{J}, t \models \psi \text{ U } \varphi \quad :\Leftrightarrow \quad \textit{es gibt ein } c \neq \infty,\ t \leq c \leq b: \\ \mathcal{IS}, \mathcal{J}, c \models \varphi \textit{ und für alle } e,\ t \leq e < c: \\ \mathcal{IS}, \mathcal{J}, e \models \psi$$

- ψ S φ, *Since-Operator: seit φ galt, gilt ψ (und φ galt irgendwann)*

$$\mathcal{IS}, \mathcal{J}, t \models \psi \text{ S } \varphi \quad :\Leftrightarrow \quad \textit{es gibt ein } c \neq -\infty,\ a \leq c \leq t: \\ \mathcal{IS}, \mathcal{J}, c \models \varphi \textit{ und für alle } e,\ c < e \leq t: \\ \mathcal{IS}, \mathcal{J}, e \models \psi$$

- *A φ, All-Path-Quantor: auf allen Pfaden gilt φ*

$$\mathcal{IS}, \mathcal{J}, t \models A\ \varphi \quad :\Leftrightarrow \quad \textit{für alle } \mathcal{J}' \in \mathcal{IS} \textit{ mit } t \in \mathcal{J}' \textit{ und } \mathcal{J}'(t) = \mathcal{J}(t): \\ \mathcal{IS}, \mathcal{J}', t \models \varphi$$

Vom Chop-Operator und diesen zusätzlichen Basisoperatoren lassen sich die für die Formalisierung der FTA-Semantiken notwendigen temporalen Operatoren ableiten. Alle ITL-Operatoren werden auf *einem* Intervall $\mathcal{J}$ ausgewertet, und die möglichen Ausführungspfade $\mathcal{IS}$ sind bei der Auswertung nicht relevant. Deshalb können die ITL-Operatoren aus Abschnitt 5.2 äquivalent in diesem temporallogischen Rahmen definiert werden. Wir leiten sie analog zur Definition 5.2 vom Chop-Operator ab.

Definition 8.3 *abgeleitete Operatoren*

- $\Diamond\ \varphi$, *Eventually-Operator: es gilt irgendwann* φ
  $\Diamond\ \varphi := \text{true U } \varphi$[1]
- $\Box\ \varphi$, *Always-Operator: es gilt immer* φ
  $\Box\ \varphi := \neg\Diamond\ \neg\varphi$
- $\Diamond_i\ \varphi$, *in einigen initialen Intervallen gilt* φ
  $\Diamond_i\ \varphi := \varphi$ **;** true
- $\Box_i\ \varphi$, *in allen initialen Intervallen gilt* φ
  $\Box_i\ \varphi := \neg\ \Diamond_i\ \neg\ \varphi$
- $\Diamond_s\ \varphi$, *in einigen Sub-Intervallen gilt* φ
  $\Diamond_s\ \varphi :=$ true **;** φ **;** true
- $\Box_s\ \varphi$, *in allen Sub-Intervallen gilt* φ
  $\Box_s\ \varphi := \neg\ \Diamond_s\ \neg\ \varphi$
- $\psi \text{ P } \varphi$ *Precedes-Operator: wenn irgendwann* φ *gilt, gilt zuvor (oder gleichzeitig)* ψ
  $\psi \text{ P } \varphi := \neg(\neg\psi \text{ U } (\varphi \wedge \neg\ \psi)) \vee \Box\neg\varphi$
- $\ominus\!\!\Diamond\ \varphi$, *Past-Eventually-Operator: es galt irgendwann* φ
  $\ominus\!\!\Diamond\ \varphi := \text{true S } \varphi$
- $\boxminus\ \varphi$, *Past-Always-Operator: es galt immer* φ
  $\boxminus\ \varphi := \neg\ \ominus\!\!\Diamond\ \neg\ \varphi$
- $E\ \varphi$, *Exists-Path-Operator: es gibt einen Pfad, auf dem gilt* φ
  $E\ \varphi := \neg\ A\ \neg\ \varphi$

Das folgende Lemma zeigt die expliziten Definitionen der abgeleiteten Operatoren.

Lemma 8.1 *Explizite Semantik abgeleiteter Operatoren*

$$\begin{array}{lll}
\mathcal{IS}, \mathcal{J}, t \models \Diamond\varphi & :\Leftrightarrow & \textit{es gibt ein } c \neq \infty \textit{ mit } t \leq c \leq b : \\
 & & \mathcal{A}, \mathcal{IS}, \mathcal{J}, c \models \varphi \\
\mathcal{IS}, \mathcal{J}, t \models \Box\varphi & :\Leftrightarrow & \textit{für alle } c \neq \infty \textit{ mit } t \leq c \leq b : \\
 & & \mathcal{A}, \mathcal{IS}, \mathcal{J}, c \models \varphi
\end{array}$$

[1] Alternativ: $\Diamond\ \varphi :=$ true **;** φ.

$$
\begin{array}{rcl}
\mathcal{IS}, \mathcal{J}, t \models \Diamond_{\mathrm{i}}\varphi & :\Leftrightarrow & \textit{es gibt ein } c \neq \infty \textit{ mit } t \leq c \leq b: \\
 & & \mathcal{A}, \mathcal{IS}, \mathcal{J}\,|_{[a,c]}, t \models \varphi \\
\mathcal{IS}, \mathcal{J}, t \models \Box_{\mathrm{i}}\varphi & :\Leftrightarrow & \textit{für alle } c \neq \infty \textit{ mit } t \leq c \leq b: \\
 & & \mathcal{A}, \mathcal{IS}, \mathcal{J}\,|_{[a,c]}, t \models \varphi \\
\mathcal{IS}, \mathcal{J}, t \models \Diamond_{\mathrm{x}}\varphi & :\Leftrightarrow & \textit{es gibt ein } c, d \neq \infty \textit{ mit } t \leq c \leq d \leq b: \\
 & & \mathcal{A}, \mathcal{IS}, \mathcal{J}\,|_{[a,d]}, c \models \varphi \\
\mathcal{IS}, \mathcal{J}, t \models \Box_{\mathrm{x}}\varphi & :\Leftrightarrow & \textit{für alle } c, d \neq \infty \textit{ mit } t \leq c \leq d \leq b: \\
 & & \mathcal{A}, \mathcal{IS}, \mathcal{J}\,|_{[a,d]}, c \models \varphi \\
\mathcal{IS}, \mathcal{J}, t \models \psi \ \mathrm{P} \ \varphi & :\Leftrightarrow & \textit{wenn für alle } c \neq \infty,\ t \leq c \leq b \textit{ mit} \\
 & & \mathcal{IS}, \mathcal{J}, c \models \varphi \textit{ ein } d,\ t \leq d \leq c, \textit{ existiert mit} \\
 & & \mathcal{IS}, \mathcal{J}, d \models \psi \\
\mathcal{IS}, \mathcal{J}, t \models \Diamond_{-}\varphi & :\Leftrightarrow & \textit{es gibt ein } c \neq -\infty \textit{ mit } a \leq c \leq t: \\
 & & \mathcal{IS}, \mathcal{J}, c \models \varphi \\
\mathcal{IS}, \mathcal{J}, t \models \Box_{-}\varphi & :\Leftrightarrow & \textit{für alle } c \neq -\infty \textit{ mit } a \leq c \leq t: \\
 & & \mathcal{IS}, \mathcal{J}, c \models \varphi \\
\mathcal{IS}, \mathcal{J}, t \models E\ \varphi & :\Leftrightarrow & \textit{wenn es ein } \mathcal{J}' \in \mathcal{IS} \textit{ mit } t \in \mathcal{J}' \textit{ und } \mathcal{J}'(t) = \mathcal{J}(t) \textit{gibt:} \\
 & & \mathcal{IS}, \mathcal{J}', t \models \varphi
\end{array}
$$

An dieser Stelle möchten wir auf den Unterschied zwischen dem Chop-Operator „;“ und dem Precedes-Operator „P“ hinweisen. Während der Chop-Operator das entsprechende Intervall der Interpretation wirklich in zwei Teile zerlegt, wertet der Precedes-Operator die Formeln an verschiedenen Zeitpunkten aus, aber im selben Intervall.

Eine Formel φ ist genau dann über den Ausführungspfaden $\mathcal{IS} \subseteq J(IV)$ wahr, d. h. $\mathcal{IS} \models_{ITL^+} \varphi$, wenn für alle $\mathcal{J} \in \mathcal{IS}$ mit $0 \in \mathcal{J}$ gilt: $\mathcal{IS}, \mathcal{J}, 0 \models_{ITL^+} \varphi$. Dann gilt φ in dem durch $\mathcal{IS}$ beschriebenen System. Eine Formel ist gültig, $\models_{ITL^+} \psi$, wenn für alle $\mathcal{IS} \subseteq J(IV)$ gilt: $\mathcal{IS} \models_{ITL^+} \varphi$. Wie man leicht sieht, ist damit die Gültigkeit von ITL-Formeln nach der Definition aus Abschnitt 5.2 äquivalent zu obiger Definition, die an ein Übergangssystem $\mathcal{IS}$ gebunden ist. Für alle Formeln $\varphi \in Fma_{ITL}$ gilt: $\models_{ITL^+} \varphi \Leftrightarrow \forall\, \mathcal{IS} \subseteq J(IV)$: $\mathcal{IS} \models_{ITL^+} \varphi \Leftrightarrow \forall\ \mathcal{IS} \subseteq J(IV)$, $\mathcal{J}$ mit $\mathcal{J} \in \mathcal{IS}$, $0 \in \mathcal{J}$: $\mathcal{IS}$, $\mathcal{J}$, $0 \models_{ITL^+} \varphi \ \overset{*}{\Leftrightarrow}\ \forall\ \mathcal{J}$: $\mathcal{J} \models_{ITL} \varphi \Leftrightarrow\ \models_{ITL} \varphi$. Im Schritt (*) gehen wir zur Semantikdefinition aus Abschnitt 5.2 über, in der die Intervalle so definiert sind, dass sie immer bei 0 beginnen, also $0 \in \mathcal{J}$. Sowohl in ITL als auch in ITL$^+$ sprechen wir bei Gültigkeit über alle Interpretationen (in ITL$^+$ über alle Übergangssysteme $\mathcal{IS}$ und mit „für alle $\mathcal{J} \in \mathcal{IS}$“ auch über alle Interpretationen).

Bemerkung Wir präsentieren hier eine sehr allgemeine Definition für eine Temporallogik mit Vergangenheit und CTL-Pfadquantoren. Systeme werden durch Mengen von Interpretationen definiert, deren Interpretationen beliebig weit in die Vergangenheit zurückreichen können. So kann ein System beobachtet werden, das vor beliebig langer Zeit gestartet wurde. Wir beginnen unsere Beobachtung jedoch immer zum Zeitpunkt 0. Deshalb fordern wir für die Gültigkeit von Formeln, dass nur Interpretationen $\mathcal{J}$ mit $0 \in \mathcal{J}$ betrachtet werden.

Konkerete Systembeschreibungen schränken die allgemeine Definition häufig ein. So sind typische Einschränkungen bei Statecharts oder parallelen Programmen, dass sie zu einem definierten Startzeitpunkt 0 beginnen, d h. für alle $\mathcal{J} \in \mathcal{IS}$ gilt: $dom(\mathcal{J}) = [0, b]$.

8.2 Vergleich der FTA-Semantiken

In diesem Abschnitt vergleichen wir verschiedene FTA-Semantiken. Für dynamische Systeme ist es notwendig, Ursache/Wirkungs-Beziehungen zu betrachten und wir vergleichen deshalb die Formalisierungen von Hansen et al. [HRS94] im DC, von Bruns und Anderson [BA93] im μ-Kalkül, die Semantik aus Abschnitt 5.3 in ITL und die CTL-Formalisierung aus Abschnitt 7.2. Alle vier Formalisierungen betrachten Ursache/Wirkungs-Beziehungen in Fehlerbäumen.

Aus der Literatur ist noch die Formalisierung von Górski und Wardziński [GW95] bekannt, die Ursache/Wirkungs-Beziehungen betrachtet. Sie setzt aber die explizite Beschreibung solcher Beziehungen im Systemmodell voraus. Dies ist in herkömmlichen Beschreibungssprachen für dynamische Systeme, wie z. B. Statecharts, parallelen Programmen oder Temporallogik, nicht möglich. Deshalb kann die FTA-Formalisierung von Górski und Wardziński nicht für solche Modellierungssprachen eingesetzt werden und wir betrachten sie hier nicht weiter (Górski gibt in [Gór94] eine Modellierungssprache mit expliziter Beschreibungsmöglichkeit von Ursache/Wirkungs-Beziehungen an). Auch die klassische, boolesche Interpretation der Fehlerbaumgatter ist für die Beschreibung dynamischer Systeme nicht ausreichend, da sie keine Ursache/Wirkungs-Beziehungen beschreiben kann.

Im Folgenden betrachten wir die Formalisierung der Ursache/Wirkungs-Beziehung in den verschiedenen Semantiken, die durch Ursache/Wirkungs-Gatter zwischen den Unterereignissen und dem entsprechenden Ausgangsereignis beschrieben werden. Wir weisen nochmals darauf hin, dass Ursachen und Wirkungen aus Fehlerbäumen Ereignisse sind. Ereignisse werden als temporallogische Formeln über dem formalen Modell definiert. Wir nennen ein Ereignis punktuelles Ereignis, wenn es durch die Untermenge der prädikatenlogischen Formeln spezifiziert werden kann, sonst temporales Ereignis. Die Begriffe Ereignis, Ursache und Wirkung verwenden wir häufig auch synonym für die entsprechende Formel. Für den Vergleich der Formalisierungen ist der Typ eines Gatters nicht relevant und wir kürzen Ursachen $\varphi_1 \wedge \ldots \wedge \varphi_n$ für *UW-Und*- bzw. $\varphi_1 \vee \ldots \vee \varphi_n$ für *UW-Oder*-Gatter mit φ ab und schreiben beispielsweise statt $\neg(\neg \diamondsuit_{x} (\varphi_1 \wedge \ldots \wedge \varphi_n) \mathbin{;} \diamondsuit_{i}\, \psi)$ nur $\neg(\neg \diamondsuit_{x}\, \varphi \mathbin{;} \diamondsuit_{i}\, \psi)$.

8.2.1 FTA-Formalisierung von Hansen et al.

Hansen et al. [HRS94] schlagen eine Definition der FTA-Semantik im Duration Calculus (DC, [CHR91]) vor. Der DC ist eine Erweiterung der ITL, dem ein kontinuierliches Zeitmodell zugrunde liegt. Im DC werden die Operatoren aus der ITL über einem kontinuierlichen Zeitmodell und zusätzlich quantifizierende Operatoren definiert. Der *Integraloperator* $\int$ bestimmt beispielsweise die Dauer kontinuierlicher Ereignisse. Die zusätzlichen DC-Operatoren und die kontinuierlichen Aspekte sind für die Definition der FTA-Semantik

jedoch nicht relevant. Interpretieren wir die verbleibenden Operatoren des DC wieder auf einem diskreten Zeitmodell, erhalten wir die ursprünglichen ITL-Operatoren und wir können so die Formalisierung von Hansen et al. in ITL$^+$ wie folgt ausgedrücken.

Definition 8.4 *Formalisierung im DC*

Für eine Ursache φ und eine Wirkung ψ gilt:

$$\boxed{\scriptstyle i}\ (\Diamond\ \psi \rightarrow \Diamond\ \varphi)$$

Diese Formalisierung fordert: wenn die Wirkung ψ eintritt, dann tritt auch die Ursache φ ein. Diese Bedingung muss auf allen initialen Pfaden erfüllt sein, also insbesondere auch auf dem kürzesten initialen Pfad, auf dem die Wirkung ψ eintritt. Deshalb muss die Ursache *vor* (oder zumindest gleichzeitig mit) der Wirkung ψ eintreten.

Problematisch ist diese Formalisierung, wenn temporale Ereignisse, z. B. „zehn Sekunden lang Bremsversagen“, auftreten. Dann lässt die DC-Formalisierung aus Definition 8.4 zu, dass die Ursache beginnt (und endet), während die Wirkung anhält. Dies wird durch das Zeitdiagramm in Abbildung 8.1 veranschaulicht. In der ersten Phase ist keine Wirkung

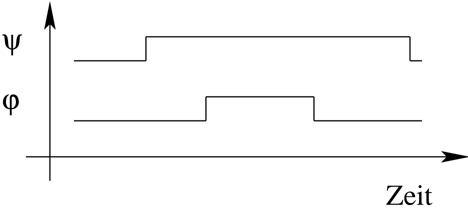


Abbildung 8.1: temporale Ursache φ und Wirkung ψ (I)

ψ (z. B. Bremsversagen) vorhanden, dann gilt ψ und schließlich wieder nicht. Die Ursache φ verhält sich analog, beginnt aber *nachdem* die Wirkung eintrat und endet zuvor.

Als Beispiel kann man sich die Wirkung „zehn Sekunden Bremsversagen“ und das Ereignis „fünf Sekunden Unterdruck im Bremsschlauch“ vorstellen. Beginnt der Druckabfall im Bremsschlauch erst nachdem das Bremsversagen eintritt, kann dies nicht das ursächliche Ereignis sein. Das Bremsversagen muss von einem anderen Ereignis hervorgerufen werden. Die DC-Semantik „akzeptiert“ jedoch dieses Szenario, d. h. die Bedingung aus Definition 8.4 kann nachgewiesen werden. Damit wird (fälschlicherweise) die Vollständigkeit der Ursache/Wirkungs-Beziehung nachgewiesen und die tatsächliche Ursache kann übersehen werden.

Diese Schwäche tritt nur bei temporalen Ereignissen auf. Beschränken wir uns bei der FTA auf punktuelle Ereignisse, so entspricht die DC-Semantik der Intuition, dass die Ursache *vor* der Wirkung eintritt. Die Ursache muss in jedem Intervall eintreten, in dem auch die Wirkung eintritt und so muss sie vor (oder gleichzeitig mit) der Wirkung eintreten und Ursache und Wirkung können bei punktuellen Ereignissen nicht überlappen.

8.2.2 FTA-Formalisierung von Bruns und Anderson

Bruns und Anderson [BA93] geben eine Semantikdefinition im μ-Kalkül [Koz83] mit Vergangenheit an. Diese lässt sich in ITL$^+$ äquivalent folgendermaßen formalisieren.

Definition 8.5 *Formalisierung im μ-Kalkül*

Für eine Ursache φ und eine Wirkung ψ gilt:
$\Box\ (\psi \rightarrow A \diamondminus \varphi)$

Diese Formalisierung fordert: wenn die Wirkung ψ eintritt, musste irgendwann zuvor die Ursache φ eintreten. Diese Bedingung muss für alle Pfade gelten, die zur Wirkung ψ führen. Die Semantik wurde ursprünglich nur für punktuelle Ereignisse formalisiert. Als mögliche Systemmodelle werden in [BA93] Zustandsübergangssystemen oder Mengen aus Sequenzen von Zuständen (Mengen von Pfaden) angegeben. Ereignisse sind darin atomare Aussagen über Mengen von Zuständen.

Wenn diese Semantik auf temporale Ereignisse übertragen wird, treten ähnliche Probleme wie bei der DC-Formalisierung nach Definition 8.4 auf. Die Formalisierung im μ-Kalkül fordert nur, dass die Ursache vor der Wirkung beginnt, nicht aber, dass sie beendet sein muss. So können temporale Ursachen und Wirkungen überlappen. Dies verdeutlicht das

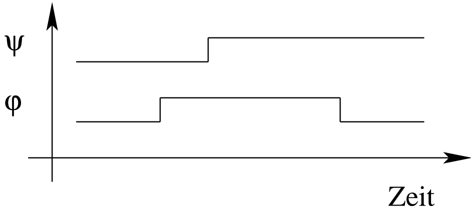


Abbildung 8.2: temporale Ursache φ und Wirkung ψ (II)

Zeitdiagramm in Abbildung 8.2. Als Beispielszenario kann eine „Kollision“ (Wirkung ψ) nur dann eintreten, wenn mindestens „zehn Sekunden Bremsversagen“ vorliegt. Tritt, wie im Zeitdiagramm, die Wirkung ψ schon nach 5 Sekunden ein, so kann nicht „zehn Sekunden Bremsversagen“ die Ursache φ sein. Es muss ein anderes Ereignis die Kollision verursacht haben. Jedoch lässt sich die Formel aus Definition 8.5 für dieses Szenario nachweisen. Damit kann die μ-Kalkül-Semantik nicht adäquat auf temporale Ereignisse erweitert werden.

Betrachtet man in den Fehlerbäumen nur punktuelle Ereignisse, entspricht die Semantik der gewünschten Ursache/Wirkungs-Beziehung, denn die Ursache beginnt und endet, da sie punktuell ist, vor der Wirkung.

8.2.3 FTA-Formalisierung in CTL

In Abschnitt 7.2 wurde die Formalisierung der Vollständigkeit von Fehlerbaumgattern in CTL für den automatischen Nachweis mit Modellprüfern vorgestellt.

Definition 8.6 *Formalisierung in CTL*

Für eine Ursache φ und eine Wirkung ψ gilt:
$A\ (\varphi \ \mathrm{P}\ \psi)$

Für alle Pfade eines Systems gilt, wenn in einem Zustand die Wirkung ψ eintritt, dann muss die Ursache φ zuvor oder gleichzeitig eingetreten sein. Analog zur μ-Kalkül-Semantik fordert die CTL-Formalisierung nur, dass die Ursache vor der Wirkung beginnt. Ursache und Wirkung werden auf dem selben Pfad ausgewertet (siehe Vergleich von P mit **;**), und deshalb kann sich auch bei der CTL-Formalisierung eine temporale Ursache mit der Wirkung überlappen. Verläufe wie im Zeitdiagramm in Abbildung 8.2 können fälschlicherweise als vollständig gezeigt werden. Für punktuelle Ereignisse ist die CTL-Vollständigkeitsdefinition jedoch ebenfalls korrekt.

CTL und Akzeptor-Automaten

Mit der Akzeptor-Automaten-Konstruktion aus Abschnitt 7.3 haben wir die Vollständigkeitsprüfung von Fehlerbäumen in CTL auch auf temporale Ereignisse erweitern können. Temporale Ereignisse werden dabei durch einen Automaten, der parallel zum Systemmodell läuft, beobachtet. Sobald ein temporales Ereignis vollständig beobachtet wurde, geht der Akzeptor-Automat in einen Endzustand über. Für den Vollständigkeitsnachweis der FTA-Gatter in CTL weisen wir nach, dass der Endzustand des entsprechenden Akzeptor-Automaten vor Eintreten der Wirkung erreicht wurde. Die Ursache trat damit vollständig vor der Wirkung ein. Durch diese Konstruktion ist es nicht mehr möglich, dass die Ursachen mit der Wirkung überlappen.

Es ist zu bemerken, dass die Wirkung eines Ursache/Wirkungs-Gatter *nicht* durch einen Akzeptor-Automaten beschrieben werden darf, wenn temporale Ursachen abgeschlossen sein sollen bevor die temporale Wirkung eintritt. Sobald die Wirkung ψ ein temporales Ereignis ist und ihr Eintreten durch einen Akzeptor-Automaten beobachtet wird, können sich Ursachen φ und Wirkung ψ wieder überschneiden. Der Akzeptor-Automat signalisiert immer das *Ende* eines temporalen Ereignisses im akzeptierenden Zustand. Die Wirkung ψ kann deshalb *vor* dem Eintreten der Ursache *beginnen*. Es ist also nicht gewährleistet, dass die Ursache vor Beginn der Wirkung beendet ist. Deshalb fordern wir für die Behandlung temporaler Ereignisse in CTL, dass nur die Ursachen mit Akzeptor-Automaten beschrieben werden. Temporale Wirkungen, wie n Zeitschritte φ beschreiben wir dann mit $EX_{[n]}\varphi$ (es genügt ein Pfad, auf dem die Wirkung auftritt). In Abschnitt 8.4.2 werden wir zeigen, dass die Akzeptor-Automaten-Konstruktion mit der CTL-Vollständigkeitsprüfung die ITL-Semantik für Fehlerbaumgatter erhält.

Bei der Modellprüfung des Fehlerbaums für den Drucktank im vorigen Kapitel konnten wir sowohl die Ursache als auch die Wirkung mit Akzeptor-Automaten beschreiben, da sich die Urachen und Wirkungen in diesem Beispiel überlappen. Die Ursache für die Wirkung „länger als 60 sek pumpen“ ist „Relais K_2 länger als 60 sek geschlossen“. Natürlicherweise ist das Relais geschlossen, während die Pumpe läuft. Hier muss nur gewährleistet werden, dass die Ursachen beendet sind, bevor die entsprechende Wirkung zuende ist.

8.2.4 FTA-Formalisierung in ITL

Für eine adäquate Beschreibung der Ursache/Wirkungs-Beziehung schlagen wir die ITL-Semantik aus Abschnitt 5.3 vor.[2] Sie benutzt den Chop-Operator „;" um ein Intervall in zwei Subintervalle zu zerlegen.

Definition 8.7 *Formalisierung in ITL*

Für eine Ursache φ und eine Wirkung ψ gilt:

$$\neg(\neg \Diamond_{\mathrm{x}}\, \varphi \;\mathbf{;}\; \Diamond_{\mathrm{i}}\, \psi)$$

Für die Abläufe eines Systems wird gefordert, dass es keinen Pfad geben kann, auf dem die Wirkung ψ eintritt, zuvor aber nicht irgendwann die Ursache φ. Durch die Zerlegung des Intervalls wird erreicht, dass die Ursache innerhalb eines Intervalls auftreten muss, welches vor Beginn der Wirkung abgeschlossen ist. Damit kann bei Ereignissen mit Dauer die Ursache nicht in die Wirkung hineinragen. Ursachen und Wirkungen können deshalb nicht überlappen. Aus diesem Grund und wegen des minimalen Schnittmengen-Theorems aus Abschnitt 5.4 ist dies eine adäquate Semantik für die Ursache/Wirkungs-Beziehung in Fehlerbäumen dynamischer Systeme, die sowohl punktuelle Ereignisse als auch temporale Ereignisse korrekt verknüpft.

8.2.5 Zusammenfassung

In den vorigen Teilabschnitten haben wir vier verschiedene FTA-Semantiken vorgestellt und miteinander verglichen. Dabei zeigten drei der vorgestellten Semantiken Schwächen bei temporalen Ereignissen. Nur die ITL-Semantik aus Abschnitt 5.3 beschreibt die Vollständigkeit der Gatter so, dass die Ursache/Wirkungs-Beziehung auch für temporale Ereignisse der Intuition entspricht. Die CTL-Formalisierung (siehe Abschnitt 8.2.3) wurde für den automatischen Vollständigkeitsnachweis mit CTL-Modellprüfern entwickelt. CTL ist aber nicht ausdrucksstark genug, um die Ursache/Wirkungs-Beziehung von temporalen Ereignissen korrekt auszudrücken. Deshalb haben wir in Abschnitt 7.2.2 zusätzlich eine Akzeptor-Automaten-Konstruktion angegeben, die auch für temporale Ereignisse die Vollständigkeitsprüfung mit CTL-Modellprüfern erlaubt. Die Korrektheit dieser Konstruktion betrachten wir in Abschnitt 8.4.2 genauer. Die obigen Semantiken sind in Abbildung 8.3 nochmals zusammengefasst. Für jedes Fehlerbaumgatter aus Abschnitt 5.1 wird die entsprechende Formalisierung für zwei Untereregnisse angegeben.

Bruns und Anderson formalisieren in [BA93] auch Zerlegungsgatter („immediate causality", $\Box\psi \leftrightarrow \varphi$). Für punktuelle Ereignisse sind $\Box$ und $\boxtimes$ äquivalent und deren Zerlegungsgatter-Formalisierung entspricht der ITL-Semantik, wenn wir darin die Korrektheit und Vollständigkeit zusammen betrachten (Korrektheit: $\boxtimes\varphi \rightarrow \psi$, Vollständigkeit: $\boxtimes\psi \rightarrow \varphi$). Für temporale Ereignisse genügt es allerdings nicht, die Äquivalenz zwischen Ursachen und Wirkungen auf allen Pfaden nachzuweisen, wie es Bruns und Anderson mit dem $\Box$

[2] Die Formalisierung kann in den DC übernommen werden, um kontinuierliche Systeme und Ereignisse mit Dauer (mit dem $\int$-Operator) zu beschreiben (siehe [Sch03]).

Vollst.	ITL	Hansen	Bruns	CTL
ψ / Z / φ_1 φ_2	$\boxtimes\ (\psi \rightarrow \varphi_1 \wedge \varphi_2)$	–	$\Box\ \psi \leftrightarrow \varphi_1 \wedge \varphi_2$	AG $(\psi \rightarrow \varphi_1 \wedge \varphi_2)$
ψ / Z / φ_1 φ_2	$\boxtimes\ (\psi \rightarrow \varphi_1 \vee \varphi_2)$	–	$\Box\ \psi \leftrightarrow \varphi_1 \vee \varphi_2$	AG $(\psi \rightarrow \varphi_1 \vee \varphi_2)$
ψ / Z – χ / φ	$\boxtimes\ (\psi \rightarrow \varphi \wedge \chi)$	–	–	AG $(\psi \rightarrow \varphi \wedge \chi)$
ψ / UW / φ_1 φ_2	$\neg(\neg\Diamond_{\times}\ (\varphi_1 \wedge \varphi_2)\ ;\ \Diamond_{i}\ \psi)$	$\boxed{i}\ (\Diamond_{\times}\ \psi \rightarrow$ $\Diamond_{\times}\ (\varphi_1 \wedge \varphi_2))$	$\Box\ (\psi \rightarrow$ A $\Diamond_{-}\ (\varphi_1 \wedge \varphi_2))$	A $((\varphi_1 \wedge \varphi_2)$ P $\psi)$
ψ / A / φ_1 φ_2	$\neg(\neg\Diamond_{\times}\ \varphi_1\ ;\ \Diamond_{i}\ \psi)$ $\wedge\ \neg(\neg\Diamond_{\times}\ \varphi_2\ ;\ \Diamond_{i}\ \psi)$	–	–	A $(\varphi_1$ P $\psi)$ $\wedge$ A $(\varphi_2$ P $\psi)$
ψ / UW / φ_1 φ_2	$\neg(\neg\Diamond_{\times}\ (\varphi_1 \vee \varphi_2)\ ;\ \Diamond_{i}\ \psi)$	$\boxed{i}\ (\Diamond_{\times}\ \psi \rightarrow$ $\Diamond_{\times}\ (\varphi_1 \vee \varphi_2))$	$\Box\ (\psi \rightarrow$ A $\Diamond_{-}\ (\varphi_1 \vee \varphi_2))$	A $((\varphi_1 \vee \varphi_2)$ P $\psi)$
ψ / UW – χ / φ	$\neg(\neg\Diamond_{\times}\ \varphi\ ;\ \Diamond_{i}\ \psi)$ $\wedge\ \neg(\neg\Diamond_{\times}\ \chi\ ;\ \Diamond_{i}\ \psi)$	$\boxed{i}\ (\Diamond_{\times}\ \psi \rightarrow$ $\Diamond_{\times}\ (\varphi \wedge \chi))$	–	A $(\varphi$ P $\psi)$ $\wedge$ A $(\chi$ P $\psi)$

Abbildung 8.3: Vollständigkeitsbedingungen für Fehlerbäume

Operator fordern, sondern es müssen alle Subintervalle ($\boxtimes$ Operator) betrachtet werden (siehe ITL-Semantik). Hansen et al. [HRS94] betrachten keine Zerlegungsgatter. Analog zu unserer ITL-Semantik geben sie eine Formalisierung von *Block*-Gattern an, die dem *UW-Und*-Gatter entspricht. Die Nebenbedingung wird dabei als ein Unterereignis betrachtet.

Obwohl Bruns und Anderson keine asynchronen *UW-Und*-Gatter aufführen und Hansen et al. diese sogar explizit ablehnen, haben Untersuchungen der FTA an verschiedenen Beispielen die Notwendigkeit dieser Gatter gezeigt. Auch wenn Ursachen zusammen ein Ereignis hervorrufen, müssen sie dazu nicht gleichzeitig auftreten. Deshalb wurde sowohl für die ITL- als auch für die CTL-Semantik das asynchrone *UW-Und*-Gatter formalisiert.

Wir haben in Abschnitt 5.3 auch Korrektheitsbedingungen für die Fehlerbaumgatter beschrieben. Bruns und Anderson haben ebenfalls Korrektheitsbedingungen formalisiert, die sie in [BA93] als „sufficient causality" ($\Box\ (\varphi \to \mathrm{A} \Diamond \psi)$) bezeichnen. Diese Korrektheitsbedingung fordert für jedes Auftreten einer Ursache, dass eine Wirkung folgen muss. Diese starke Forderung haben wir explizit abgelehnt. Nachdem eine Wirkung (z. B. Unfall) eingetreten ist, muss eine mögliche Ursache (z. B. Bremsausfall, der durch den Unfall entstanden ist) nicht nochmals die Wirkung verursachen (siehe Abschnitt 5.3.1, Seite 61).

8.3 Spezifikation der erweiterten ITL in KIV

Die ITL$^+$ wurde vollständig im Spezifikations- und Verifikationswerkzeug KIV [BRS$^+$00] algebraisch spezifiziert. Die gesamte strukturierte Spezifikation befindet sich im Anhang C. Der Entwicklungsgraph ist nochmals in Abbildung 8.4 abgebildet und gibt einen Überblick über die relevanten Teile der Spezifikation. Die Semantik einer Formel (Spezifikation *semantics*, Seite 237) wird über einer *structure* M definiert, die aus einer Menge von Interpretationen IS (Menge aller Intervalle, die ein Systemmodell beschreiben kann), einer aktuellen Interpretation I und einem aktuellem Zeitpunkt *now*, an dem das Intervall ausgewertet wird, besteht (IS entspricht $\mathcal{IS}$, I entspricht $\mathcal{J}$ und *now* entpricht t aus dem temporallogischen Rahmen aus Abschnitt 8.1).

Temporallogische Formeln sind analog zu der Semantik aus Abschnitt 8.1 spezifiziert. Als Basisoperatoren der ITL$^+$ wurden die Operatoren **;** (Spezifikation *itl-sem*, Seite 234), $\Diamond_i$, $\Diamond_x$, U (Spezifikation *ctl-sem*, Seite 232), S und A (Spezifikation *past-sem*, Seite 231) definiert. Darauf aufbauend wurden $\boxed{i}$, $\boxtimes$, P (Spezifikation *ctl-syn*, Seite 233) und $\Diamond_-$ (Spezifikation *past-syn* Seite 232) als Abkürzungen definiert. Dazu ist zu bemerken, dass die explizite Semantikdefinition der abgeleiteten Operatoren, wie sie im Lemma 5.1 und Lemma 8.1 angegeben sind, den entsprechenden abgeleiteten Definitionen entsprechen. Diese Äquivalenz wurde zur Validierung der ITL-Spezifikation in KIV formal nachgewiesen.

Der Entwicklungsgraph enthält außerdem die Spezifikation von Fehlerbäumen (Spezifikation *fta*, Seite 228) zum Nachweis des minimalen Schnittmengen-Theorems aus Abschnitt 5.4. Das Theorem 5.1 wurde über dieser Spezifikation formuliert und nachgewiesen.

Die Äquivalenzen der verschiedenen FTA-Semantiken für punktuelle Ereignisse wurde in der Fallstudie als Lemma der Spezifikation *fta-eq*, Seite 229, formuliert und nachgewiesen. Ein punktuelles Ereignis wir durch eine prädikatenlogische Formel beschrieben. Für den

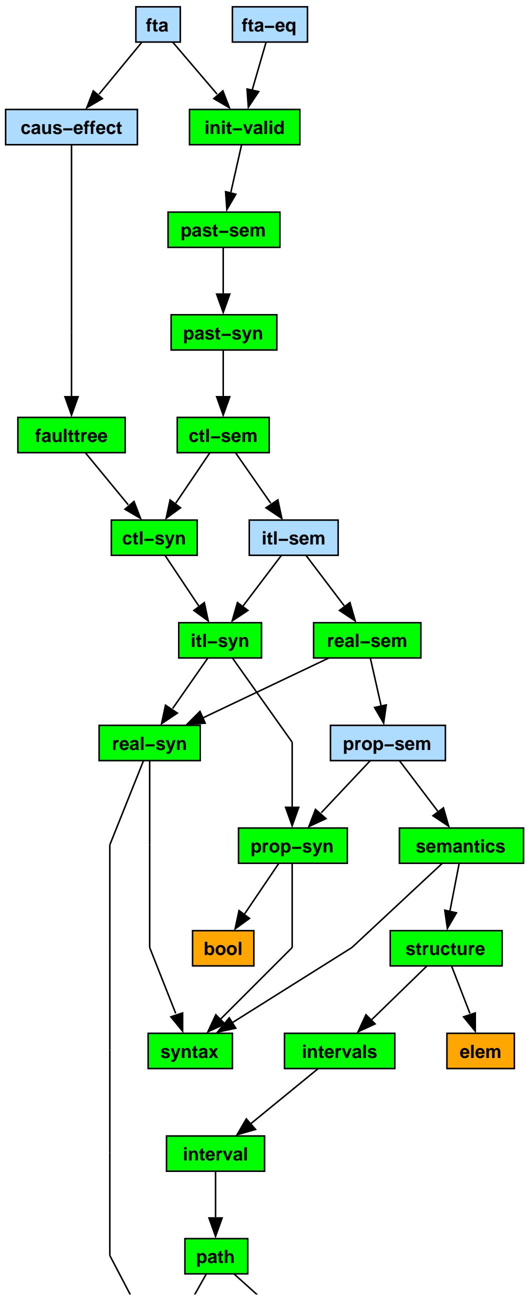


Abbildung 8.4: Entwicklungsgraph: Spezifikation von Temporallogik und Fehlerbäumen

Äquivalenzbeweis haben wir nicht die Formelmenge für Ereignisse auf prädikatenlogische Formeln eingeschränkt, sondern eine etwas allgemeinere Charakterisierung für punktuelle Ereignisse gewählt. Eine prädikatenlogische Formel $\Phi \in Fma_{PL}$ wird über dem aktuellen Zustand ausgewertet, d. h. $\mathcal{IS}, \mathcal{J}, t \models_{ITL^+} \Phi :\Leftrightarrow \mathcal{J}(t) \models_{PL} \Phi$. Für die Auswertung ist also nur die Belegung $\sigma = \mathcal{J}(t)$ zum aktuellen Zeitpunkt relevant. Wir definieren ein Prädikat $point(\varphi)$, das genau dann gilt, wenn die φ Auswertung nur vom aktuellen Zeitpunkt abhängt. Für alle prädikatenlogischen Formeln $\Phi \in Fma_{PL}$ gilt damit $point(\Phi)$.

Definition 8.8 *punktuelles Ereignis*

Sei $\varphi \in \mathrm{Fma}_{ITL^+}$ eine Formel der erweiterten ITL, $\mathcal{IS}$ und $\mathcal{IS}'$ eine Menge von Interpretationen $\mathcal{IS}$, $\mathcal{IS}' \in J(IV)$, dann gilt

$point(\varphi)$ genau dann, wenn

für alle $\mathcal{IS}$, $\mathcal{IS}'$ mit $\mathcal{J} \in \mathcal{IS}$, $\mathcal{J}' \in \mathcal{IS}'$, $t \in \mathcal{J}$, $t \in \mathcal{J}'$ und $\mathcal{J}(t) = \mathcal{J}'(t)$ gilt:

$$\text{wenn } \mathcal{IS}, \mathcal{J}, t \models \varphi, \text{ dann } \mathcal{IS}', \mathcal{J}', t \models \varphi.$$

Die Definition besagt, dass die Auswertung von φ nur von der Belegung $\mathcal{J}(t)$ abhängt und sowohl von der Intervallstruktur $\mathcal{J}$ als auch von den konkreten Systemabläufen $\mathcal{IS}$ unabhängig ist. Für punktuelle Formeln φ mit $point(\varphi)$ und $a \leq t \leq c$ bedeutet dies, dass aus $\mathcal{IS}, \mathcal{J}|_{[a,c]}, t \models \varphi$ für alle $t \leq d \leq max(\mathcal{J})$ folgt $\mathcal{IS}, \mathcal{J}|_{[a,d]}, t \models \varphi$.

Die Definition von $point(\varphi)$ befindet sich in der Spezifikation *fta-eq*. Die gesamte strukturierte Spezifikation besteht aus 28 einzelnen Spezifikationen (+ 10 Bibliotheksspezifikationen) mit ca. 748 Zeilen Spezifikation, 321 Axiomen für 121 verschiedene Operatoren. Für das Projekt, mit einem Gesamtaufwand von ca. 10 Tagen, mussten ca. 30 Hilfstheoreme bewiesen werden.

8.4 Äquivalenz der FTA-Semantiken

8.4.1 Punktuelle Ereignisse

Für die semantische Äquivalenz der FTA-Semantiken für punktuelle Ereignisse gilt folgendes Theorem.

Theorem 8.1 *Äquivalenz der FTA-Semantiken*
Für jede Menge von Intervallen $\mathcal{IS} \in J(IV)$ und punktuelle Formeln φ und ψ mit $point(\varphi)$ bzw. $point(\psi)$ gilt:

1. $\mathcal{IS} \models \neg(\neg \Diamond_{\mathrm{x}} \varphi \ ; \ \Diamond_{\mathrm{i}} \psi) \Leftrightarrow \mathcal{IS} \models \Box_{\mathrm{i}}(\Diamond_{\mathrm{x}} \psi \rightarrow \Diamond_{\mathrm{x}} \varphi)$
2. $\mathcal{IS} \models \neg(\neg \Diamond_{\mathrm{x}} \varphi \ ; \ \Diamond_{\mathrm{i}} \psi) \Leftrightarrow \mathcal{IS} \models A(\varphi \ \mathrm{P} \ \psi)$
3. *wenn die Menge $\mathcal{IS}$ nur solche Abläufe hat, so dass für alle $\mathcal{J} \in \mathcal{IS}$ mit $0 \in \mathcal{J}$ $min(\mathcal{J}) = 0$, dann gilt:*

$$\mathcal{IS} \models \neg(\neg \Diamond_{\mathrm{x}} \varphi \ ; \ \Diamond_{\mathrm{i}} \psi) \Leftrightarrow \mathcal{IS} \models \Box(\psi \rightarrow A \Diamond_{-} \varphi)$$

Ein interessantes Detail des 3. Teils des Theorems ist, dass hier die Äquivalenz nur für Interpretationen $\mathcal{J}$ mit unterer Grenze beim Zeitpunkt 0 gilt ($min(\mathcal{J}) = 0$). Beginnen die Pfade bevor die Beobachtung startet, kann sich der Vergangenheitsoperator $\Diamond\!\!\!\!{\scriptstyle i}$ auf einen Zeitpunkt vor dem Start der Beobachtung beziehen. Dies ist mit den anderen Temporaloperatoren nicht möglich und die Äquivalenz würde nicht gelten. Üblicherweise betrachten wir auch nur Systemmodelle, die im Initialzustand beginnen. Wie schon bemerkt, starten Statechart-Modelle und parallele Programme immer zum Zeitpunkt 0.

Der Beweis des Theorems 8.1 wurde im KIV-System geführt. Die interaktiven Beweise sind in ihrer Detailtreue jedoch sehr schwer übersichtlich zu präsentieren. Deshalb geben wir hier einen äquivalenten mathematischen Beweis an. Das Beweisprinzip beruht auf dem Ersetzen der Temporaloperatoren durch ihre semantische Definitionen, bis die Ursachen bzw. Wirkungen ohne führenden Operator auftreten. Dann wird die Äquivalenz dieser Formel unter der Voraussetzung, dass Ursachen und Wirkungen punktuell sind, gezeigt.

Beweis 8.1 *Äquivalenz der FTA-Semantiken*
Im Folgenden betrachten wir für Pfandmengen $\mathcal{IS}$ nur diejenigen Ausführungspfade $\mathcal{J} \in \mathcal{IS}$ in denen der Startzeitpunkt 0 enthalten ist. a und b bezeichnen $min(\mathcal{J})$ bzw. $max(\mathcal{J})$, die untere und obere Schranke von $\mathcal{J}$ (analog a', b' für $\mathcal{J}'$).

Für alle drei Teilbeweise ist es notwendig, die ITL-Formalisierung zu betrachten. Setzen wir darin zuerst die Semantikdefinitionen ein.

$$\mathcal{IS} \models \neg(\neg\Diamond\!\!\!\!\times\varphi \;\boldsymbol{;}\; \Diamond\!\!\!\!{\scriptstyle i}\,\psi)$$

$$\overset{\models}{\Leftrightarrow} \quad \textit{für alle } \mathcal{J} \in \mathcal{IS} : \mathcal{IS}, \mathcal{J}, 0 \models \neg(\neg\Diamond\!\!\!\!\times\varphi \;\boldsymbol{;}\; \Diamond\!\!\!\!{\scriptstyle i}\,\psi)$$

$$\overset{\neg}{\Leftrightarrow} \quad \textit{es ex. kein } \mathcal{J} \in \mathcal{IS} : \mathcal{IS}, \mathcal{J}, 0 \models (\neg\Diamond\!\!\!\!\times\varphi \;\boldsymbol{;}\; \Diamond\!\!\!\!{\scriptstyle i}\,\psi)$$

$$\overset{\boldsymbol{;}}{\Leftrightarrow} \quad \textit{es ex. kein } \mathcal{J} \in \mathcal{IS}, \textit{ so dass ein } c \neq \infty \textit{ ex.: } 0 \leq c \leq b, \qquad (1)$$
$$\mathcal{IS}, \mathcal{J}\,|_{[a,c]}, 0 \models \neg\Diamond\!\!\!\!\times\varphi \textit{ und } \mathcal{IS}, \mathcal{J}, c \models \Diamond\!\!\!\!{\scriptstyle i}\,\psi$$

$$\overset{\Diamond\!\!\!\!{\scriptstyle i}}{\Leftrightarrow} \quad \textit{es ex. kein } \mathcal{J} \in \mathcal{IS}, \textit{ so dass ein } c, c' \neq \infty \textit{ ex: } 0 \leq c \leq c' \leq b,$$
$$\mathcal{IS}, \mathcal{J}\,|_{[a,c]}, 0 \models \neg\Diamond\!\!\!\!\times\varphi \textit{ und } \mathcal{IS}, \mathcal{J}\,|_{[a,c']}, c \models \psi$$

$$\overset{point(\psi)}{\Leftrightarrow} \quad \textit{es ex. kein } \mathcal{J} \in \mathcal{IS}, \textit{ so dass ein } c \neq \infty \textit{ ex.: } 0 \leq c \leq b,$$
$$\mathcal{IS}, \mathcal{J}\,|_{[a,c]}, 0 \models \neg\Diamond\!\!\!\!\times\varphi \textit{ und } \mathcal{IS}, \mathcal{J}, c \models \psi$$

$$\overset{\neg,\Diamond\!\!\!\!\times}{\Leftrightarrow} \quad \textit{es ex. kein } \mathcal{J} \in \mathcal{IS}, \textit{ so dass ein } c \neq \infty \textit{ ex.: } 0 \leq c \leq b,$$
$$\textit{für alle } c', c'', 0 \leq c' \leq c'' \leq c : \; \mathcal{IS}, \mathcal{J}\,|_{[a,c'']}, c' \models \neg\varphi \textit{ und}$$
$$\mathcal{IS}, \mathcal{J}, c \models \psi$$

$$\overset{point(\varphi)}{\Leftrightarrow} \quad \textit{es ex. kein } \mathcal{J} \in \mathcal{IS}, \textit{ so dass ein } c \neq \infty \textit{ ex.: } 0 \leq c \leq b, \qquad (2)$$
$$\textit{für alle } c', 0 \leq c' \leq c : \; \mathcal{IS}, \mathcal{J}\,|_{[a,c]}, c' \models \neg\varphi \textit{ und}$$
$$\mathcal{IS}, \mathcal{J}, c \models \psi$$

Zuerst wird die Definition der Gültigkeit einer Formel unter einer Spezifikation eingesetzt. Danach werden die Operatoren $\neg$, $\boldsymbol{;}$ und $\Diamond\!\!\!\!{\scriptstyle i}$ abgewickelt. Da ψ ein punktuelles Ereignis ist

und damit die Gültigkeit von ψ nur vom aktuellen Zeitpunkt c abhängt, kann im folgenden Schritt die obere Grenze des Intervalls $\mathcal{J} \mid_{[a,c']}$ auch auf $max(\mathcal{J}) = b$ gesetzt werden (point(ψ)). Nun werden die Definitionen von $\neg$ und $\Diamond_{\times}$ eingesetzt und im letzten Schritt wird die Voraussetzung, dass φ ein punktuelles Ereignis ist, ausgenutzt, um wiederum die obere Intervallgrenze abzuändern (c'' wird durch c ersetzt). Die abgeleiteten Gleichungen (1) und (2) gehen in die folgenden Beweise ein.

ITL-Formalisierung $\Leftrightarrow$ DC-Formalisierung

Für den Äquivalenznachweis wird auch die DC-Formalisierung abgewickelt:

$$
\begin{aligned}
& \mathcal{IS} \;\models \boxed{\scriptstyle i}(\Diamond_{\times}\psi \rightarrow \Diamond_{\times}\varphi) \\
\overset{\models}{\Leftrightarrow}\quad & \textit{für alle } \mathcal{J} \in \mathcal{IS} : \mathcal{IS}, \mathcal{J}, 0 \;\models \boxed{\scriptstyle i}(\Diamond_{\times}\psi \rightarrow \Diamond_{\times}\varphi) \\
\overset{\boxed{\scriptstyle i}}{\Leftrightarrow}\quad & \textit{für alle } \mathcal{J} \in \mathcal{IS}, t \neq \infty, 0 \leq t \leq b : \mathcal{IS}, \mathcal{J} \mid_{[a,t]}, 0 \;\models \Diamond_{\times}\psi \rightarrow \Diamond_{\times}\varphi \\
\overset{\rightarrow}{\Leftrightarrow}\quad & \textit{für alle } \mathcal{J} \in \mathcal{IS}, t \neq \infty, 0 \leq t \leq b : \mathcal{IS}, \mathcal{J} \mid_{[a,t]}, 0 \;\models \neg(\Diamond_{\times}\psi \wedge \neg\Diamond_{\times}\varphi) \\
\overset{\neg,\wedge}{\Leftrightarrow}\quad & \textit{es ex. kein } \mathcal{J} \in \mathcal{IS}, t \neq \infty, 0 \leq t \leq b : \\
& \quad \mathcal{IS}, \mathcal{J} \mid_{[a,t]}, 0 \;\models \Diamond_{\times}\psi \textit{ und } \mathcal{IS}, \mathcal{J} \mid_{[a,t]}, 0 \;\models \neg\Diamond_{\times}\varphi \\
\overset{\Diamond_{\times}}{\Leftrightarrow}\quad & \textit{es ex. kein } \mathcal{J} \in \mathcal{IS}, t \neq \infty, 0 \leq t \leq b : \\
& \quad \mathcal{IS}, \mathcal{J} \mid_{[a,t]}, 0 \;\models \neg\Diamond_{\times}\varphi \textit{ und} \\
& \quad \textit{ex. } t', t'' \textit{ mit } 0 \leq t' \leq t'' \leq t : \mathcal{IS}, \mathcal{J} \mid_{[a,t'']}, t' \;\models \psi \\
\overset{point(\psi)}{\Leftrightarrow}\quad & \textit{es ex. kein } \mathcal{J} \in \mathcal{IS}, t \neq \infty, 0 \leq t : \\
& \quad \mathcal{IS}, \mathcal{J} \mid_{[a,t]}, 0 \;\models \neg\Diamond_{\times}\varphi \textit{ und} \\
& \quad \textit{ex. } t' \textit{ mit } 0 \leq t' \leq t : \; \mathcal{IS}, \mathcal{J}, t' \;\models \psi \\
\overset{\neg\Diamond_{\times}}{\Leftrightarrow}\quad & \textit{es ex. kein } \mathcal{J} \in \mathcal{IS}, t \neq \infty, 0 \leq t \leq b \textit{ und} \\
& \quad \textit{für alle } t'', t''', 0 \leq t'' \leq t''' \leq t : \; \mathcal{IS}, \mathcal{J} \mid_{[a,t''']}, t'' \;\models \neg\varphi \textit{ und} \\
& \quad \textit{ex. } t' \textit{ mit } 0 \leq t' \leq t : \; \mathcal{IS}, \mathcal{J}, t' \;\models \psi \\
\overset{point(\varphi)}{\Leftrightarrow}\quad & \textit{es ex. kein } \mathcal{J} \in \mathcal{IS}, t \neq \infty, 0 \leq t \leq b \textit{ und} \qquad (3) \\
& \quad \textit{für alle } t'', 0 \leq t'' \leq t : \; \mathcal{IS}, \mathcal{J} \mid_{[a,t]}, t'' \;\models \neg\varphi \textit{ und} \\
& \quad \textit{ex. } t' \textit{ mit } 0 \leq t' \leq t : \; \mathcal{IS}, \mathcal{J}, t' \;\models \psi
\end{aligned}
$$

Analog zur ITL-Formalisierung werden die Operatoren ersetzt und an zwei Stellen die Voraussetzung, dass φ und ψ punktuell sind, ausgenutzt um die obere Intervallgrenze abzuändern. Im vorletzten Schritt haben wir zwei Schritte zusammengefasst und $\neg \Diamond_{\times} \varphi$ durch $\boxtimes \neg \varphi$ ersetzt und $\boxtimes$ sofort eingesetzt.

Um den Teilbeweis zu schließen, müssen die verbleibenden Variablen, die Zeitpunkte im Systemablauf darstellen, entsprechend gesetzt werden.

(2) $\Rightarrow$ (3) *setze $t := c,\; t' := c,\; t'' := c'$*

(2) $\Leftarrow$ (3) *da $t' \leq t$ folgt aus (3):*

es ex. kein $\mathcal{J} \in \mathcal{IS}, t' \neq \infty, a \leq t' \leq b$ und
für alle $t'', 0 \leq t'' \leq t'$: $\mathcal{IS}, \mathcal{J} \mid_{[a,t']}, t'' \models \neg\varphi$ und
$\mathcal{IS}, \mathcal{J}, t' \models \psi$
setze $c := t'$, $c' := t''$

ITL-Formalisierung $\Leftrightarrow$ μ-Kalkül Formalisierung

Nach dem gleichen Beweisprinzip wickeln wir die Formel des μ-Kalküls ab:

$$\mathcal{IS} \models (\Box\psi \rightarrow A\diamond\varphi)$$
$\overset{\models}{\Leftrightarrow}$ *für alle $\mathcal{J} \in \mathcal{IS}$: $\mathcal{IS}, \mathcal{J}, 0 \models \Box(\psi \rightarrow A\diamond\varphi)$*
$\overset{\Box}{\Leftrightarrow}$ *für alle $\mathcal{J} \in \mathcal{IS}, t \neq \infty, 0 \leq t \leq b$: $\mathcal{IS}, \mathcal{J}, t \models \psi \rightarrow A\diamond\varphi$*
$\overset{\rightarrow,\neg,\wedge}{\Leftrightarrow}$ *es gibt kein $\mathcal{J} \in \mathcal{IS}, t \neq \infty, 0 \leq t \leq b$:*
$\mathcal{IS}, \mathcal{J}, t \models \psi$ und $\mathcal{IS}, \mathcal{J}, t \models \neg A\diamond\varphi$
$\overset{\neg A}{\Leftrightarrow}$ *es gibt kein $\mathcal{J} \in \mathcal{IS}, t \neq \infty, 0 \leq t \leq b$:*
$\mathcal{IS}, \mathcal{J}, t \models \psi$ und es ex. ein $\mathcal{J}' \in \mathcal{IS}, t \in \mathcal{J}', \mathcal{J}'(t) = \mathcal{J}(t)$:
$\mathcal{IS}, \mathcal{J}', t \models \neg\diamond\varphi$
$\overset{\neg\diamond}{\Leftrightarrow}$ *es gibt kein $\mathcal{J} \in \mathcal{IS}, t \neq \infty, 0 \leq t \leq b$:* (4)
$\mathcal{IS}, \mathcal{J}, t \models \psi$ und
es ex. ein $\mathcal{J}' \in \mathcal{IS}, t \in \mathcal{J}', \mathcal{J}'(t) = \mathcal{J}(t)$, dass für alle $t' \neq -\infty, a' \leq t' \leq t$:
$\mathcal{IS}, \mathcal{J}', t' \models \neg\varphi$

Wieder setzen wir die Definitionen der Operatoren ein. Im 3. Schritt ($\rightarrow, \neg, \wedge$) wird die Implikation $\psi \rightarrow A \diamond \varphi$ durch $\neg$ ($\psi \wedge \neg$ $A \diamond \varphi$) ersetzt und die Negation und Konjunktion eingesetzt. Die folgenden Einsetzungen sind offensichtlich.

Für die Äquivalenz müssen die Zeitpunkte folgendermaßen gewählt werden:

(2) $\Rightarrow$ (4) *da $0 \neq -\infty$:*
setze $t := c, t' := c', \mathcal{J}' := \mathcal{J}$

(2) $\Leftarrow$ (4) *wegen der Voraussetzung $min(\mathcal{J}') = 0$ ist $a' = 0$:*
setze $c := t, c' := t', \mathcal{J} := \mathcal{J}'$

CTL-Formalisierung $\Leftrightarrow$ ITL-Formalisierung

Schließlich wird noch die CTL-Formalisierung abgewickelt:

$$\mathcal{IS} \models A(\varphi \text{ P } \psi)$$
$\overset{\models}{\Leftrightarrow}$ *für alle $\mathcal{J} \in \mathcal{IS}$: $\mathcal{IS}, \mathcal{J}, 0 \models A(\varphi \text{ P } \psi)$*

$$
\begin{array}{ll}
\overset{A}{\Leftrightarrow} & \textit{für alle } \mathcal{J} \in \mathcal{IS}, \mathcal{J}' \in \mathcal{IS}, \mathcal{J}'(0) = \mathcal{J}(0): \\
& \quad \mathcal{IS}, \mathcal{J}', 0 \models \varphi \ \mathrm{P} \ \psi \\
\overset{P}{\Leftrightarrow} & \textit{für alle } \mathcal{J} \in \mathcal{IS}, \mathcal{J}' \in \mathcal{IS}, \mathcal{J}'(0) = \mathcal{J}(0): \\
& \quad \mathcal{IS}, \mathcal{J}', 0 \models \neg\, (\neg\, \varphi \ \mathrm{U} \ \psi \wedge \neg\varphi) \vee \Box\neg\, \psi \\
\overset{\neg,\vee}{\Leftrightarrow} & \textit{es gibt kein } \mathcal{J} \in \mathcal{IS}, \mathcal{J}' \in \mathcal{IS}, \mathcal{J}'(0) = \mathcal{J}(0): \\
& \quad \mathcal{IS}, \mathcal{J}', 0 \models \neg\varphi \ \mathrm{U} \ \psi \wedge \neg\varphi \textit{ und} \\
& \quad \mathcal{IS}, \mathcal{J}', 0 \models \Diamond\psi \\
\overset{\mathrm{U},\Diamond}{\Leftrightarrow} & \textit{es gibt kein } \mathcal{J} \in \mathcal{IS}, \mathcal{J}' \in \mathcal{IS}, \mathcal{J}'(0) = \mathcal{J}(0): \\
& \quad\quad \textit{es ex. ein } t \neq \infty, 0 \leq t \leq b': \ \mathcal{IS}, \mathcal{J}', t \models \psi \wedge \neg\varphi \textit{ und} \\
& \quad\quad\quad \textit{für alle } t', 0 \leq t' < t: \ \mathcal{IS}, \mathcal{J}', t' \models \neg\varphi \\
& \textit{und es ex. ein } t'' \neq \infty, 0 \leq t'' \leq b': \ \mathcal{IS}, \mathcal{J}', t'' \models \psi
\end{array}
\qquad (5)
$$

Das korrekte Setzen der Variablen schließt den Beweis.

(2) $\Rightarrow$ **(5)** *es gibt kein* $\mathcal{J} \in \mathcal{IS}$:
es ex. ein $c \neq \infty$, $0 \leq c \leq b$ *so dass alle* $c', 0 \leq c' \leq c$
$\mathcal{IS}, \mathcal{J}, c' \models \neg\varphi$ *und* $\mathcal{IS}, \mathcal{J}, c \models \psi$
setze $t := c, t' := c', t'' := c, \mathcal{J}' := \mathcal{J}$

(2) $\Leftarrow$ **(5)** *abschwächen der Formel ergibt*
es gibt kein $\mathcal{J} \in \mathcal{IS}, \mathcal{J}' \in \mathcal{IS}, \mathcal{J}'(0) = \mathcal{J}(0)$:
es ex. ein $t \neq \infty, 0 \leq t \leq b'$: $\mathcal{IS}, \mathcal{J}', t \models \psi \wedge \neg\varphi$ *und*
für alle $t', 0 \leq t' < t$: $\mathcal{IS}, \mathcal{J}' \mid_{[a',t]}, t' \models \neg\varphi$
setze $c := t, c' := t', \mathcal{J} := \mathcal{J}'$

□

8.4.2 Die Akzeptor-Automaten-Konstruktion

Die Akzeptor-Automaten-Konstruktion benutzen wir bei der Modellprüfung von Gatterbedingungen von Fehlerbäumen, um temporale Ursachen so in CTL einzubetten, dass sie auf dem Pfad zur Wirkung eintreten und abgeschlossen sind, bevor die Wirkung eintritt. In Abschnitt 8.2.3 haben wir bereits erwähnt, dass wir die Wirkung eines Gatters nicht ebenfalls mit Akzeptor-Automaten beschreiben dürfen. Ansonsten können sich Ursachen und Wirkungen wieder überschneiden. Wir zeigen nun formal, dass die Akzeptor-Automaten-Konstruktion Ursachen tatsächlich so in CTL einbettet, dass sie vor der Wirkung abgeschlossen sind.

Pandya zeigt in [Pan01a] eine Einbettung linearer Temporaloperatoren in CTL. Er definiert die linearen Temporaloperatoren als eine Abwandlung von Duration Calculus-Operatoren über einer diskreten Zeitachse und nennt sie deshalb *Quantified Discrete-Time Duration Calculus* Formeln (QDDC-Formeln). Der QDDC definiert den Chop-Operartor

„;“ über endlichen Intervallen und leitet davon die üblichen temporallogischen Operatoren $\Box$, $\Diamond$, ... ab. Zusätzlich enthält der QDDC die quantifizierenden Operatoren aus Definition 5.3 (siehe Abschnitt 5.2). Wir haben die temporallogischen Operatoren bereits über endlichen und unendlichen Intervallen definiert und verzichten deshalb auf die explizite Definition der QDDC-Formeln. Sei Fma_{QDDC} die Menge der QDDC-Formeln, dann gilt: $Fma_{QDDC} \subset Fma_{ITL}$.

Pandya bezeichnet für $\varphi \in Fma_{QDDC}$ mit $CTL(\varphi)$ die Verwendung von φ innerhalb einer CTL-Formel (φ kommt atomar in einer CTL-Formel vor).

Lemma 8.2
Für eine Pfadmenge $\mathcal{IS} \in J(IV)$ und eine Formel $\varphi \in \mathrm{Fma}_{QDDC}$ gilt:

$$\mathcal{IS} \models CTL(\varphi) \Leftrightarrow \mathcal{IS} \mid\mid a_\varphi \models CTL(\varphi'),$$

wobei a_φ der Akzeptor-Automat für die QDDC-Formel φ ist und φ' eine boolscher Ausdruck über Variablen, der das Erreichen der Endzustände des Akzeptor-Automaten beschreibt.

Das Lemma 8.2 nach Pandya [Pan01a] zeigt die Äquivalenz zwischen der Akzeptor-Automaten-Konstruktion und der Einbettung von QDDC-Formeln in CTL. Wir müssen nun zeigen, dass die Einbettung einer in QDDC definierten Ursache in die CTL-Vollständigkeitsbedingung der ITL-Semantik von Fehlerbaumgattern entspricht.

Für eine Pfadmenge $\mathcal{IS} \in J(IV)$, einen Pfad $\mathcal{J} \in \mathcal{IS}$, einen Zeitpunkt $t \in \mathcal{J}$ und $a = min(\mathcal{J})$ wurde die Einbettung $[\varphi]_{DC}$ einer Formeln $\varphi \in Fma_{QDDC}$ in CTL folgendermaßen definiert [Pan01a]:

$$\mathcal{IS}, \mathcal{J}, t \models [\varphi]_{\mathrm{DC}} \quad :\Leftrightarrow \quad \mathcal{IS}, \mathcal{J}\,|_{[a,t]}, a \models \varphi$$

Eine eingebettete Formel $\varphi \in Fma_{QDDC}$ gilt also zu einem Zeitpunkt t in CTL, wenn φ auf dem Pfad vom Ausgangszustand bis zu t gilt (φ wird also immer auf einem endlichen Intervall ausgewertet).

Lemma 8.3
Für alle Pfadmengen $\mathcal{IS} \in J(IV)$ und $\varphi, \psi \in \mathrm{Fma}_{QDDC}$ gilt:

$$\mathcal{IS} \models \neg(\neg \Diamond_{\mathrm{x}} \varphi \ ; \ \Diamond_{\mathrm{i}} \psi) \Leftrightarrow \mathcal{IS} \models A([\Diamond_{\mathrm{x}} \varphi]_{DC} \ \mathrm{P} \ \Diamond_{\mathrm{i}} \psi)$$

Das Lemma 8.3 zeigt, dass jede temporale Ursache $\varphi \in Fma_{QDDC}$ so in CTL eingebetter werden kann, dass sie vollständig abgeschlossen ist, bevor die Wirkung eintritt. Mit dem Lemma 8.2 folgt für die Vollständigkeitsbedingung von Fehlerbäumen

$$\mathcal{IS} \models \mathrm{A}([\Diamond_{\mathrm{x}} \varphi]_{ITL} \ \mathrm{P} \ \Diamond_{\mathrm{i}} \psi) \Leftrightarrow \mathcal{IS} \mid\mid a_\varphi \models \mathrm{A}(\varphi' \ \mathrm{P} \ \Diamond_{\mathrm{i}} \psi)$$

für den Akzeptor-Automaten a_φ und den akzeptierenden Zuständen, die mit φ' beschrieben werden. φ' ist ein punktuelles Ereignis und die Akzeptor-Automaten-Konstruktion in CTL spiegelt die ITL-Formalisierung korrekt wider. Temporale Ursachen sind vor Beginn der Wirkung abgeschlossen.

Ziel dieser Einbettung ist es, den gesamten Vollständigkeitsnachweis für Fehlerbaumgatter in CTL zu führen. Jedoch ist die Wirkung $\diamonddot\psi$ keine CTL-Formel. Wie bereits erwähnt, können wir eine temporale Wirkung nicht mit Akzeptor-Automaten in CTL einbetten, sondern müssen eine adäquate CTL-Formalisierung für die Wirkung finden. Ausgehend von dem Zustand, in dem die temporale Wirkung beginnt, genügt ein Pfad, auf dem die Wirkung folgt. Wir können ein Ereignis mit Dauer (nach Górski) mit $EG_{[n]}\psi$ bzw. ein dauerhaftes Ereignis mit $EG\psi$ spezifizieren, wenn das Ereignis eine Wirkung beschreibt.

Mit den obigen Betrachtungen kann die Vollständigkeit von Fehlerbaumgattern mit temporalen Ereignissen in CTL nachgewiesen werden.

Für den Beweis von Lemma 8.2 verweisen wir auf Pandya [Pan01a], der die Einbettung von QDDC-Formeln in CTL entwickelt hat. Bleibt schließlich noch das Lemma 8.3 zu zeigen. Beide Vollständigkeitsbedingungen in Lemma 8.3 beinhalten, dass es einen Zeitpunkt t gibt, *vor* dem die Ursache und *ab* dem die Wirkung gilt. Dieser Punkt wird in ITL durch den **;**-Operator, in CTL durch den P-Operator beschrieben. In reiner CTL mit einer temporalen Ursache kann die Ursache jedoch über den Zeitpunkt t hinaus gelten. Genau dies wird durch die QDDC-Einbettung verhindert, denn die temporale Ursache muss auf dem Pfad gelten, der bei t endet.

Beweis 8.2 *Lemma 8.3*
Für den Beweis setzen wir die Definitionen der temporalen Operatoren und der Einbettung $[]_{DC}$ *von QDDC-Formeln in CTL ein.*

$$\mathcal{IS} \models A[\diamondtimes\varphi]_{DC} \text{ P } \diamonddot\psi$$

$$\overset{\models}{\Leftrightarrow} \quad \textit{für alle } \mathcal{J} \in \mathcal{IS}: \ \mathcal{IS}, \mathcal{J}, 0 \models A[\diamondtimes\varphi]_{DC} \text{ P } \diamonddot\psi$$

$$\overset{A}{\Leftrightarrow} \quad \textit{für alle } \mathcal{J} \in \mathcal{IS}, \mathcal{J}' \in \mathcal{IS}, \mathcal{J}'(0) = \mathcal{J}(0): \\ \mathcal{IS}, \mathcal{J}', 0 \models [\diamondtimes\varphi]_{DC} \text{ P } \diamonddot\psi$$

$$\overset{\mathrm{P}}{\Leftrightarrow} \quad \textit{für alle } \mathcal{J} \in \mathcal{IS}, \mathcal{J}' \in \mathcal{IS}, \mathcal{J}'(0) = \mathcal{J}(0): \\ \mathcal{IS}, \mathcal{J}, 0 \models \neg\ (\neg\ [\diamondtimes\varphi]_{DC} \text{ U } \diamonddot\psi \wedge \neg[\diamondtimes\varphi]_{DC}) \vee \Box\neg\ \diamonddot\psi$$

$$\overset{\neg,\vee}{\Leftrightarrow} \quad \textit{es gibt kein } \mathcal{J} \in \mathcal{IS}, \mathcal{J}' \in \mathcal{IS}, \mathcal{J}'(0) = \mathcal{J}(0): \\ \mathcal{IS}, \mathcal{J}', 0 \models \neg[\diamondtimes\varphi]_{DC} \text{ U } \diamonddot\psi \wedge \neg[\diamondtimes\varphi]_{DC} \textit{ und} \\ \mathcal{IS}, \mathcal{J}', 0 \models \Diamond\diamonddot\psi$$

$$\overset{\mathrm{U},\Diamond}{\Leftrightarrow} \quad \textit{es gibt kein } \mathcal{J} \in \mathcal{IS}, \mathcal{J}' \in \mathcal{IS}, \mathcal{J}'(0) = \mathcal{J}(0): \\ \textit{es ex. ein } t \neq \infty, 0 \leq t \leq b': \ \mathcal{IS}, \mathcal{J}', t \models \diamonddot\psi \wedge \neg[\diamondtimes\varphi]_{DC} \textit{ und} \\ \textit{für alle } t', 0 \leq t' < t: \ \mathcal{IS}, \mathcal{J}', t' \models \neg[\diamondtimes\varphi]_{DC} \\ \textit{und es ex. ein } t'' \neq \infty, 0 \leq t'' \leq b': \ \mathcal{IS}, \mathcal{J}', t'' \models \diamonddot\psi$$

$$\Leftrightarrow \quad \textit{es gibt kein } \mathcal{J} \in \mathcal{IS}, \mathcal{J}' \in \mathcal{IS}, \mathcal{J}'(0) = \mathcal{J}(0): \\ \textit{es ex. ein } t \neq \infty, 0 \leq t \leq b': \ \mathcal{IS}, \mathcal{J}', t \models \diamonddot\psi \textit{ und} \\ \textit{für alle } t', 0 \leq t' \leq t: \ \mathcal{IS}, \mathcal{J}', t' \models \neg[\diamondtimes\varphi]_{DC} \\ \textit{und es ex. ein } t'' \neq \infty, 0 \leq t'' \leq b': \ \mathcal{IS}, \mathcal{J}', t'' \models \diamonddot\psi$$

$$\overset{DC}{\Leftrightarrow} \quad \textit{es gibt kein } \mathcal{J} \in \mathcal{IS}, \mathcal{J}' \in \mathcal{IS}, \mathcal{J}'(0) = \mathcal{J}(0) :$$
$$\textit{es ex. ein } t \neq \infty, 0 \leq t \leq b' : \ \mathcal{IS}, \mathcal{J}', t \models \diamond\psi \textit{ und}$$
$$\mathcal{IS}, \mathcal{J}' \mid_{[a',t]}, 0 \models \neg\diamond\varphi \tag{6}$$
$$\textit{und es ex. ein } t'' \neq \infty, 0 \leq t'' \leq b' : \ \mathcal{IS}, \mathcal{J}', t'' \models \diamond\psi$$

(1) ⇒ (6) setze $t := c, a' := a, t'' := c, \mathcal{J}' := \mathcal{J}$

(1) ⇐ (6) Abschwächen der Formel ergibt
es gibt kein $\mathcal{J} \in \mathcal{IS}, \mathcal{J}' \in \mathcal{IS}, \mathcal{J}'(0) = \mathcal{J}(0)$:
es ex. ein $t \neq \infty, 0 \leq t \leq b'$: $\mathcal{IS}, \mathcal{J}', t \models \diamond\psi$ und
$\mathcal{IS}, \mathcal{J}' \mid_{[a',t]}, 0 \models \neg\diamond\varphi$
setze $c := t, a := a', \mathcal{J} := \mathcal{J}'$

□

KAPITEL 9

Methodik formaler FTA

Ein wesentlicher Punkt bei der Entwicklung sicherheitsrelevanter Systeme ist, die Sicherheitsaspekte „von Anfang an“ in das System zu integriert („Safety must be designed into a System“ [Lev95]). Nachträgliches Hinzufügen von (Sicherheits-) Funktionalität macht ein System komplizierter und damit fehleranfälliger. Die frühe sicherheitstechnische Analyse wird durch die formale FTA unterstützt. Formale Modelle können schon in den frühen Phasen des Entwicklungsprozesses softwarebasierter Systeme, den Analyse- und Designphasen, erstellt werden. Dies gilt gleichermaßen für die FTA.

Im Folgenden stellen wir ein Vorgehensmodell für die Durchführung einer formalen FTA vor. Die formale FTA sollte während der Analyse- und Designphase des herkömmlichen (Software-) Entwicklungsprozesses durchgeführt werden. Zur Veranschaulichung stellen wir die Methodik der formalen FTA an zwei Beispielen vor. Das erste Beispiel ist eine reale Industriefallstudie, die wir im Rahmen unserer Aktivitäten am Lehrstuhl zur sicherheitstechnischen Untersuchung softwarebasierter Systeme durchgeführt haben. Das zweite stammt aus einem Standardwerk zur FTA und vergleicht die herkömmliche FTA mit der formalen FTA.

9.1 Phasen einer formalen FTA

Die Modellentwicklung sicherheitskritischer Anwendungen mit formaler FTA wird in vier Phasen unterteilt (vgl. Abbildung 9.1). Wir nennen diese Phasen Anforderunganalyse, Sicherheitsanalyse, Integration und Auswertung. Diese vier Phasen werden im Wesentlichen nacheinander ausgeführt. Ergebnisse einer Phase fließen in die folgende Phase ein. Um von den Erkenntnissen der folgenden Phase zu profitieren, fließen aber auch Korrekturen und Änderungen, die in den späteren Phasen entstehen, in die vorhergehende Phase zurück. Diese Iterationen sind erforderlich, da in den fortgeschrittenen Phasen detailliertere Erkenntnisse des Systems erarbeitet werden, die Ergebnisse der vorherigen Phase beeinflussen können.

In der erste Phase wird die *Anforderungsanalyse* durchgeführt. Sie dient dazu, ein detailliertes Verständnis der Anwendungsdomäne zu erhalten. Die zweite Phase besteht aus

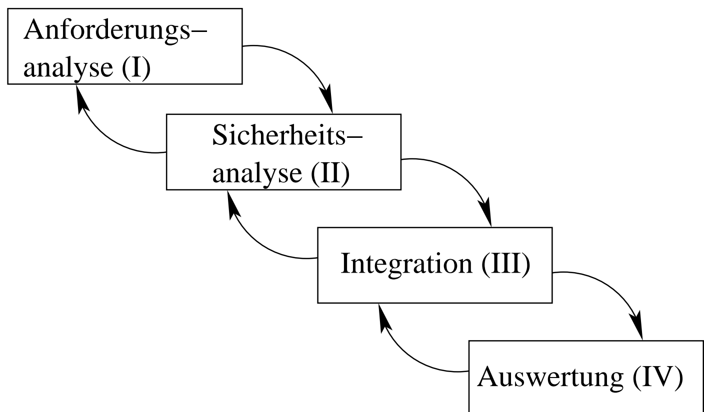


Abbildung 9.1: Phasen einer formalen FTA

der *Sicherheitsanalyse* des Systems mit formalen Methoden und der FTA. Diese beiden Techniken werden in dieser Phase noch getrennt angewendet, um die verschiedenen Betrachtungsweisen der Techniken – die funktionale Sicht und die Betrachtung der Sicherheit – zu erhalten. In der *Integration*sphase werden die Ergebnisse und Erkenntnisse der beiden Techniken ausgetauscht und in die jeweils andere Analysemethode eingebracht. Dies führt zu Verbesserungen im formalen Modell und in den Fehlerbäumen. Diese Phase enthält auch die Validation der Fehlerbäume, die auf zwei verschiedenen Stufen, lose bzw. eng gekoppelte formale FTA genannt, durchgeführt werden kann. Schließlich fasst die vierte Phase die Analyseergebnisse in einer *Auswertung* zusammen. Anhand von Abbildung 9.2 werden im Folgenden die einzelnen Arbeitsschritte der vier Phasen detailliert beschreiben.

9.1.1 Anforderungsanalyse (I)

Wie in jedem Entwicklungsprozess beginnt auch die Erstellung von hochsicheren Systemspezifikationen mit der Anforderungsanalyse. Für die formale FTA ist die Anforderungsanalyse von besonderer Bedeutung, da formale Modelle sehr detaillierte Informationen über das zu entwickelnde System benötigen. Werden solche Informationen erst später gewonnen, führt dies zu Änderungen in den formalen Modellen, die sich auf schon geführte Beweise auswirken bzw. neue Beweise erfordern. Änderungen sind deshalb sehr aufwendig (und teuer). Ein weiterer Punkt ist, dass formale Modelle häufig nicht als Grundlage für die Kommunikation mit den Auftraggeber geeignet sind, da sie schwer verständlich sind. Deshalb kann der Informationsaustausch mit dem Kunden meist nur in der Anforderungsanalyse stattfinden.

Die Verständigungsprobleme zwischen Auftraggeber und Entwickler werden durch die Erstellung von *semi-formalen* Spezifikationen (z. B. UML-Diagramme) und ausführbaren Prototypen zur Simulation zu reduzieren versucht. Graphische Spezifikationen mit definierter Semantik, wie z. B. Statecharts, sind für den Auftraggeber leichter zu verstehen und geben dem Entwickler eine präzise Grundlage für die weitere Entwicklung. Prototy-

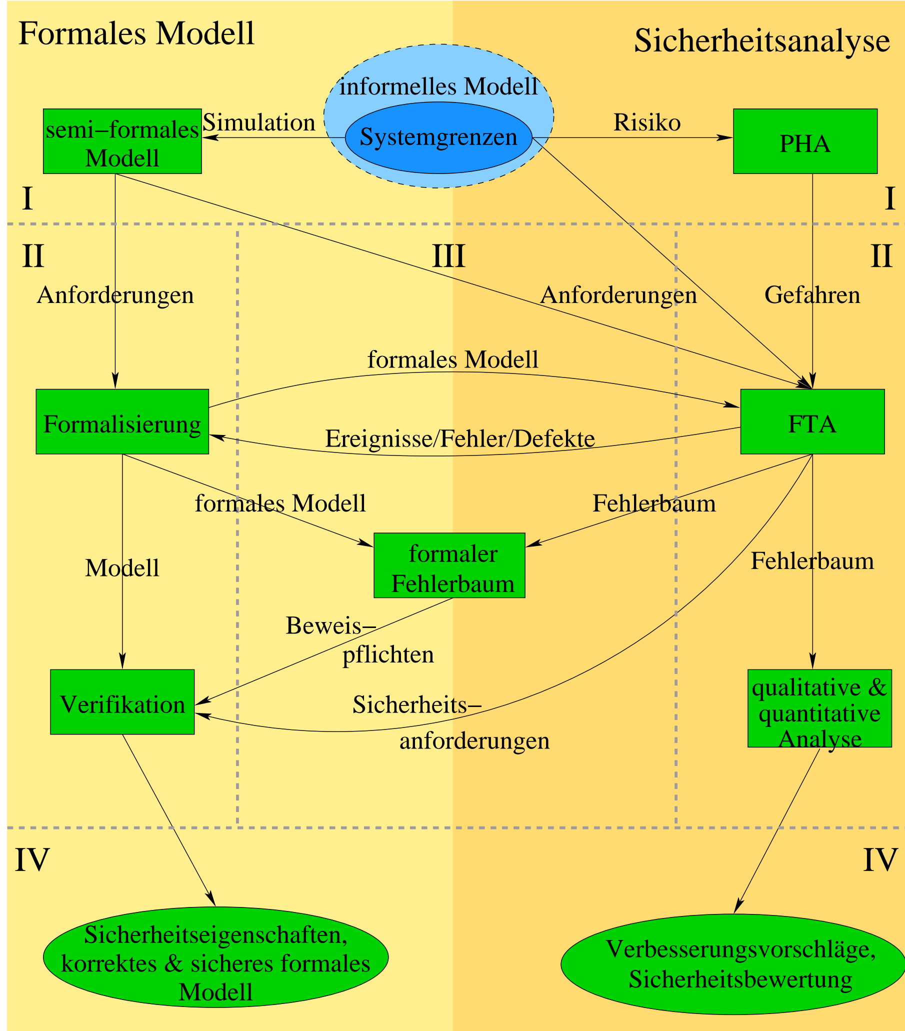


Abbildung 9.2: Vorgehensmodell

pen dienen dazu, verschiedene Szenarien durchzuspielen und die vom Kunden erwarteten Anforderungen mit den vom Entwickler implementierten Ergebnissen abzugleichen. Beide Techniken dienen dazu, Missverständnisse auszuräumen, klare *Systemanforderungen* zu definieren und stellen den Ausgangspunkt für die formale Sicherheitsanalyse in Phase II dar (siehe Abbildung 9.2).

Neben der präzisen Definition der Systemanforderungen müssen in der Anforderungsanalyse auch die *Systemgrenzen* genau festgelegt werden. Wird nur ein sicherheitskritischer Kern des Gesamtsystems betrachtet, beziehen sich die Systemanforderungen und -grenzen auf diesen Kern. Dies erfordert wiederum ein gemeinsames Verständnis der Systementwickler und des Kunden von der Systemfunktionalität und dem betrachteten Systemumfang.

Zweitens schlagen wir auf Seite der sicherheitstechnischen Analyse eine *preliminary*

hazard analysis (PHA, [Sto96]) vor, um eine vollständige Liste der Systemgefährdungen aufzustellen. Die PHA untersucht die erkannten Gefährdungen zusammen mit den funktionalen Anforderungen des Systems, um kritische Systemfunktionalitäten zu identifizieren und in ihrer Wichtigkeit zu klassifizieren. Die kritischen Systemfunktionalitäten müssen im funktionalen Design des Systems besonders sorgfältig betrachtet werden. Andererseits bilden die in der PHA identifizierten Gefährdungen den Ausgangspunkt für die FTA in der nächsten Phase.

9.1.2 Sicherheitsanalyse (II)

Die zweite Phase startet mit der getrennten Erstellung der FTA und des formalen Modells. Erfahrungen aus unseren bisherigen Sicherheitsanalysen zeigen, dass es sehr wichtig ist, die Formalisierung und die Sicherheitsanalyse unabhängig voneinander durchzuführen. Ansonsten ist die FTA zu sehr auf das formale Modell fixiert und verliert die Sicht auf mögliche Fehler, auf Defekte oder unspezifiziertes Verhalten. Eine unabdingbare Vorbedingung für diesen getrennten Ansatz ist jedoch, dass bei den Entwicklern der formalen Modelle und der FTA dasselbe Verständnis des Systems und der Systemgrenzen vorliegt. Deshalb betonen wir hier nochmals die Notwendigkeit einer sorgfältigen und detaillierten Anforderungsanalyse in Phase I.

Für die formale Behandlung des Systems muss zuerst eine Spezifikationssprache zur *Formalisierung* ausgewählt werden. Die Spezifikationssprache muss die einfache und adäquate Modellierung des Systems erlauben und von einer Verifikationsumgebung unterstützt werden, um allgemeine Sicherheitsbedingungen und Vollständigkeitsbedingungen der formalen FTA nachweisen zu können. Wir haben in dieser Arbeit Vollständigkeitsbedingungen für die FTA in ITL und in CTL formuliert, d. h. die Verifikationsumgebung muss entweder den Nachweis von ITL- oder von CTL-Bedingungen erlauben.

Funktionale Korrektheit Während der Formalisierung werden dann die informellen Anforderungen der Analysephase in die formale Spezifikationssprache der Verifikationsumgebung übersetzt und wir erhalten ein formales Systemmodell. Das formale Modell beschreibt das „Soll"-Verhalten des Systems. Ein Abgleich mit den informellen Anforderungen und dem semi-formalen Modell ist notwendig, um ein korrektes formales Modell zu erhalten. Um die funktionale Korrektheit des Modells nachzuweisen und um es zu validieren, werden gewünschte Systemeigenschaften formalisiert und verifiziert. Dieses Vorgehen nennt man *Validation durch Verifikation* des formalen Modells. Die Verifikation von Validierungseigenschaften verlangt meist ein iteratives Vorgehen, da fehlgeschlagene Beweisversuche auch zu Korrekturen des formalen Modells und zu neuen Beweisen führen können. Der Formalisierungsprozess endet schließlich in einem funktional korrekten Modell mit nachgewiesenen Sicherheitseigenschaften.

Informelle Sicherheitsanalyse Parallel zur Formalisierung untersucht die *FTA* alle Gefährdungen, die in der PHA identifiziert wurden. Dabei werden in dynamischen Syste-

men nicht nur die Gatter der klassischen FTA verwendet, sondern auch die Ursache/Wirkungs-Gatter, die wir in Abschnitt 5.1 beschrieben haben. Die Analyse ergibt eine Menge von Fehlerbäumen, deren minimale Schnittmengen der Ausgangspunkt für eine *qualitative Analyse* ist. Sind für die Basisereignisse die Ausfallwahrscheinlichkeiten bekannt, kann auch eine *quantitative* Bewertung der Systemsicherheit durchgeführt werden. Die quantitative und qualitative Analyse wird analog zur herkömmlichen FTA durchgeführt (siehe Abschnitt 2.2.2). Die FTA deckt damit Schwächen des Systems auf und zeigt die Sensitivität gegenüber Komponentenausfällen und Defekten. Werden bei dieser Analyse eklatante Schwächen des Systems entdeckt, müssen diese sofort im formalen Modell behoben werden. Ansonsten fließen die Ergebnisse der FTA und das formale Modell in die Integrationsphase ein.

9.1.3 Integration (III)

Die dritte Phase bringt die Ergebnisse der formalen Modellierung und der Sicherheitsanalyse zusammen. Meist führt dies sowohl in den Fehlerbäumen als auch im formalen Modell zu Anpassungen. Beobachtungen in verschiedenen Fallstudien zeigten, dass Fehlerbäume häufig Ursache/Wirkungs-Beziehungen beschreiben, die im formalen Modell nicht gelten. Der Grund dafür ist, dass der Sicherheitsingenieur ein anderes Verständnis der Systemfunktionalität hat, als der Entwickler des formalen Models. Der Fehlerbaum muss geändert und an das formale Modell angepasst werden. Auf der anderen Seite liefert die FTA oft weitere Sicherheitsbedingungen, die über dem formalen Modell nachgewiesen werden müssen. Sicherheitsbedingungen, die aus dem Fehlerbaum folgen, sind beispielsweise Gefährdungen oder anderer Zwischenereignisse des Fehlerbaums, die nicht eintreten dürfen. Außerdem beschreibt der Fehlerbaum in den Basisereignissen Komponentenfehler. Komponentenfehler oder mögliche Defekte sind meist nicht im formalen Modell enthalten, denn bei der Formalisierung wird sich auf das funktionale Verhalten des Systems, also das Soll-Verhalten, konzentriert. Jedoch müssen alle Ereignisse der FTA im Modell ausgedrückt werden können. Ist dies nicht der Fall, kann das formale Modell um diese Ereignisse erweitert werden, d. h. Ausfälle und Defekte werden im formalen Modell berücksichtigt. Alternativ dazu kann der Fehlerbaum so abgeändert werden, dass diese Ereignisse nicht mehr auftreten. Welche Fehler im formalen Modell betrachtet werden müssen und welche vernachlässigt werden können, entscheidet der Sicherheitsingenieur. An dieser Stelle unterscheiden wir zwischen Hard- und Softwarefehler.

Hardware- vs. Softwarefehler Die FTA eines softwarebasierten Systems deckt sowohl relevante Hardware- als auch Softwarefehler auf. Hardwarefehler kann man nicht ausschließen, da sie nicht von der Modellierung der aktuellen Anlage abhängen, sondern durch betriebsbedingte Defekte hervorgerufen werden. Deshalb kann die Ausfallwahrscheinlichkeit nur durch Wahl zuverlässigerer Komponenten bzw. Einbau redundanter Komponenten verringert werden. An dieser Stelle helfen die qualitativen und quantitativen Analysemöglichkeiten der FTA, verschiedene Systemmodelle miteinander zu vergleichen und Vorschläge für Verbesserungen des Modells zu liefern. Jedoch müssen alle als relevant eingestuften,

hardwarebedingten Fehler im formalen Modell berücksichtigt werden, um einerseits ein adäquates Systemmodell zu erhalten und um andererseits in der formalen FTA über diese Fehlerereignisse sprechen zu können. Für die Modellierung mit Statecharts haben wir in Abschnitt 4.3 beschrieben, wie Hardwareausfälle in das formale Modell integriert werden können. An diesem Punkt unterscheiden wir uns wesentlich von Bruns und Anderson. Sie schlagen in [BA93] für den Vollständigkeitsnachweis von Fehlerbäumen vor, den Fehlerbaum auf die Ereignisse einzuschränken, die im formalen Modell berücksichtigt sind.

Für softwarebedingte Fehler sehen wir eine andere Vorgehensweise vor. Mit „softwarebedingte“ bezeichnen wir Teile im Modell, die später in Software realisiert werden. Hier folgen wir Leveson: „... if design errors are found in the [fault] tree through this process, they should be fixed rather than left in the code and assigned a probability“ [Lev95]. Die meisten der softwarebedingten Fehler sollten schon während des Nachweises der funktionalen Korrektheit entdeckt und behoben worden sein. Jedoch ist diese Verifikation unter der Voraussetzung durchgeführt worden, dass alle (Hardware-) Komponenten korrekt, d. h. wie spezifiziert, funktionieren. Softwarebedingte Fehler können verborgen geblieben sein, wenn sie nur zusammen mit mit einem Hardwareausfall zum Tragen kommen (entspricht einem *Und*-Gatter im Fehlerbaum). Dies wird durch die FTA entdeckt. Das formale Modell muss nun geändert werden, um alle softwarebedingten Fehler auszuschließen.

Lose vs. eng gekoppelte formale FTA Nachdem das formale Modell und die FTA aneinander angepasst wurden, gibt es zwei Stufen der formalen FTA. Wenn die FTA und das formale Modell durch Austausch von Ergebnissen gekoppelt werden, nennen wir dies *lose gekoppelte formale FTA*. Die FTA wird dann hauptsächlich dazu genutzt, um Schwachstellen des Systems zu ermitteln und Sicherheitseigenschaften für das System abzuleiten. Die aus der FTA stammenden Sicherheitseigenschaften werden dann formalisiert und über dem formalen Modell nachgewiesen. Um aussagekräftige Ergebnisse zu erhalten, muss die FTA relativ zum formalen Modell sein, d. h. die (Ursache/Wirkungs-)Beziehungen und weitere Bedingungen der Formalisierung müssen so in der FTA berücksichtigt sein, dass sie dem formalen Modell entsprechen. Obwohl dieser Austausch nicht durchgängig formal ist, zeigen verschiedene Fallstudien [ORS+02, RST00b] den gegenseitigen Nutzen. Die formalen Modelle erhalten eine Systematik, um wichtige Sicherheitseigenschaften ableiten zu können. Außerdem erhält die FTA eine formale Basis. Zusätzlich zeigt die qualitative und quantitative Analyse der FTA Schwächen und Verbesserungsmöglichkeiten des formalen Modells auf.

Für ein größeres Maß an Sicherheit formalisiert die *eng gekoppelte formale FTA* neben den Sicherheitseigenschaften der FTA alle Ereignisse des Fehlerbaums über dem formalen Modell. Der Nachweis der Beweispflichten für die Fehlerbaumgatter aus Kapitel 5 garantiert dann die Vollständigkeit des Fehlerbaums. Zusätzlich können zur Validation des Fehlerbaums noch die Korrektheitsbedingungen nachgewiesen werden. Sowohl für die Formalisierung der Ereignisse als auch für den Nachweis der Gatterbedingungen schlagen wir einen top-down-Ansatz vor. Kann in einem Fehlerbaum nämlich die Korrektheitsbedingung eines Gatters nicht gezeigt werden, sind möglicherweise Ereignisse als Ursachen aufgeführt,

die nicht zur untersuchten Gefährdung führen und deshalb nicht betrachtet werden müssen. Das Weglassen dieser Ereignisse macht die Betrachtung ganzer Unterbäume in der formalen FTA überflüssig. Deshalb kann auch der Nachweis der Korrektheit von Fehlerbaumgattern nützlich sein (obwohl wir in der Diskussion in Abschnitt 5.1 gute Gründe gesammelt haben, die Korrektheit nicht in der FTA-Semantikdefintion zu betrachten). Der nicht erfolgreiche Nachweis der Vollständigkeit führt dazu, dass zusätzliche oder andere Ursachen im Fehlerbaum betrachtet werden müssen. Da sich solche Änderungen immer auf ganze Unterbäume eines Gatters auswirken, kann der top-down-Ansatz eine Menge Beweisarbeit sparen, da Fehler frühzeitig entdeckt werden.

Ist die Vollständigkeit eines formalen Fehlerbaums bewiesen und die Verifikation weist außerdem nach, dass mindestens ein Basisereignis jeder minimalen Schnittmenge ausgeschlossen wird, tritt die analysierte Gefährdung nicht ein. Dies garantiert das minimale Schnittmengen-Theorem. Die eng gekoppelte formale FTA ergibt deshalb eine sehr hohe Sicherheitsgarantie, sowohl bezüglich funktionaler Sicherheit als auch bezüglich Ausfallsicherheit.

Im Vergleich zwischen der eng und lose gekoppelten formalen FTA bietet die eng gekoppelte formale FTA ein größeres Maß an Sicherheit. Demgegenüber steht der Aufwand für die Formalisierung und Validierung des Fehlerbaums, der sich in höheren Kosten für die Durchführung niederschlägt. Wir haben auch mit der lose gekoppelten formalen FTA sehr gute Ergebnisse erzielt. Durch die informelle Verknüpfung der formalen Modelle mit der FTA entsteht weniger Aufwand bei der Durchführung, jedoch wird eine mögliche Quelle für Sicherheitslücken – ein unvollständiger Fehlerbaum – nicht geschlossen.

Modellprüfung vs. interaktive Verifikation Sowohl die lose als auch die eng gekoppelte formale FTA erfordern Verifikation. Wenn das System als endliches, diskretes Modell dargestellt werden kann, können die Beweispflichten für Fehlerbäume automatisch mittels Modellprüfung nachgewiesen werden. Dazu wurden in Abschnitt 7.2 die Vollständigkeitsbedingungen der Fehlerbaumgatter in CTL angegeben und eine Methodik entwickelt, mit der sowohl punktuelle als auch temporale Ereignisse korrekt behandelt werden können. Die Beweispflichten werden dann mit einem CTL-Modellprüfer nachgewiesen.

Der interaktive Nachweis der Fehlerbaumbedingungen ist notwendig, wenn die Modellprüfung nicht anwendbar ist, d. h das System nicht zustandsendlich spezifiziert werden kann. Die interaktive Verifikationsumgebung KIV unterstützt den Nachweis von ITL-Formeln und kann deshalb die FTA-Bedingungen aus Abschnitt 5.3.1 nachweisen. Zur Spezifikation (eingebetteter) dynamischer Systeme wurde in Abschnitt 6 die Möglichkeit geschaffen, in KIV Statechart-Spezifikationen zu erstellen und darüber temporallogische Beweise zu führen. Damit kann in KIV die Korrektheit der Fehlerbaumgatter und die Vollständigkeit formaler Fehlerbäume über einem Statechart-Systemmodell nachgewiesen werden.

Beide Methoden zur Validierung der FTA werden in dieser Arbeit vorgestellt. In Abschnitt 7.3 haben wir den Fehlerbaum des Drucktanks mit CTL-Modellprüfung validiert und in Kapitel 10 sehen wir eine formale FTA für die funktbasierte Bahnübergangsteuerung

mit eine Statechart-Modellierung und der interaktiven ITL-Verifikation.

9.1.4 Auswertung (IV)

Schließlich fasst die vierte Phase die Ergebnisse der formalen FTA zusammen. Die formale FTA liefert ein funktional korrektes Modell mit verifizierten Sicherheitseigenschaften, eine Bewertung der Sicherheit des Modells bei Komponentenausfällen und bei der eng gekoppelten formalen FTA vollständige Fehlerbäume.

Die Sicherheitsbewertung durch die qualitative und quantitative Auswertung der Fehlerbäume ermöglicht es, Aussagen über ein formales Modell zu machen, die über die funktionale Korrektheit hinausgehen. Formale Modelle können bezüglich ihrer Schwachstellen und Fehlersensitivität bewertet werden. Diese Bewertung erlaubt es sogar, die Güte verschiedener formaler Modelle zu vergleichen. Modelle können, obwohl funktional korrekt, inhärent fehleranfälliger sein, als andere. Betrachtet man eine Bahnübergangsteuerung mit den Modellierungsalternativen „Zug erhält Freigabe-Signal bei gesichertem Bahnübergang" und „Zug erhält Stopp-Signal bei ungesichertem Bahnübergang", dann führt ein Funkausfall nur bei der zweiten Alternative zu einer Kollision. Diese Designalternative ist damit wesentlich unsicherer, selbst wenn beide funktional korrekt spezifiziert und entwickelt wurden. Solche Sicherheitsbetrachtungen konnten bisher für formale Modelle nicht gemacht werden, können nun aber mit der formalen FTA durchgeführt werden. Führen wir keinen Vergleich verschiedener Modelle durch, sondern betrachten ein einzelnes Systemmodell mit der formalen FTA, dann weisen die minimalen Schnittmengen der FTA auf Schwächen und Verbesserungsmöglichkeiten in diesem Modell hin.

Die Sicherheitsanalyse von Modellen stützt sich auf die minimalen Schnittmengen von Fehlerbäumen. Die eng gekoppelte formale FTA weist die Vollständigkeit des Fehlerbaums nach. Mit dem minimalen Schnittmengen-Theorem aus Abschnitt 5.1 folgt für vollständige Fehlerbäume, dass jede mögliche Ursache für die untersuchte Gefährdung in den minimalen Schnittmengen enthalten ist. Die formale FTA erhöht damit die Qualität der minimalen Schnittmengen und der gesamten Sicherheitsanalyse.

Zusammenfassend garantiert die formale FTA ein funktional korrektes Modell mit einem Höchstmaß an Sicherheit bei Komponentenausfällen.

9.2 Anwendungen

In diesem Abschnitt stellen wir den Ablauf einer formalen FTA am Beispiel einer losen und einer eng gekoppelten formalen FTA vor.

9.2.1 Lose gekoppelte formale FTA

Das Vorgehen der lose gekoppelten formalen FTA lässt sich eindrucksvoll an der Industriefallstudie „Höhenkontrolle Elbtunnel" [ORS+02, ORS+03] erläutern. Im Hamburger Elbtunnel gibt es verschieden hohe Tunnelröhren. Nur durch die größte Röhre können überhohe

Fahrzeuge, die eine Höhe von über 4 m haben, fahren, ohne mit den Tunneleingängen zu kollidieren. Das Höhenkontrollsystem hat nun die Aufgabe, die Einfahrt überhoher Fahrzeuge in die niederen Tunneleingänge zu verhindern und diese Fahrzeuge in den größeren Tunnel zu lotsen. Dazu nutzt das Höhenkontrollsystem verschiedene Arten von Sensoren. Aus dem Zusammenspiel der Sensoren ergibt sich die Zuordnung von überhohen Fahrzeugen zu Fahrbahnen bzw. Tunneleinfahrten. Die Höhenkontrolle berechnet diese Zuordnung aus den Sensordaten und sobald ein überhohes Fahrzeug auf einen der niederen Tunneleingänge zufährt wird ein Alarm ausgelöst. Die Aufgabe war nun, eine Sicherheitsprüfung des Höhenkontrollsystems durchzuführen. Dazu haben wir eine lose gekoppelte formale FTA nach dem oben beschriebenen Ablauf durchgeführt.

Phase I In der Anforderungsanalyse mussten wir zuerst einmal das Zusammenspiel der Sensoren verstehen, aus dem die Zuordnung der Fahrzeuge zu den Tunneleingängen berechnet wird. Nach Studium der informellen Beschreibung wurde in STATEMATE ein Statechart-Modell erstellt und die Simulationsumgebung von STATEMATE [HLN+90] genutzt, um mit den Auftraggebern die Modellierung zu diskutieren und Unklarheiten über die Systemfunktionalität auszuräumen. Die Simulation des Statechart-Modells war von großem Nutzen, da den Auftraggebern die Statechart-Notation nicht geläufig war. Nachdem die Soll-Funktionalität der Höhenkontrolle geklärt war, wurde uns die Gefährdungsliste von den Auftraggebern vorgegeben. Es sollten die Gefährdungen „Kollision“ (eines überhohen Fahrzeugs mit dem Tunneleingang) und „Fehlalarm“ (Alarm, ohne dass ein überhohes Fahrzeug in einen niederen Tunnel einfährt) untersucht werden. Ein Fehlalarm kann entstehen, wenn ein Sensor fälschlicherweise ein überhohes Fahrzeug meldet.

Phase II Für die formale FTA wurde das gesamte Höhenkontrollsystem mit endlichen Automaten modelliert. Die Statechart-Modellierung aus der Analysephase diente dazu als Basis für die Modellierung. Besonders günstig war in diesem Fall, dass sowohl Statecharts als auch endliche Automaten Zustandsübergangssysteme beschreiben, die sich leicht vergleichen lassen. Außerdem wurden die Gefährdungen „Kollision“ und „Fehlalarm“ mit der FTA untersucht. Die sicherheitstechnische Auswertung der FTA ergab, dass jede Maßnahme zur Verringerung des Kollisionsrisikos das Fehlalarmrisiko erhöht und umgekehrt.

Phase III In der Integrationsphase wurden die Ergebnisse der Formalisierung und der FTA zusammengeführt. Aus der FTA flossen die Sicherheitsanforderungen „keine Kollision“ und „keine Fehlalarme“ ein, die im formalen Modell nachgewiesen wurden. Bei der Verifikation wurde festgestellt, dass das gleichzeitige Passieren zweier überhoher Fahrzeuge eines Kontrollpunktes zu einer Kollision führen kann, da der Kontrollpunkt nur ein Fahrzeug erkennt. Das zweite Fahrzeug wird von den Sensoren „übersehen“. Diese Schwachstelle wurde während der informellen FTA nicht entdeckt und zeigt den Nutzen einer formalen FTA. Die Anforderung „kein Fehlalarm“ konnte ebenfalls nur unter bestimmten Vorbedingungen gezeigt werden. Die Vorbedingungen beschreiben Szenarien für Fehlalarme, die inhärent in dem Höhenkontrollsystem enthalten sind und auch schon bekannt waren.

Außerdem zeigt die FTA einige Schwächen der Höhenkontrolle auf, die sich aus der Positionierung der Sensoren ergeben. Die Positionen der Sensoren sind jedoch fest vorgegeben, so dass diese Schwächen leider nicht beseitigt werden können. Das Fehlalarmrisiko könnte jedoch durch die Installation eines Zusatzsensors stark reduziert werden, ohne dabei die Kollisionsgefahr zu erhöhen. Obwohl dieser Vorschlag baulich leicht zu realisieren gewesen wäre, ist er leider nicht in die Höhenkontrolle übernommen worden.

Phase IV Für die Auswertung der formalen FTA fassen wir die obigen Ergebnisse nochmals zusammen. Im Höhenkontrollsystem für den Elbtunnel führt genau ein Situation zu einer Kollision, nämlich das gleichzeitige Passieren eines Sensors von zwei überhohen Fahrzeugen. In zwei weiteren Situationen kommt es zu einem Fehlalarm. Die Möglichkeit von Fehlalarmen war vor unserer Analyse schon bekannt, jedoch nicht, dass mit der gegebenen Steuerung eine Kollision möglich ist. Neben verschiedenen Verbesserungsvorschlägen, die technisch nicht realisierbar sind, konnte die formale FTA einen Verbesserungsvorschlag ausarbeiten, der leicht realisierbar ist.

Die Höhenkontrolle des Hamburger Elbtunnels dient hier zur Veranschaulichung der Methodik einer lose gekoppelten formalen FTA und ist keine vollständige Beschreibung unserer Untersuchungen. Für weitere Details der Fallstudie, der formalen Modellierung und der sicherheitstechnischen Analyse verweisen wir auf [ORS+02, ORS+03].

9.2.2 Eng gekoppelte formale FTA

Für die Fallstudie „Drucktank“ aus Abschnitt 3.1 haben wir in Abschnitt 7.3 eine eng gekoppelte formale FTA durchgeführt. Wir fassen hier den methodischen Ablauf nochmals zusammen.

Phase I Im *Fault Tree Handbook* [VGRH81] ist schon eine detaillierte Systembeschreibung vorhanden. Deshalb bestand die Anforderungsanalyse in der intensiven Studie dieser Beschreibung. Auf eine prototypische Implementierung oder Simulation des Systems haben wir verzichtet, da in dieser Untersuchung keine Kommunikation mit Auftraggebern stattfand. Wir mussten also unsere Vorstellung vom System niemandem vermitteln.

Phase II In der Sicherheitsanalysephase profitierten wir wiederum vom *Fault Tree Handbook*, das einen (informellen) Fehlerbaum vorgegeben hat. Wir mussten also nur noch ein formales Modell erstellen. Die Fallstudie Drucktank kann zustandsendlich modelliert werden, weshalb wir wieder eine Spezifikation mit endlichen Automaten für die CTL-Modellprüfung erstellt haben (siehe Abbildung 7.2, Seite 144).

Phase III In der Integrationsphase haben wir die Ereignisse des Fehlerbaums formalisiert. Dazu mussten alle möglichen Ausfälle von Komponenten in das Automatenmodell des Drucktanks integriert werden, die im Fehlerbaum auftreten. Wir realisierten dies, indem wir in das Modell des Drucktanks Fehlerübergänge hinzufügten, die den Ausfall verschiedener

Hardwarekomponenten modellieren (siehe Abschnitt 7.3). Die Spezifikation von Fehlverhalten mit Statecharts aus Abschnitt 4.3 konnten wir dabei leicht auf die Automatenmodelle übertragen. Mit diesem modifizierten Modell konnten wir die Ereignisse des Fehlerbaums formalisieren, die FTA-Gatterbedingungen erzeugen und mittels CTL-Modellprüfung nachweisen. Die Spezifikation und den Nachweis der Vollständigkeitsbedingungen haben wir bereits in Abschnitt 7.3 beschrieben. Für weitere Details verweisen wir auf diesen Abschnitt und auf [TS03].

Phase IV Als Ergebnis der eng gekoppelten formalen FTA erhalten wir einen vollständigen Fehlerbaum und ein sicherheitsgeprüftes, formales Modell für den Drucktank. Mit den qualitativen und quantitativen Sicherheitsanalyseergebnissen aus dem *Fault Tree Handbook* [VGRH81] können wir die Güte des um Komponentenausfälle erweiterten formalen Modells bewerten. Sie sind in Abschnitt 3.1.1 zusammengefasst. Es hat sich aber gezeigt, dass selbst ein informell gut untersuchtes Beispiel wie der Drucktank, von einer formalen FTA präzisiert werden kann. Wir haben, wie erwartet, keine Fehler in der Analyse gefunden, jedoch ist ein Gatter, das informell als Zerlegungsgatter beschrieben ist, über dem formalen Modell ein Ursache/Wirkungs-Gatter.

9.3 Zusammenfassung

Wir haben eine Methodik zur Anwendung der formalen FTA entwickelt und erfolgreich an zwei Beispielen demonstriert. Die Höhenkontrolle des Elbtunnels demonstrierte die lose gekoppelte formale FTA und der Drucktank die eng gekoppelte formale FTA. Der Drucktank wurde bereits ausführlich diskutiert und haben wir haben uns bei diesem Beispiel auf die Vorgehensweise der formalen FTA konzentriert. Ein weiteres Beispiel einer eng gekoppelten formalen FTA präsentieren wir im folgenden Kapitel.

Die Beispiele zeigen, dass beide Ausprägungen der formalen FTA zur sicherheitstechnischen Untersuchung realer Anwendungen eingesetzt werden können. Die eng gekoppelte formale FTA ist aufwendiger, garantiert dafür aber ein Höchstmaß an Sicherheit. Für den Einsatz in realen Projekten ist es deshalb wichtig, zwischen Kosten und Sicherheit abzuwägen. Für eine Einschätzung, welchen Sicherheitsgewinn die eng gekoppelte formale FTA gegenüber der lose gekoppelten formalen FTA bringt und welche Zusatzkosten dafür entstehen, ist es noch zu früh. Dazu müssen noch eine Reihe von Studien, möglichst im industriellen Umfeld, durchgeführt werden. Die obigen Beispiele zeigen jedoch, dass sowohl die lose als auch die eng gekoppelte formale FTA in realen Anwendungen, mit den in dieser Arbeit geschaffenen Grundlagen, durchgeführt werden können.

KAPITEL 10

Formale FTA: Funkbasierter Bahnübergang

Die KIV Spezifikation des funkbasierten Bahnübergangs (kurz FunkFahrBetrieb, FFB) enthält den Kern der Fallstudie aus Abschnitt 3.2. Wir haben in der Spezifikation Vereinfachungen gegenüber dem vollständigen Modell des funkbasierten Bahnübergangs dahingehend vorgenommen, dass Schranken und Lichtzeichenanlagen nicht explizit modelliert sind, sondern nur deren Funktionalität — erhält der Bahnübergang einen Schließbefehl, so ist er nach einer gewissen Zeit gesichert. Bei der Spezifikation des Zugs wurde auf die Modellierung der manuellen Freigabe verzichtet. Diese etwas vereinfachte Version des FFBs haben wir mit Statecharts in KIV modelliert, die informelle FTA aus Abschnitt 3.2.1 formalisiert und daraus die Vollständigkeitsbedingungen für die Fehlerbaumgatter erzeugt und nachgewiesen.

10.1 Spezifikation

Die Spezifikation des funkbasierten Bahnübergangs ist in Abbildung 10.1 in graphischer Notation dargestellt. Die textuelle Repräsentation in KIV-Syntax folgt ausschnittsweise im nächsten Kapitel und vollständig im Anhang D. Der FFB besteht aus drei *Oder*-Zuständen, der Zug- und Bahnübergangsteuerung und der Kommunikationsschnittstelle. Alle drei Komponenten laufen parallel im Statechart *ffb*. Für die formale FTA spezifizieren wir auch Fehlverhalten oder Ausfälle von Komponenten. Diese Fehlerübergänge sind in Abbildung 10.1 als hinterlegte Übergänge oder statische Reaktionen dargestellt. Beschränken wir uns bei den Erläuterungen zur FFB-Spezifikation zunächst auf ein korrekt funktionierendes System (ohne Fehlerübergänge) und betrachten Ausfälle von Komponenten im Anschluss.

Die Kommunikation übermittelt die Signale zwischen dem Zug und dem Bahnübergang, die dabei verzögert werden. Für die Modellierung der verzögerten Signale benötigen wir zwei Bezeichner, die sich auf dasselbe Signal beziehen. Ein Signal wird vom Zug bzw. Bahnübergang gesendet und mit tiefgestellten $_{snd}$ gekennzeichnet, das zweite Signal wird vom Bahnübergang bzw. Zug empfangen und mit tiefgestelltem $_{rcv}$ gekennzeichnet. Die Kommunikationsschnittstelle vermittelt die Signale zwischen dem Zug und dem Bahnüber-

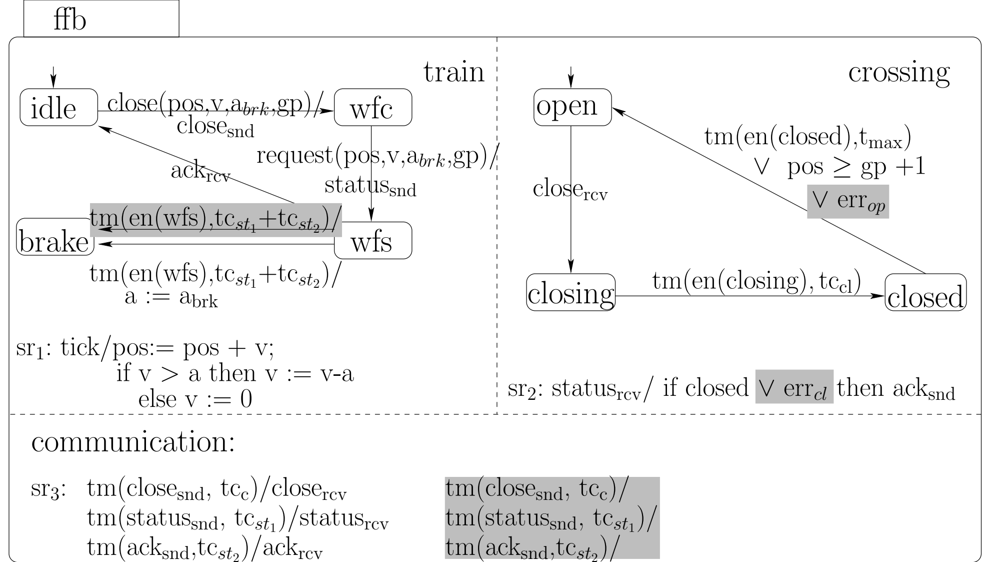


Abbildung 10.1: Funkbasierter Bahnübergang

gang. So wird z. B. in der statischen Reaktion sr_3 mit $tm(close_{snd}, tc_c)/close_{rcv}$[1] der Schließbefehl vom Zug $close_{snd}$ durch ein Timeout-Ereignis verzögert an den Bahnübergang als $close_{rcv}$ gesendet. Der Bahnübergang erhält um tc_c Zeiteinheiten verzögert den Schließbefehl.

Der Zug *train* berechnet mit der statischen Reaktion sr_1 in jedem Makro-Schritt seine neue Position und Geschwindigkeit. Ein Makro-Schritt entspricht einer Zeiteinheit, so dass in jedem Makro-Schritt die Position um $v\frac{m}{s} * 1s = v\,m$ erhöht wird. Zur Vereinfachung gehen wir davon aus, dass der Zug nicht beschleunigt und damit eine konstante Geschwindigkeit fährt, falls er nicht vor einem ungesicherten Bahnübergang abbremsen muss.

Zur Erläuterung der Kontrolllogik sehen wir uns das darunterliegende Modell zum Berechnen der Brems- und Einschaltpunkte in Abbildung 10.2 an. Der Punkt *gp* kennzeichnet den *Gefahrenpunkt*, also den Punkt, an dem der Zug anhalten muss, wenn der Bahnübergang nicht geschlossen ist. Um vor dem Gefahrenpunkt zum Stehen zu kommen, muss der Zug an der Position pos_{brk}, dem Bremseinschaltpunkt, anfangen zu bremsen. An dieser Stelle muss der Zugsteuerung also bekannt sein, ob der Bahnübergang gesichert ist, oder nicht. Dazu fragt die Zugsteuerung am Punkt pos_{status} den Status (gesichert oder ungesichert) des Bahnübergangs ab. Dieser Punkt liegt vor dem Bremseinschaltpunkt, um die Laufzeiten für die Kommunikation zu berücksichtigen. Damit der Bahnübergang geschlossen ist, wenn er die Statusabfrage erhält, sendet der Zug an der Position pos_{close} den Schließbefehl. Dieser Punkt liegt wiederum soweit vor dem Punkt pos_{status}, dass dem Bahnübergang genügend Zeit zum Schließen bleibt. Die genauen Entfernungen von pos_{close}, pos_{status} und pos_{brk} zum Gefahrenpunkt *gp* betrachten wir im Unterabschnitt 10.1.2

[1] *tm* steht für Timeout. Das Ereignis *tm(e,n)* wird n-Zeiteinheiten nach dem Ereignis e erzeugt.

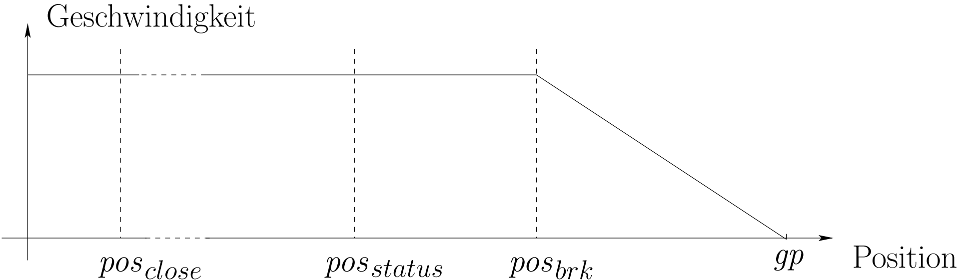


Abbildung 10.2: Geschwindigkeitskurve

Kehren wir nun wieder zur Abbildung 10.1 zurück. Im Folgenden sei *pos* die aktuelle Position des Zuges, v die aktuelle Geschwindigkeit, a die aktuelle Beschleunigung und gp der Gefahrenpunkt „Bahnübergang“. Das Prädikat $close(pos, v, a, gp)$ verwendet diese vier Parameter um zu bestimmen, ob sich der Zug an der Position pos_{close} befindet und dem Bahnübergang der Schließbefehl gesendet werden muss. Sobald $close(pos, v, a, gp)$ wahr ist, sendet der Zug einen Schließbefehl $close_{snd}$ über die Kommunikationsschnittstelle an den Bahnübergang. Nach der Verzögerungszeit tc_c reicht die Kommunikation den Befehl $close_{rcv}$ an den Bahnübergang weiter und dieser schließt die Schranken. Dazu wechselt der Bahnübergang in den Zustand *closing*. In *closing* werden die Schranken geschlossen. Nach tc_{cl} Zeiteinheiten sind die Schranken dann geschlossen und der Bahnübergang nimmt, durch das Timeout-Ereignis $tm(en(closing), tc_{cl})$[2] ausgelöst, den Zustand *closed* ein. An der Position pos_{status} ist $request(pos, v, a, gp)$ wahr und der Zug stellt, wieder durch die Kommunikation vermittelt, die Statusabfrage. Im gesicherten Zustand *closed* beantwortet die Bahnübergangsteuerung die Statusabfrage mit ack_{snd}, ansonsten wird sie verworfen. Wenn innerhalb der üblichen Kommunikationszeit $tc_{st_1} + tc_{st_2}$ der Zug die Bestätigung ack_{rcv} erhält, passiert er den Bahnübergang, ansonsten bremst er mit der maximalen Verzögerung a_{brk} ab und kommt vor dem Gefahrenpunkt Bahnübergang (gp in Abbildung 10.2) zum Stehen. Damit der Straßenverkehr nicht unnötig blockiert wird, öffnet der Bahnübergang die Schranken, wenn der Zug über den Gefahrenpunkt hinausfährt ($pos \geq gp + 1$) oder die Schranken zu lange geschlossen sind ($tm(en(closed), t_{max})$).

10.1.1 Spezifikation von Fehlverhalten

Bisher haben wir das Soll-Verhalten des FFBs besprochen. Formale Modelle beschränken sich meist auf die Modellierung des Soll-Verhaltens. Mit der formalen FTA können wir jedoch auch Ausfälle oder Defekte von Komponenten betrachten. Wir betrachten im Folgenden einen Bremsausfall des Zugs, den Ausfall der Kommunikationsschnittstelle, ein fehlerhaftes Senden der Freigabe und ein vorzeitiges Öffnen der Schranken. Diese Defekte oder Ausfälle sind in Abbildung 10.1 hinterlegt dargestellt.

[2] $en(closing)$ kennzeichnet das Betreten von *closing* (*en: entering*).

Der Bremsausfall wird durch einen zusätzlichen Übergang von *wfs* nach *brake* modelliert. Der Zustand *brake* wird aktiv, wenn der Zug bremst. Der zusätzliche Übergang ermöglicht es, nach *brake* zu gelangen, ohne die Verzögerung auf a_{brk} zu setzen. Der Zug befindet sich im Zustand *brake*, ohne wirklich zu bremsen.

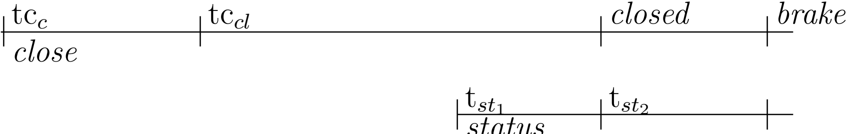
Analog modellieren die zusätzlichen statischen Reaktionen in *communication*, den Ausfall der Kommunikation. Zu jeder statischen Reaktion fügen wir eine zweite mit gleicher Aktivierungsbedingung hinzu, die deshalb in Konflikt steht und indeterministisch gewählt wird, die aber kein Ereignis sendet. Die zusätzliche statische Reaktion modelliert also den Ausfall der Kommunikation.

Schließlich modellieren wir die Fehler im Bahnübergang *crossing* durch zusätzliche boolesche Fehlerbedingungen, die in jedem Makro-Schritt einen zufälligen Wert bekommen. Wird err_{cl} wahr, kann eine Statusabfrage selbst dann mit einer Freigabe quittiert werden, wenn der Bahnübergang nicht gesichert ist. Falls err_{op} wahr ist, können die Schranken frühzeitig geöffnet werden, d. h. bevor der Zug passiert hat ($pos \geq gp$) bzw. der Timeout $tm(en(closed), t_{max})$ den Bahnübergang veranlasst, die Schranken wieder zu öffnen.

10.1.2 Bremskurvenberechnung

Wir gehen für die Fallstudie FFB davon aus, dass die Beschleunigung entweder Null ist, der Zug also eine konstante Geschwindigkeit fährt, oder gleich der maximalen Verzögerung a_{brk} und der Zug bremst, um vor dem Bahnübergang zum Stehen zu kommen.

Betrachten wir nochmals die Bremskurve in Abbildung 10.2. Die Berechnung der Punkte, an denen der Zug dem Bahnübergang ein Schließsignal bzw. die Statusabfrage sendet, basieren auf der Berechnung des Bremseinsatzpunktes. Zwischen der Statusabfrage am Punkt pos_{status} und dem Bremseinsatzpunkt pos_{brk} vergeht Zeit für die Kommunikation (Verbindungsaufbau, Senden der Daten, ...). Das Versenden der Statusabfrage vom Zug zum Bahnübergang benötigt die Zeit tc_{st_1}, das Senden der Freigabe vom Bahnübergang zum Zug nochmals tc_{st_2} Zeiteinheiten. Da der Zug eine konstante Geschwindigkeit v fährt, liegt der Punkt pos_{status} um $(tc_{st_1} + tc_{st_2}) * v$ vor pos_{brk}. Zwischen dem Schließsignal am Punkt pos_{close} und dem Bremseinsatzpunkt muss das Schließsignal an den Bahnübergang gesendet werden (Dauer: tc_c) und der Bahnübergang schließen (Dauer: tc_{cl}). Der Bahnübergang muss vollständig geschlossen sein, wenn die Statusabfrage den Bahnübergang erreicht. Deshalb muss noch die Antwortzeit für die Statusabfrage berücksichtigt werden. Damit muss pos_{close} um $(tc_c + tc_{cl} + tc_{st_2}) * v$ vor dem Bremseinsatzpunkt liegen. Den zeitlichen Verlauf verdeutlicht noch einmal der folgende Zeitstrahl, in dem die Zeitkonstanten für die Kommunikation eingetragen sind.



Die Zeitkonstanten tc_c, tc_{st_1} und tc_{st_2} können unterschiedlich gewählt werden und so verschieden lange Kommunikationszeiten modellieren. Soll jede Kommunikation die gleiche

Zeit benötigen, wählen wir $tc_{st_1} = tc_{st_2} = tc_c$. Zum Zeitpunkt *closed* müssen die Schranken geschlossen sein, bei *brake* beginnt der Bremsvorgang. Dementsprechend muss bei *close* der Schließvorgang eingeleitet und bei *status* die Statusabfrage gesendet werden.

Mit der Distanz $brake_d(v, a_{brk})$, die der Zug beim Bremsvorgang zurücklegt, ist dann $close_d(v, a_{brk})$ die Distanz zum Gefahrenpunkt Bahnübergang, bei der man das Schließsignal senden muss und $status_d(v, a_{brk})$ die Distanz, bei der man den Status abfragen muss.

$$\begin{aligned} close_d(v, a_{brk}) &:= brake_d(v, a_{brk}) + (tc_c + tc_{cl} + tc_{st_2}) * v + v \\ status_d(v, a_{brk}) &:= brake_d(v, a_{brk}) + (tc_{st_1} + tc_{st_2}) * v + v \end{aligned}$$

Dem Bahnübergang muss dann an der Position der Schließbefehl bzw. die Statusabfrage gesendet werden, an dem die oben definierten Distanzen zum Gefahrenpunkt Bahnübergang *gp* unterschritten werden. Der zusätzliche Abstand von v ist notwendig, da der Zug, wenn er noch vor der entsprechenden Position ist, im folgenden Makro-Schritt diese Position schon um maximal v überschritten haben kann. Wir können nun die Prädikate $close(pos, v, a_{brk}, gp)$ und $request(pos, v, a_{brk}, gp)$ aus Abbildung 10.1 definieren.

$$\begin{aligned} close(pos, v, a_{brk}, gp) &:\Leftrightarrow pos \geq gp - close_d(v, a_{brk}) \\ request(pos, v, a_{brk}, gp) &:\Leftrightarrow pos \geq gp - status_d(v, a_{brk}) \end{aligned}$$

Bleibt noch zu zeigen, wie sich die Bremsdistanz $brake_d(v, a_{brk})$ berechnet. Die Zugsteuerung verzögert während des Bremsvorgangs mit $a := a_{brk}$. Während jedes Makro-Schrittes wird die Zuggeschwindigkeit und -position mit der statischen Reaktion

$$tick/pos := pos + v; \textbf{if } v > a \textbf{ then } v := v - a \textbf{ else } v := 0$$

berechnet. Für die Verzögerung $a0$ und die Ausgangsgeschwindigkeit v_0 vor Beginn des Bremsvorgangs ist die Distanz bis zum Stillstand deshalb

$$(\sum_{i=0}^{v_0/a} v_0 - i * a).$$

Der Zug bremst v_0/a Makro-Schritte mit der Verzögerung a und legt dabei im i-ten Makro-Schritt die Distanz $v_0 - i * a$ zurück. Damit berechnet sich die Bremsdistanz für $a > 0$ mit einem Sicherheitsabstand von v zu

$$\begin{aligned} brake_d(v_0, a) = (\sum_{i=0}^{v_0/a} v_0 - i * a) + v &= (v_0 * \sum_{i=0}^{v_0/a} 1) - (a * \sum_{i=0}^{v_0/a} i) + v \\ &= (v_0/a + 1) * v_0 - a * \frac{(v_0/a + 1)(v_0/a)}{2} + v \\ &= \frac{(v_0/a + 1)(2v_0 - a(v_0/a))}{2} + v. \end{aligned}$$

Die benutzten Funktionen sind für die Fallstudie algebraisch spezifiziert worden. Die Funktion $./. : nat \times nat \rightarrow nat$ berechnet die Division auf natürlichen Zahlen (bei nicht ganzzahligem Ergebnis wird abgerundet, siehe Spezifikation *nat-div*, Seite 248). Die Sorte *nat*

ist in der algebraischen Spezifikation *nat-basic1*, Seite 250, spezifiziert, die Funktionen $. + .$ in *nat-basic1*, Seite 250, die Funktion $. - .$ in *nat*, Seite 249 und die Funktion $. * .$ in *nat-mult*, Seite 249. Auf diesen Spezifikationen aufbauend sind schließlich in *ffb-math*, Seite 247 die Funktionen $brake_d(v, a_{brk})$, $close_d(v, a_{brk})$ und $status_d(v, a_{brk})$ sowie die Prädikate $close(pos, v, a_{brk}, gp)$ und $request(pos, v, a_{brk}, gp)$ definiert.

Diese Spezifikationen bilden die mathematischen Grundlagen für die Statechart-Modellierung des FFBs. Die KIV-Spezifikation für das Statechart *train* findet sich in der Spezifikation *train-ctrl*, Seite 244, für *communication* in der Spezifikation *communication*, Seite 245, für *crossing* in Spezifikation *crossing-ctrl*, Seite 246 und das Statechart *ffb* fasst in der Spezifikation *ffb*, Seite 243, diese drei Statecharts zu einem *Und*-Zustand zusammen.

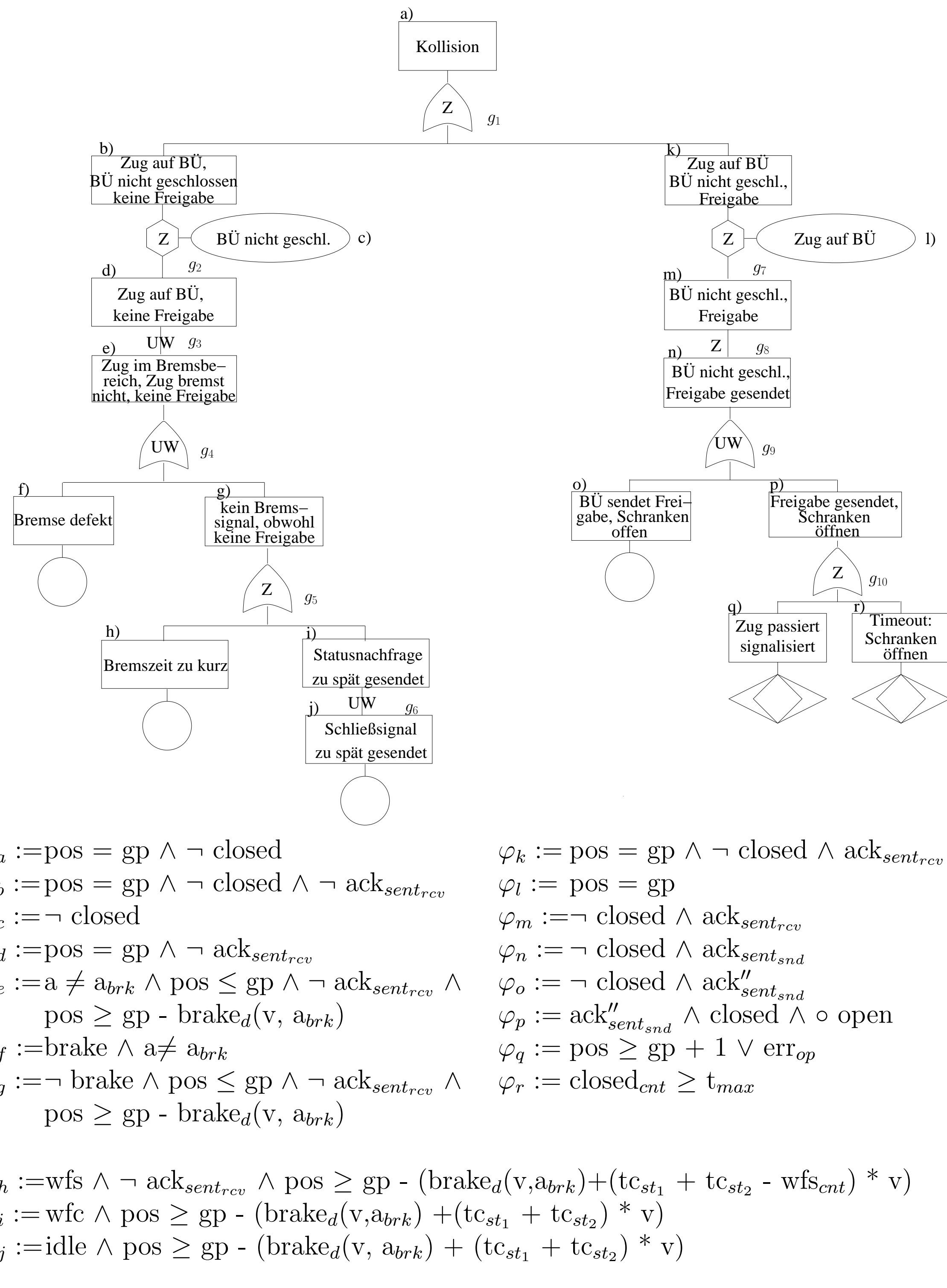
10.2 FTA: Kollision im funkbasierten Bahnübergang

Betrachten wir nun die FTA des funkbasierten Bahnübergangs, die in Abbildung 10.3 dargestellt ist. Der Fehlerbaum unterscheidet sich geringfügig von der FTA aus Abschnitt 3.2, da wir hier nur einen Ausschnitt der gesamten Fallstudie spezifiziert haben. Beim Zug wurde beispielsweise das Odometer und die Bremsen nicht detaillierter spezifiziert und deshalb werden diese Komponenten auch im Fehlerbaum nicht berücksichtigt, d. h. die Ereignisse f) und h) sind Basisereignisse und werden im Vergleich zum Fehlerbaum aus Abschnitt 3.2.1 nicht weiter untersucht. Ein weiterer Unterschied besteht darin, dass Ursachen, die nicht notwendigerweise zu einer Wirkung führen, im formalen Fehlerbaum explizit durch *Block*-Gatter gekennzeichnet sind (siehe Abschnitt 5.1, Motivation der FTA-Semantik). Im ursprünglichen Fehlerbaum sind diese *Block*-Gatter nicht vorhanden, sondern implizit in den *Oder*-Gattern enthalten. Die formale FTA konkretisiert hier die informelle FTA.

Die modellierten Hardwareausfälle sind als Blätter im Fehlerbaum zu finden. Ereignis f) repräsentiert den Bremsausfall, d. h. das Statechart befindet sich im Zustand *brake*, der Zug bremst aber nicht ($a = 0$). Die Ereignisse h) und j) sind Softwarefehler, die nicht eintreten sollten. Das Senden der Freigabe bei geöffneten Schranken (err_{cl}) findet sich in Ereignis o) und ein fehlerhaftes „Zug passiert“ Signal (err_{op}) in Ereignis q). Das (zu frühe) öffnen der Schranken durch den Timeout findet sich in Ereignis r). Dies ist ein „softer“ Fehler, der durch eine geänderte Kontrollsteuerung verhindert werden kann (die maximale Schließzeit t_{max} müsste mindestens größer $t_{st_2} + \frac{brake_d(v, a_{brk})}{v}$ sein).

Unterhalb des Fehlerbaums befindet sich die Formalisierung der einzelnen Ereignisse a) - r) des Fehlerbaums. Z. B. ist das Top-Ereignis Kollision, a), durch $\varphi_a := pos = gp \wedge \neg\ closed$ formalisiert. Zustände und Variablen entsprechen den Bezeichnern aus Abbildung 10.1, die Konstanten und Definition der Bremsdistanz $brake_d$ haben wir im vorhergehenden Abschnitt 10.1.2 besprochen (wfs_{cnt} in φ_h und $closed_{cnt}$ in φ_r entsprechen der Zeit, die im Zustand *wfs* bzw. *closed* verbracht wurde).

Da sowohl ack_{snd} als auch ack_{rcv} punktuelle Ereignisse im Statechart-Modell und nur einen Mikro-Schritt sichtbar sind, benötigen wir für die Formalisierung von „Freigabe gesendet“ bzw. „(keine) Freigabe erhalten“ eine Variable $ack_{sent_{snd}}$ ($ack_{sent_{rcv}}$) als Merker. $ack_{sent_{snd}}$ ($ack_{sent_{rcv}}$) ist dann wahr, wenn der Bahnübergang die Freigabe schon gesendet



φ_a :=pos = gp $\wedge$ $\neg$ closed

φ_b :=pos = gp $\wedge$ $\neg$ closed $\wedge$ $\neg$ ack$_{sent_{rcv}}$

φ_c :=$\neg$ closed

φ_d :=pos = gp $\wedge$ $\neg$ ack$_{sent_{rcv}}$

φ_e :=a $\neq$ a$_{brk}$ $\wedge$ pos $\leq$ gp $\wedge$ $\neg$ ack$_{sent_{rcv}}$ $\wedge$
pos $\geq$ gp - brake$_d$(v, a$_{brk}$)

φ_f :=brake $\wedge$ a$\neq$ a$_{brk}$

φ_g :=$\neg$ brake $\wedge$ pos $\leq$ gp $\wedge$ $\neg$ ack$_{sent_{rcv}}$ $\wedge$
pos $\geq$ gp - brake$_d$(v, a$_{brk}$)

φ_k := pos = gp $\wedge$ $\neg$ closed $\wedge$ ack$_{sent_{rcv}}$

φ_l := pos = gp

φ_m :=$\neg$ closed $\wedge$ ack$_{sent_{rcv}}$

φ_n := $\neg$ closed $\wedge$ ack$_{sent_{snd}}$

φ_o := $\neg$ closed $\wedge$ ack$''_{sent_{snd}}$

φ_p := ack$''_{sent_{snd}}$ $\wedge$ closed $\wedge$ $\circ$ open

φ_q := pos $\geq$ gp + 1 $\vee$ err$_{op}$

φ_r := closed$_{cnt}$ $\geq$ t$_{max}$

φ_h :=wfs $\wedge$ $\neg$ ack$_{sent_{rcv}}$ $\wedge$ pos $\geq$ gp - (brake$_d$(v,a$_{brk}$)+(tc$_{st_1}$ + tc$_{st_2}$ - wfs$_{cnt}$) * v)

φ_i := wfc $\wedge$ pos $\geq$ gp - (brake$_d$(v,a$_{brk}$) +(tc$_{st_1}$ + tc$_{st_2}$) * v)

φ_j :=idle $\wedge$ pos $\geq$ gp - (brake$_d$(v, a$_{brk}$) + (tc$_{st_1}$ + tc$_{st_2}$) * v)

Abbildung 10.3: FTA: Kollision

hat bzw. der Zug schon die Freigabe erhalten hat. Um dies zu erreichen, spezifizieren wir die ITL-Formeln

$$\begin{aligned}\varphi_{snd} &:= \neg\, ack_{sent_{snd}} \wedge \Box ack_{snd} \to ack''_{sent_{snd}} \wedge \neg\, ack_{snd} \to ack''_{sent_{snd}} = ack_{sent_{snd}} \text{ und}\\ \varphi_{rcv} &:= \neg\, ack_{sent_{rcv}} \wedge \Box ack_{rcv} \to ack''_{sent_{rcv}} \wedge \neg\, ack_{rcv} \to ack''_{sent_{rcv}} = ack_{sent_{rcv}}.\end{aligned}$$

φ_{snd} beobachtet das Ereignis ack_{snd} und bei Eintreten wird $ack_{sent_{snd}}$ gesetzt. Wenn ack_{snd} nicht eintritt, bleibt $ack_{sent_{snd}}$ unverändert. Da $ack_{sent_{snd}}$ mit false initialisiert wird, ist $ack_{sent_{snd}}$ immer dann wahr, wenn irgendwann einmal zuvor ack_{snd} gegolten hat. Entsprechendes gilt für $ack_{sent_{rcv}}$. Für den Nachweis der Vollständigkeitsbedingungen fügen wir die Formeln φ_{snd} und φ_{rcv} zum Statechart *ffb* hinzu, d. h. das Statechart *ffb*, φ_{snd} und φ_{rcv} bilden die Systembeschreibung. So können wir in den formalisierten Ereignissen des Fehlerbaums über $ack_{sent_{rcv}}$ und $ack_{sent_{snd}}$ sprechen. Da sowohl φ_{snd} als auch φ_{rcv} nur Statechart-Variablen beobachten, schränken sie das Verhalten des Statecharts *ffb* nicht ein.

Betrachten wir nun die Vollständigkeitsbedingungen, die sich aus der obigen Formalisierung der Fehlerbaum-Ereignisse ergibt. Im Fehlerbaum in Abbildung 10.3 sind die Gatter mit g_1 - g_{10} gekennzeichnet und mit Ihrem Typ beschriftet. Hat ein Gatter nur eine Ursache (wie z. B. g_2), müssen wir nur zwischen *UW*- und *Z*-Gatter unterscheiden, denn die Formalisierung von *Oder*-, *Und*- bzw. *Block*-Gatter (mit Nebenbedingung true) fallen in diesem Fall zusammen.

In Tabelle 10.1 zeigen wir ausführlich die ersten 4 Gatter des linken Teilbaums für den Fehlerbaum Kollision. In der Tabelle links ist das Gatter aus dem Fehlerbaum dargestellt, in der Mitte die Semantik des Gatters nach Abschnitt 5.3.1 und rechts die Instantiierte Vollständigkeitsbedingung. Für die übrigen Gatter geben wir in Tabelle 10.2

φ_a / Z g_1 / φ_b, φ_k	$\Box_{\mathrm{x}} \varphi_a \longrightarrow \varphi_b \vee \varphi_k$	$\Box_{\mathrm{x}}(pos = gp \wedge \neg closed) \longrightarrow$ $((pos = gp \wedge \neg closed \wedge \neg ack_{sent_{rcv}}) \vee$ $(pos = gp \wedge \neg closed \wedge ack_{sent_{rcv}}))$
φ_b / Z – φ_c g_2 / φ_d	$\Box_{\mathrm{x}}(\varphi_b \longrightarrow \varphi_c \wedge \varphi_d)$	$\Box_{\mathrm{x}}(pos = gp \wedge \neg closed \wedge \neg ack_{sent_{rcv}}) \longrightarrow$ $((\neg\, closed) \wedge (pos = gp \wedge \neg ack_{sent_{rcv}}))$
φ_d / UW g_3 / φ_e	$\neg\, (\neg\, \Diamond_{\mathrm{x}}(\varphi_e)\ ;\ \Diamond_{\mathrm{i}}\varphi_d)$	$\neg(\neg\Diamond_{\mathrm{x}}((a \neq a_{brk} \wedge \neg ack_{sent_{rcv}} \wedge pos \leq gp \wedge$ $pos \geq gp - brake_d(v, a_{brk})))\ ;$ $\Diamond_{\mathrm{i}}(pos = gp \wedge \neg ack_{sent_{rcv}}))$
φ_e / UW g_4 / φ_f, φ_g	$\neg\, (\neg\, \Diamond_{\mathrm{x}}(\varphi_f \vee \varphi_g)\ ;\ \Diamond_{\mathrm{i}}\varphi_e)$	$\neg(\neg\Diamond_{\mathrm{x}}((brake \wedge a \neq a_{brk}) \vee$ $(\neg\, brake \wedge \neg ack_{sent_{rcv}} \wedge pos \leq gp \wedge$ $pos \geq gp - brake_d(v, a_{brk})))\ ;$ $\Diamond_{\mathrm{i}}(a \neq a_{brk} \wedge \neg ack_{sent_{rcv}} \wedge pos \leq gp \wedge$ $pos \geq gp - brake_d(v, a_{brk})))$

Tabelle 10.1: Formalisierung der Gatter (I)

nur die instantiierte Vollständigkeitsbedingung an. Für ein Gatter g_i entspricht φ_{g_i} der Vollständigkeitsbedingung für das Gatter. Die Gatterbedingung g_8 benötigt z. B. die oben

$$
\begin{aligned}
\varphi_{g_5} :=\ & \boxtimes(a \neq a_{brk} \wedge pos \leq gp \wedge \neg ack_{sent_{rcv}} \wedge pos \geq gp - brake_d(v, a_{brk})) \rightarrow \\
& ((wfs \wedge \neg ack_{sent_{rcv}} \wedge pos \geq gp - (brake_d(v, a_{brk}) + (tc_{st_1} + tc_{st_2} - wfs_{cnt}) * v)) \\
& \vee (wfc \wedge pos \geq gp - (brake_d(v, a_{brk}) + (tc_{st_1} + tc_{st_2}) * v))) \\
\varphi_{g_6} :=\ & \neg(\neg\otimes(idle \wedge pos \geq gp - (brake_d(v, a_{brk}) + (tc_{st_1} + tc_{st_2}) * v)) \ ; \\
& \oplus(wfc \wedge pos \geq gp - (brake_d(v, a_{brk}) + (tc_{st_1} + tc_{st_2}) * v)) \\
\varphi_{g_7} :=\ & \boxtimes(pos = gp \wedge \neg closed \wedge ack_{sent_{rcv}}) \rightarrow (pos = gp) \wedge (\neg closed \wedge ack_{sent_{rcv}}) \\
\varphi_{g_8} :=\ & \boxtimes(\neg closed \wedge ack_{sent_{rcv}}) \rightarrow (\neg closed \wedge ack_{sent_{snd}}) \\
\varphi_{g_9} :=\ & \neg(\neg\otimes((\neg closed \wedge ack''_{sent_{snd}}) \vee (ack''_{sent_{snd}} \wedge closed \wedge \circ open)) \ ; \\
& \oplus(\neg closed \wedge ack_{sent_{snd}})) \\
\varphi_{g_{10}} :=\ & \boxtimes(ack''_{sent_{snd}} \wedge closed \wedge \circ open) \rightarrow \\
& (pos \geq gp + 1 \vee err_{op}) \vee (closed_{cnt} \geq t_{max})
\end{aligned}
$$

Tabelle 10.2: Formalisierung der Gatter (II)

beschriebene Variable $ack_{sent_{snd}}$ zum Beobachten, dass der Bahnübergang eine Freigabe gesendet hat. Damit beschreibt das Gatter g_8, dass der Zug (bei geöffneten Schranken) nur eine Freigabe erhalten hat ($ack_{sent_{rcv}}$), wenn (bei geöffneten Schranken) der Bahnübergang irgendwann einmal eine Freigabe gesendet hat (beobachtet durch $ack_{sent_{snd}}$). Es werden also keine Freigaben generiert. Diese Bedingung ist wahr, da Defekte in der Kommunikation nur dahingehend auftreten können, dass sie Signale nicht weitersenden. Die Kommunikationsschnittstelle erzeugt keine Signale, auch nicht fälschlicherweise.

10.2.1 Nachweis der Vollständigkeit

Für den Nachweis der Vollständigkeit müssen wir die Gatterbedingungen φ_{g_i} nachweisen. Der Fehlerbaum gibt uns eine Zerlegung der Ursachen für die Kollision auf die Komponenten des FFBs vor. Sobald wir nur über Ereignisse einer Komponente sprechen, können wir den Nachweis lokal über diese Komponente führen. Aus dem Fehlerbaum aus Abbildung 10.3 ergibt sich, dass die Gatterbedingungen g_1, g_2, g_7 und g_8 über dem gesamten Statechart *ffb* geführt werden müssen. Die Gatterbedingungen g_3 – g_6 können über dem Statechart *train*, die Gatterbedingungen g_9 und g_{10} über dem Statechart *crossing* gezeigt werden.

Bemerkung Um den Nachweis der Gatterbedingungen über den Statecharts *train* und *crossing* auf das Gesamtchart *ffb* liften zu können, benötigen wir eine kompositionale Statechart Semantik (die in dieser Arbeit vorgestellte Semantik ist nicht kompositional). Zur Demonstration der formalen FTA wurde dieser Punkt hier vernachlässigt und die Gatterbedingungen trotzdem lokal nachgewiesen. Das Beispiel zeigt aber, dass eine komponentenbasierte Beweisunterstützung für Statecharts eine wichtige Weiterentwicklung ist.

Ein Statecharts SC (*ffb*, *train* oder *crossing*) beschreibt alle Abläufe, die die Vollständigkeitsbedingungen erfüllen sollen und ist deshalb eine Vorbedingung für φ_{g_i}. Der Nachweis beginnt in einer initialen Konfiguration des Statecharts, die $\widehat{SC}$ beschreibt. Eine zusätzliche Vorbedingung sind die oben beschriebenen Formeln φ_{snd} und φ_{rcv}, die das Senden bzw. Empfangen der Freigabe beobachten. φ_{snd} bzw. φ_{rcv} werden in den Voraussetzungen nur dann benötigt, wenn die nachzuweisende Vollständigkeitsbedingung $ack_{sent_{snd}}$ bzw. $ack_{sent_{rcv}}$ verwendet werden.

Diese Vorbedingungen genügen für den Nachweis der Gatterbedingungen über der Komponente Zug jedoch noch nicht. Startet die Statechart-Steuerung erst, wenn der Zug schon den Schließzeitpunkt passiert hat, kann die Steuerung nicht mehr rechtzeitig den Bahnübergang sichern bzw. den Zug abbremsen. Der Zug fährt in einen ungesicherten Bahnübergang ein. Deshalb müssen wir für den Nachweis der Vollständigkeit über der Komponenten Zug verlangen, dass sich der Zug zu Beginn noch „weit genug“ vom Bahnübergang entfernt befindet. Es muss zu Beginn also $\neg\ close(pos, v, a_{brk}, gp)$ gelten. Wir fassen die nachzuweisenden Vollständigkeitsbedingungen in der Tabelle zusammen.

Gatter	Vollständigkeitsbedingung
g_1 :	$\widehat{ffb} \wedge \varphi_{rcv} \rightarrow \varphi_{g_1}$
g_2 :	$\widehat{ffb} \wedge \varphi_{rcv} \rightarrow \varphi_{g_2}$
g_3 :	$\widehat{train} \wedge \varphi_{rcv} \wedge \neg\ close(pos, v, a_{brk}, gp) \rightarrow \varphi_{g_3}$
g_4 :	$\widehat{train} \wedge \varphi_{rcv} \wedge \neg\ close(pos, v, a_{brk}, gp) \rightarrow \varphi_{g_4}$
g_5 :	$\widehat{train} \wedge \varphi_{rcv} \wedge \neg\ close(pos, v, a_{brk}, gp) \rightarrow \varphi_{g_5}$
g_6 :	$\widehat{train} \wedge \varphi_{rcv} \wedge \neg\ close(pos, v, a_{brk}, gp) \rightarrow \varphi_{g_6}$
g_7 :	$\widehat{ffb} \wedge \varphi_{rcv} \rightarrow \varphi_{g_7}$
g_8 :	$\widehat{ffb} \wedge \varphi_{snd} \wedge \varphi_{rcv} \rightarrow \varphi_{g_8}$
g_9 :	$\widehat{crossing} \wedge \varphi_{snd} \rightarrow \varphi_{g_9}$
g_{10} :	$\widehat{crossing} \wedge \varphi_{snd} \rightarrow \varphi_{g_{10}}$

Tabelle 10.3: Vollständigkeitsbedingungen

Bis auf die Bedingungen für die Gatter g_3 und g_8 haben wir alle Vollständigkeitsbedingungen für den FFB nachgewiesen. Dabei hat sich ein typischer Automatisierungsgrad (Anzahl Benutzerinteraktionen gegenüber der Anzahl der Beweisschritte) von ca. 80% gezeigt. Die Beweisgröße liegt dabei in der Größenordnung von 100–500 Beweisschritten. Die meisten interaktiven Beweisschritte sind dabei die Anwendung der *sc unwind*-Regel zum Ausführen der Statecharts und das Anwenden der Induktionshypothese. Die Anwendung dieser beiden Regeln wurden für die Beweise nicht durch Heuristiken automatisiert, da es bei den relativ komplizierten Beweisen sehr wichtig ist, den Überblick über den Beweisfortgang zu erhalten. Die Zustandskonfiguration erkennt man leicht an der Variablenbelegung,

die vor Anwendung der *sc unwind*-Regel gilt.

10.3 Verwandte Arbeiten

In [KT02] haben wir eine Modellierung des FFBs mit STATEMATE-Statecharts präsentiert und diese in [RST00b, RST00a] sicherheitstechnisch untersucht. Dabei haben wir uns an die Fallstudienbeschreibung aus [JS99] gehalten, in der die Schranken- und Lichtzeichenanlagen explizit modelliert sind und ebenfalls ausfallen können. Die Fallstudienbeschreibung aus [JS99] dient dem Schwerpunktprogramm „Integration von Techniken der Softwarespezifikation für ingenieurwissenschaftliche Anwendungen“ [Sof98] der Deutschen Forschungsgemeinschaft als Referenzfallstudie für verschiedene Modellierungstechniken.

Die STATEMATE-Spezifikation aus [KT02] dient auch Klose als Grundlage für die LSC-Verifikation [Klo03]. LSCs (*Live Sequence Charts*, [DH99, DH98]) sind eine Erweiterung von *Message Sequence Charts* (MSCs, [IT00]) und Sequenzdiagrammen (SD) der *Unified Modelling Language* (UML, [OMG03]). LSCs beschreiben formal den Ablauf der Kommunikation zwischen verschiedenen Komponenten einer Spezifikation. Klose verwendet in [Klo03] LSCs zur Beschreibung von Anforderungen einer STATEMATE-Spezifikation. Diese Anforderungen werden mit dem Modellprüfer STVE (Statemate Verification Environment, [BDW00]) für ein STATEMATE-Modell nachgewiesen. In seiner Arbeit hat Klose für eine ganze Reihe von erwünschten und unerwünschten Szenarien gezeigt, dass sich das STATEMATE-Modell wie erwartet verhält. Ungewünschte Szenarien beschreiben den Ablauf mit defekten Komponenten (Schranken, Lichtzeichen, ...). Kann der Bahnübergang nicht gesichert werden, reagiert die Zugsteuerung trotzdem wie gewünscht und der Zug passiert den Bahnübergang nicht.

Hoenicke spezifiziert in [Hoe99] den FFB mit der kombinierten Spezifikationssprache CSP-OZ-DC [HO02b, HO02a]. In dem integrierten Ansatz dient CSP (*Communicating Sequential Processes*, [Hoa85]) zur Spezifikation des Kommunikationsverhaltens, OZ (Object-Z, [Smi00]) zur Beschreibung der Operationen auf Daten und den Änderungen im Zustandsraum bei Zustandsübergängen und schließlich DC (*Duration Calculus*, [ZH97, ZHR91]) zur Beschreibung des Realzeitverhaltens des Modells. Der DC ist über einem kontinuierlichen Zeitmodell definiert. Deshalb basiert in [Hoe99] die Berechnung der Bremskurve auch auf einem kontinuierlichen Zeitmodell, während wir ein diskretes Zeitmodell gewählt haben.

10.4 Zusammenfassung

Die formale FTA der Fallstudie FFB zeigt, dass der in dieser Arbeit entwickelte Ansatz zur formalen Sicherheitsprüfung hochkritischer Systeme möglich ist. Dies gilt auch für zustandsunendliche Systeme, deren Vollständigkeitsbedingungen nicht mehr automatisch nachgewiesen werden können. Die Spezifikation des FFBs ist zustandsunendlich, denn die Position und Geschwindigkeit des Zugs sind über natürlichen Zahlen mit nicht endlichem Wertebereich definiert. Desweiteren sind z. B. für die Bestimmung des Bremseinsatzpunk-

tes komplexe Prädikate über Funktionen mit Multiplikation und Division notwendig, die nicht von Entscheidungsprozeduren über natürlichen Zahlen gelöst werden können. Dazu benötigen wir die interaktive Statechart-Verifikation, deren Anwendbarkeit hier gezeigt wurde.

Die Bemerkung in Abschnitt 10.2.1 zeigt aber, dass die Möglichkeit zur Modularisierung der Beweise eine wichtige Weiterentwicklung ist, um die Komplexität der Beweise über parallelen Statecharts zu reduzieren. Problemlos gestaltet sich die interaktive Verifikation über *Oder*-Zustände, da dort die Kombinationsmöglichkeiten der Übergänge durch Priorisierungsregeln beschränkt sind und die Beweise deshalb einen geringeren Verzweigungsgrad haben.

KAPITEL 11

Werkzeugunterstützung

Wir haben ein Werkzeug geschaffen, mit dem wir die Anwendung der formalen FTA durchgängig unterstützen. Die Werkzeugunterstützung ist in das KIV-System integriert, das wir in Kapitel 6 schon kurz vorgestellt haben. Wir können nun für die formale FTA dynamische Systeme mit i) Statecharts spezifizieren, ii) einen Fehlerbaum erstellen, iii) den Fehlerbaum formalisieren und die Vollständigkeitsbedingungen erzeugen und schließlich iv) die Vollständigkeit des Fehlerbaums über dem Statechart-Modell nachweisen. Die Ergebnisse der Implementierungsarbeiten für diese Werkzeugunterstützung wollen wir an der Fallstudie des funkbasierten Bahnübergangs aus dem vorigen Kapitel zeigen.

In Abschnitt 11.1 besprechen wir anhand der Fallstudie, wie Statechart-Spezifikationen in KIV erstellt und Eigenschaften von Statecharts nachgewiesen werden. Dann stellen wir in Abschnitt 11.2 die Unterstützung für die formale FTA vor. Wir zeigen, wie wir in KIV Fehlerbäume erstellen, die Ereignisse formalisieren und die Vollständigkeitsbedingungen generieren. Diese Vollständigkeitsbedingungen werden mit der Beweisunterstützung für Statecharts nachgewiesen. In Abschnitt 11.3 fassen wir die Implementierungsarbeiten zusammen.

11.1 Statecharts in KIV

In KIV können wir zwischen der Spezifikations- und der Beweisebene unterscheiden. Auf Spezifikationsebene beschreiben wir das Verhalten eines Systems mit Statecharts. Dann können wir über dieser Statechart-Spezifikation Beweise führen. Dazu verlassen wir die Spezifikationsebene, laden eine bestimmte Spezifikation und gelangen in die Beweisebene. Dort können wir Lemmata erstellen, verwalten und deren Korrektheit nachweisen. Wir zeigen nun, wie wir die FFB-Fallstudie in KIV mit Statecharts spezifiziert und Eigenschaften nachgewiesen haben.

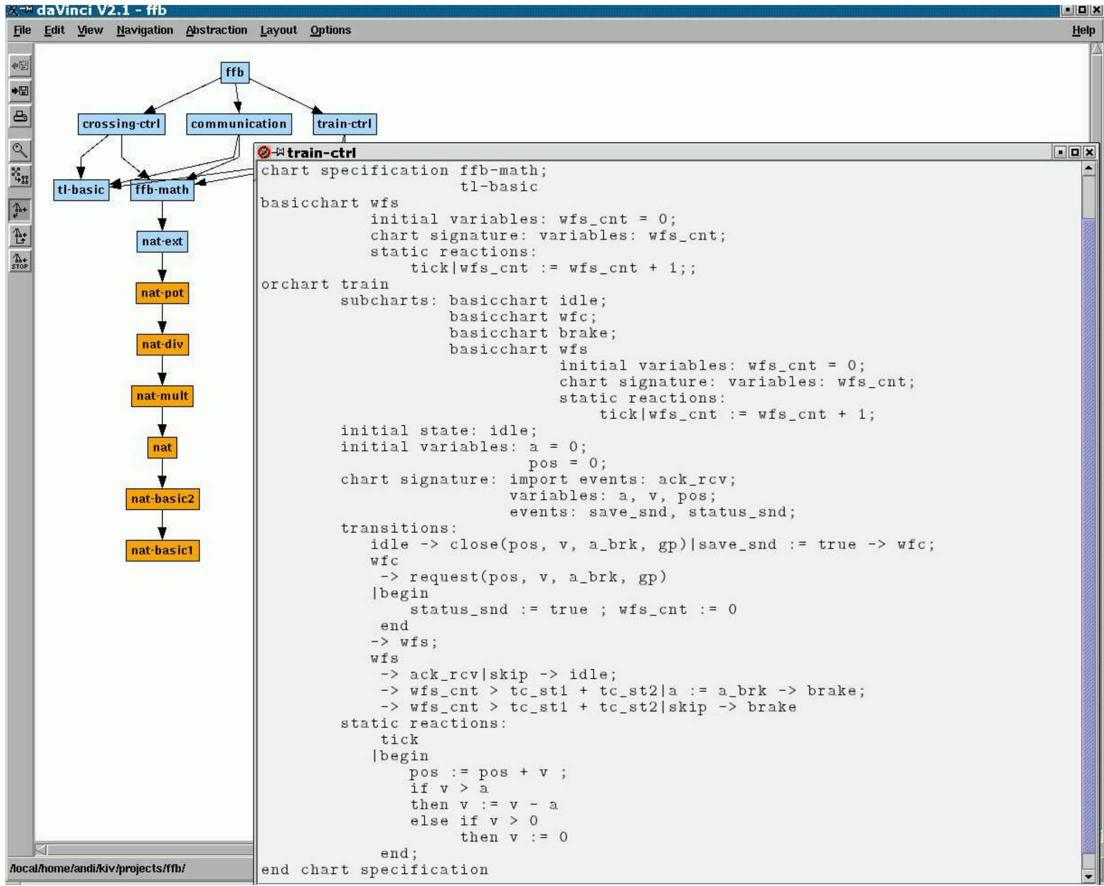


Abbildung 11.1: Entwicklungsgraph des FFBs

11.1.1 Spezifikation des funkbasierten Bahnübergangs

Strukturierte Spezifikationen werden in KIV durch einen Entwicklungsgraphen repräsentiert. In Abbildung 11.1 sehen wir im Hintergrund den Entwicklungsgraphen für den FFB. Die Spezifikationen *nat-*... stammen aus einer Bibliothek häufig benötigter Datentypen. Darin sind u. a. die natürlichen Zahlen mit den üblichen Funktionen $+$, $*$, ... spezifiziert. Darauf aufbauend wurde die Spezifikation *ffb-math* erstellt. Sie spezifiziert die Funktionen $close(pos, v, a, gp)$ und $request(pos, v, a, gp)$ für die Berechnung des Einschaltpunkts und der Position, an der die Statusabfrage gesendet wird (siehe Abschnitt 10.1.2 für Details). Die Spezifikationen *nat-*... und *ffb-math* sind algebraische Spezifikationen, die Datentypen, Funktionen und Prädikate, die in den Statechart-Spezifikationen verwendet werden, zur Verfügung stellen. Die Spezifikation *tl-basic* ist ein technisches Hilfsmittel und stellt notwendige Prädikate für temporallogische Beweise bereit.

Über diesen algebraischen Spezifikationen haben wir die drei Statechart-Spezifikationen *crossing-ctrl*, *communication* und *train-ctrl* erstellt. Sie entsprechen den *Oder*-Zuständen aus Abbildung 10.1 des vorigen Abschnitts und sind in der Spezifikation *ffb* als *Und*-

Zustand zusammengefasst. Die Syntax für Statechart-Spezifikationen in KIV sehen wir für die Spezifikation *train-ctrl* in der Abbildung 11.1 rechts. Wir sehen die Definition des *Oder*-Zustands *train*, der die *Basis*-Zustände *idle*, *wfc*, *brake* und *wfs* enthält. Der Unterzustand *wfs* spezifiziert eine statische Reaktion, die in jedem Makro-Schritt (durch *tick* ausgelöst) die Variable *wfs_cnt* erhöht. Wir benutzen diese Variable um den verzögerten Übergang von *wfs* nach *brake* zu beschreiben. Dies geschieht am Ende des Abschnitts **transitions** mit dem Übergang von *wfs*, der Aktivierungsbedingung $wfs_cnt > tc_st1 + tc_st2$ und der Aktion $a := a_brk$ in den Endzustand *brake*. Der Übergang findet um $tc_st1 + tc_st2$ Makro-Schritte verzögert statt und setzt die aktuelle Verzögerung auf den Wert a_brk, der maximalen Bremsleistung. Der Übergang $wfs \rightarrow wfs_cnt > tc_st1 + tc_st2 \mid skip \rightarrow brake$ beschreibt den Fehlerübergang zur Spezifikation des Bremsausfalls (*skip* ist das leere Programm, das eben a nicht auf a_{brk} setzt). Die restlichen Einträge in diesem Abschnitt beschreiben die weiteren Übergänge aus Abbildung 10.1. Schließlich beschreibt der Abschnitt **static reactions** die Position und Geschwindigkeit des Zuges, die sich in jedem Makro-Schritt ändern.

Die Spezifikationen *communication* und *crossing-ctrl* haben wir analog erstellt und damit das gesamte Verhalten des FFBs mit Statecharts spezifiziert. Die Timeout-Ereignisse wurden dazu über das *tick*-Ereignis modelliert, so wie wir es in Abschnitt 6.2.2 vorgestellt haben. Die vollständige KIV-Spezifikation für den FFB befindet sich in Anhang D. Über den Statechart Spezifikationen können wir nun Lemmata formulieren, die temporallogische Eigenschaften der Statecharts beschreiben. Dazu laden wir eine Spezifikation und gelangen in die Beweisebene, die wir im folgenden Abschnitt beschreiben.

Für die Spezifikationsunterstützung von Statecharts über algebraischen Spezifikationen haben wir KIV wie folgt erweitert:

- Implementierung eines Spezifikations-Typs *chartspec*

  Statecharts werden in *chartspec* Spezifikationen textuell beschrieben.

- Parsen der textuellen Syntax

  Zum Einlesen der Statechart Syntax, haben wir die KIV-Grammatik erweitert. Die Erweiterungen der Grammatik findet sich im Anhang F.

- interne KIV Datenstruktur für Statecharts

  Die Statechart-Spezifikationen werden in eine interne Datenstruktur gewandelt, die für die weiteren Berechnungen verwendet wird.

11.1.2 Beweisunterstützung

Statecharts sind Formeln, die eine Menge möglicher Abläufe beschreiben. Temporale Eigenschaften dieser Abläufe werden in Lemmata formuliert. Zu zeigende Eigenschaften können z. B. aus einer FTA stammen und Ursache/Wirkungs-Beziehungen zwischen Ereignissen beschreiben. Die Lemmata über Statecharts werden wie folgt spezifiziert.

$$[chart\ train], idle., \neg\ tick. \vdash \neg\ ((\neg\ (\text{true}; ack_{rcv}.; \text{true}); (tick. \wedge pos. > gp \wedge idle.; \text{true})),$$
$$\Box \neg\ (tick. \wedge pos. > gp \wedge idle.)$$

Der Punkt hinter einer Variable $x.$ kennzeichnet, dass es sich um eine flexible Variable x handelt und auf den aktuellen Wert verwiesen wird. $x.$ ist die KIV-Syntax für $\dot{x}$ aus Abschnitt 6.1.4. Die 1-fach und 2-fach gestrichenen Variablen werden auch in KIV mit x' bzw. x'' gekennzeichnet. Das Lemma beschreibt die Ursache/Wirkungs-Beziehung $\neg\,(\neg\,\Diamond_{x} ack_{rcv}.; \Diamond_{i}(tick. \wedge pos. > gp \wedge idle))$ mit der Ursache $ack_{rcv}.$ und der Wirkung $tick. \wedge pos. > gp \wedge idle.$. In KIV werden die Operatoren $\Diamond_{x}\,\varphi$ und $\Diamond_{i}\,\varphi$ nicht direkt unterstützt und deshalb durch Ihre Definition $\mathrm{true}; \varphi; \mathrm{true}$ bzw. $\varphi; \mathrm{true}$ spezifiziert. Der Chop-Operator $(\varphi; \psi)$ ist nur in einer „schwachen“ Form implementiert, so dass ψ nicht notwendigerweise eintreten muss. Deshalb entspricht in KIV $(\varphi; \psi) \wedge \Diamond\psi$ der üblichen Definition des Chop-Operators und durch Negation erhalten wir mit $\neg\,(\varphi; \psi) \vee \Box\neg\,\psi$, die obige Definition einer Ursache/Wirkungs-Beziehung.

Das obige Lemma verlangt nun für das Statechart *train* den Nachweis der Eigenschaft: es ist nicht möglich, dass sich in einem Makro-Schritt (*tick.*) der Zug hinter dem Bahnübergang befindet ($pos. > gp$) und weiterfährt (*idle.*), ohne zuvor eine Freigabe $ack_{rcv}.$ bekommen zu haben. Die möglichen Abläufe, die *train* beschreibt, stehen als Voraussetzungen (durch [*chart train*] dargestellt) im Antezedenten des Lemmas. Es sind nur diejenigen Abläufe gültig, in denen *idle.* und $\neg\, tick.$ im Anfangszustand gilt.

Die Eingabe eines solchen Lemmas erfolgt über eine Textdatei und wird dann in die interne Verwaltung von Beweisverpflichtungen eingelesen. Dabei ist [*chart train*] eine Referenz auf die Statechart-Formel *train*, die erst bei Beweisbeginn aus der Spezifikation erzeugt wird. Dieses Vorgehen entbindet den Benutzer davon, die Zustandsübergangsrelation, die in der Statechart-Spezifikation beschrieben ist, für jedes Lemma neu einzugeben. Nach dem Auflösen der Referenz erfolgt der Nachweis dieses Lemmas mit der Beweistechnik, die wir in Abschnitt 6.3.2 ausführlich beschrieben haben.

Für die Beweisunterstützung von Statecharts in KIV haben wir folgende Implementierungsarbeiten durchgeführt:

- Erweiterung der Formelmengen um Statechart-Formeln

  Eine Statechart Formel enthält neben der statischen Übergangsrelation, die sie repräsentiert, noch Verwaltungsinformationen, die ein effizientes Bearbeiten von Statechart-Formeln erlaubt. Zustände und Ereignisse werden durch boolesche Variablen beschrieben.

- Beweisregeln für Statecharts

  In Abschnitt 6.3.1 haben wir die Verifikation von Statecharts mit vollständiger und partieller Zustandskonfiguration vorgestellt. Die in KIV implementierten Beweisregeln unterstützen die allgemeinere Variante der partiellen Zustandskonfiguration und damit auch den Spezialfall mit vollständiger Zustandskonfiguration. Alle in dieser Arbeit vorgestellten Regeln (Initialisierung, Konsistenz und Ausführung) sind in KIV implementiert. Die wichtigsten Teile für die Ausführungsregeln von Statecharts sind

  – die Schrittberechnung von Statecharts und

  - das Beschreiben der Statechart-Schrittausführung mit sequentiellen Programmen.

- Korrektheitsmanagement für Statecharts

  Statechart-Spezifikationen sind in das Änderungsmanagement von KIV [BRSS99] eingebunden. Änderungen in Statechart-Spezifikationen werden erkannt und Beweise von Lemmata über geänderten Statecharts ungültig.

- Beweisunterstützung für Statecharts über nicht endliche Datentypen

  Wir können in Statecharts beliebige, algebraisch spezifizierte Datentypen, Funktionen und Prädikate verwenden. Z. B. beschreiben wir die Position und Geschwindigkeit des Zugs mit natürlichen Zahlen, die einen unendlichen Wertebereich haben. Prädikate über natürliche Zahlen (z. B. *close* und *request*) werden in der Zugsteuerung *train-ctrl* benutzt. Damit ist es nun möglich, Statechart-Beweise über nicht zustandsendlichen Systemen werkzeugunterstützt zu führen.

UML Statecharts

Die Statechart-Formeln dienen auch als Grundlage für die Implementierung der UML-Semantik von Statecharts [OMG03] in KIV. Die notwendigen Regeln zum Ausführen von UML-Statecharts wurden in der Diplomarbeit von Bäumler [Bäu03], Ludwig-Maximilians-Universität München, Lehrstuhl Programmierung und Softwaretechnik, Prof. M. Wirsing, in Zusammenarbeit mit dem Lehrstuhl Softwaretechnik und Programmierspachen, Universität Augsburg, Prof. W. Reif, erstellt.

Da Statecharts im Beweiskalkül als Formeln behandelt werden und für temporallogische Beweise die Strategie der symbolischen Ausführung benutzt wird, genügt es, für die Verifikation von UML-Statecharts, eine zusätzliche Kalkülregel zum Ausführen der Statecharts zu definieren. Diese Kalkülregel wurde in der Arbeit von Bäumler erstellt und führt die Statecharts nach der UML-Semantik aus. Dazu war i) eine geänderte Schrittberechnung notwendig, da sich UML- und STATEMATE-Statecharts in der Priorisierung von Übergängen unterscheiden, und ii) eine geänderte Schrittausführung, denn in der UML-Semantik nach [LP99] werden Ereignisse, über eine Event-Queue gesteuert, einzeln abgearbeitet. Die Ausführung eines Schrittes wird, wie in unserem Ansatz, durch sequentielle Programme beschrieben. Die Event-Queue konnte als algebraisch-spezifizierter Datentyp realisiert werden.

In der Diplomarbeit wurden die Kalkülregeln für UML-Statechart implementiert. Somit können wir in KIV Beweise über Statecharts sowohl nach der STATEMATE- als auch nach der UML-Semantik führen.

11.2 Fehlerbäume in KIV

Für die formale FTA haben wir in KIV die Möglichkeit geschaffen, formale Fehlerbäume zu erstellen. Fehlerbäume beschreiben Anforderungen an eine Spezifikation und werden

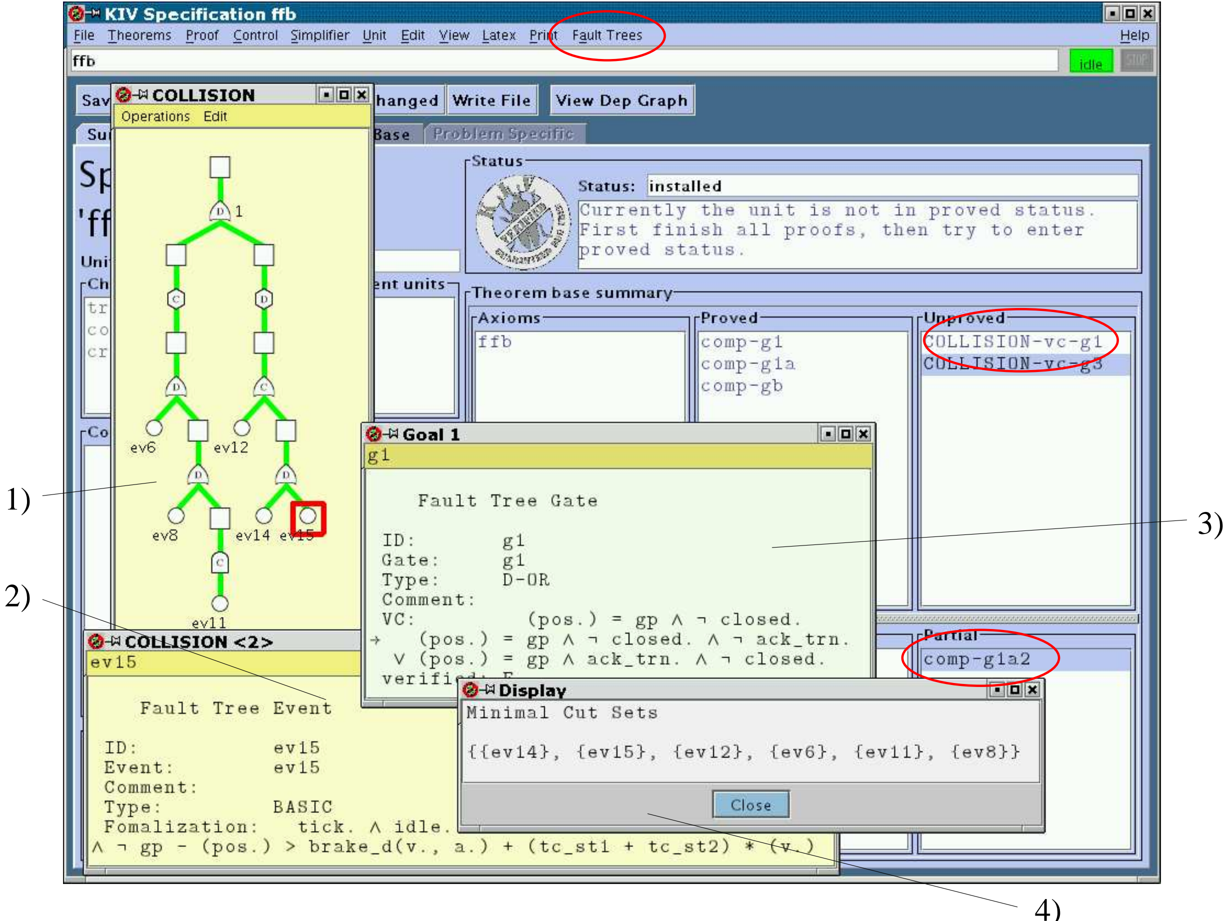


Abbildung 11.2: Fehlerbäume in KIV

deshalb auf der Beweisebene über einer Spezifikation erstellt. Dies geschieht über das Menü *Fault Tree*, das in Abbildung 11.2, oben in der Mitte umrandet, zu sehen ist. Ein Fehlerbaum (Fenster 1) kann mit den üblichen Änderungsfunktionen bearbeitet werden. Es können Gatter und Ereignisse hinzugefügt, gelöscht und geändert werden. Für die formale FTA kennzeichnen wir die verschiedenen Zerlegungs- und Ursache/Wirkungs-Gatter explizit mit D für *decomposition* bzw. (A)C für *(asynchronous) cause/consequence*.

Ereignisse in Fehlerbäumen können wir mit Kommentaren informell beschreiben und zusätzlich formal präzise definieren (siehe Fenster 2, Formalization). Die formale Definition eines Ereignisses verwenden wir in den Gatterbedingungen, die, in Abhängigkeit vom Gattertyp nach Abschnitt 5.3.1, automatisch erzeugt werden (Fenster 3, VC für *verification condition*). Diese Bedingungen sind Beweisverpflichtungen der Spezifikation und erscheinen im *Specification* Fenster als nachzuweisende Lemmata. In der Abbildung 11.2 ist rechts in der Mitte z. B. die Beweisverpflichtung *COLLISION-vc-g1* als noch nicht bewiesen gekennzeichnet (umrandet in Abschnitt *Unproved*). Ein Hilfslemma für diese Behauptung,

das teilweise bewiesen wurde, ist in Abschnitt *Partial* darunter zu sehen. Bewiesene Lemmata stehen in Abschnitt *Proved*, links neben *Unproved*. Bei Änderungen im Fehlerbaum oder Änderungen der Formalisierung der Ereignisse, die ein Fehlerbaumgatter beeinflussen, werden die entsprechenden Beweispflichten neu erzeugt, damit sie konsistent zum Fehlerbaum sind. Schließlich zeigt Fenster 4 die minimalen Schnittmengen für den Fehlerbaum. Wie man sieht, besteht der Fehlerbaum nur aus *Oder*-Gattern und alle minimalen Schnittmengen sind einelementig. Für die formale FTA des FFBs haben wir den Fehlerbaum, der in Abbildung 11.2 zu sehen ist, erstellt. Der Fehlerbaum entspricht dem aus Abbildung 10.3, Abschnitt 10.2.

Für formale Fehlerbäume in KIV haben wir folgende Erweiterungen implementiert:

- Erstellen, Ändern und Löschen von Fehlerbäumen

  Ein Fehlerbaum wird über einem dynamischen System (Statecharts, temporale Spezifikation oder parallele Programme) erstellt. Das Einfügen, Löschen und Ändern von Knoten im Fehlerbaum erfolgt über kontextsensitive Menüs.

- Formalisieren von Ereignissen

  Neben einer textuellen Beschreibung der Ereignisse können die Ereignisse mit Formeln der zugrundeliegenden Spezifikation formalisiert werden.

- automatisches Generieren der Beweispflichten

  Aus dem zugrundeliegenden dynamischen System, den formalisierten Ereignissen und den Formel-Schemata für die Fehlerbaumgatter werden Beweispflichten für den Fehlerbaum generiert.

- Korrektheitsmanagement

  Bei Änderungen im Fehlerbaum werden betroffene Beweisverpflichtungen ungültig und müssen neu generiert werden.

- Berechnen der minimalen Schnittmengen

  Für die qualitative Analyse berechnen wir die minimalen Schnittmengen für einen Fehlerbaum.

11.3 Statistik und Resultate

Die Implementierung der Werkzeugunterstützung erfolgte größtenteils in PPL (*proof programming language*). PPL ist eine typsichere funktionale Sprache, ähnlich zu Standard ML [MTH89]. PPL selbst und einige Teile von KIV (z. B. Datentypen) sind aus Gründen der Effizienz in Lisp implementiert. Die Oberfläche von KIV ist in Java geschrieben.

Die gesamte Implementierung, die im Rahmen dieser Arbeit entstanden ist, beläuft sich auf knapp 6000 Zeilen Code. Der geringste Teil mit ca. 600 Zeilen Java-Code entfällt auf die Darstellung der Fehlerbäume. Dies liegt vor allem daran, dass Routinen für die

Darstellung von Bäumen (KIV benutzt Beweisbäume zur Darstellung der Beweise) wiederverwendet werden konnten. Die restliche Implementierung besteht aus ca. 1500 Zeilen Lisp-Code für die Statechart-Datentypen und über 3000 Zeilen PPL-Code. Der PPL-Code teilt sich gleichmäßig auf die Implementierung der Fehlerbäume und der Kalkülregeln für die Verifikation von Statecharts auf. Dazu kommen noch einige Zeilen Parser-Code, der mit Lex [LMB92] erstellt wurde.

Mit der entstandenen Werkzeugunterstützung haben wir eine formale FTA für die FFB-Fallstudie durchgeführt. Dazu wurde die Statechart-Spezifikation aus Abbildung 10.1 in KIV übertragen, ein Fehlerbaum für die Gefährdung Kollision erstellt und die Ereignisse des Fehlerbaums formalisiert. Aus dem Fehlerbaum und den formalisierten Ereignissen wurden Vollständigkeitsbedingungen erzeugt und nachgewiesen. Details zur Fallstudie finden sich im Kapitel 10 und im Anhang D.

KAPITEL 12

Zusammenfassung

In dieser Arbeit haben wir eine durchgängige Methode für die formale Analyse sicherheitskritischer Systeme erstellt, die formale FTA. Sie integriert die klassische FTA in formale Methoden und verknüpft damit bewährte Sicherheitsanalysetechniken aus den Ingenieurwissenschaften mit Techniken der Softwareanalyse. Die Grundlage für die formale FTA ist eine präzise Beschreibung des sicherheitskritischen Systems mit formalen Modellen. Hierauf aufbauend werden Fehlerbäume für mögliche Systemgefährdungen erstellt und mit formalen Beweistechniken bezüglich des formalen Modells validiert. Der Sicherheitsgewinn der formalen FTA besteht darin, dass wir die beiden Sicherheitsziele *funktionale Korrektheit* und *Ausfallsicherheit* in einem einheitlichen Rahmen – einer Temporallogik – auf ein und demselben formalen Modell betrachten und dadurch hochsichere Modelle für sicherheitskritische Systeme erhalten.

Bei einer typischen formalen FTA erstellen wir aus der Anforderungsbeschreibung für ein System ein formales Modell. Wir betrachten üblicherweise softwarebasierte, dynamische Systeme und haben uns deshalb für Statecharts, oder allgemein Zustandsübergangssysteme, als Spezifikationssprache entschieden. Dieses formale Modell dient im Folgenden als präzise Beschreibung für die sicherheitstechnische Analyse durch die FTA. Als Grundlage für die Validierung der FTA benötigen wir eine formale Semantik, die Beweispflichten für die einzelnen Fehlerbaumgatter definiert. Die Beweispflichten beschreiben für jedes Gatter Bedingungen zwischen dem Ausgangsereignis und den entsprechenden Unterereignissen. Wir betrachten dynamische Systeme und erhalten deshalb temporallogische Beweispflichten. Um das Auftreten der Ereignisse im formalen Modell identifizieren zu können, müssen wir die Ereignisse formalisieren. Für den Nachweis instantiieren wir im konkreten Fehlerbaum die Beweispflichten entsprechend der Gatter mit den formal definierten Ereignissen. Der erfolgreiche Nachweis dieser Beweispflichten garantiert, dass es keine weiteren Ursachen für die mit der FTA untersuchte Gefährdung gibt, die FTA ist vollständig. Für eingeschränkte Systeme sehen wir den automatischen Nachweis der Beweispflichten mit Modellprüfern vor. Für allgemeine Systeme weisen wir die FTA-Beweispflichten mit interaktiver Verifikation nach. Als Ergebnis dieser Analyse erhalten wir ein hochsicheres Systemmodell mit einer vollständigen FTA.

Der wesentliche Beitrag dieser Arbeit besteht in

1. der Formalisierung der FTA in ITL und der formalen Fundierung der FTA-Semantik,
2. einem Beweiskalkül für Statecharts über algebraischen Spezifikationen zum Nachweis der FTA-Beweispflichten und
3. einer Methodik und Werkzeugunterstützung für die formale FTA.

Wir haben in dieser Arbeit eine Semantik für die FTA in ITL erstellt, die Schwächen bisheriger Semantiken aufgreift und beseitigt (Kapitel 5). Dies ist die erste FTA-Semantik, für die das sogenannte minimale Schnittmengen-Theorem auch für Ursache/Wirkungs-Gatter nachgewiesen wurde. Deshalb kann in unserem Ansatz die formale FTA zur sicherheitstechnischen Bewertung eines formalen Modells eingesetzt werden und wir können verschiedene formale Modelle bezüglich der Ausfallsensitivität ihrer Komponenten vergleichen. Den Nachweis des minimalen Schnittmengen-Theorems haben wir in KIV über einer algebraisch spezifizierten ITL geführt (Anhang B).

Aus der ITL-Semantik können wir für eingeschränkte Systeme äquivalente CTL-Beweispflichten für die formale FTA ableiten (Kapitel 7). Viele Modellprüfer benutzen CTL-Formeln für die Spezifikation von Beweispflichten, und wir erhalten damit einen auf CTL basierenden Rahmen für die automatische Validierung der formalen FTA. Die Anwendung haben wir an einem bekannten Beispiel, dem „Drucktank“ aus dem *Fault Tree Handbook*, demonstriert.

Schließlich haben wir zur Fundierung der ITL-Semantik verschiedene Formalisierungen der FTA formal verglichen (Kapitel 8). Dazu wurden die untersuchten FTA-Semantiken in KIV algebraisch spezifiziert. Es hat sich herausgestellt, dass die bekannten Ansätze aus der Literatur Schwächen bei temporalen Ereignissen haben. Wenn wir jedoch nur punktuelle Ereignisse betrachten (Ereignisse ohne Dauer), sind diese Ansätze ebenso wie die CTL-Bedingungen äquivalent der ITL-Semantik. Diese Äquivalenz konnten wir über den spezifizierten FTA-Semantiken nachweisen. Damit haben wir uns mit unterschiedlichen Formalisierungen der FTA intensiv auseinandergesetzt und eine Semantik in ITL erstellt, die Schwächen bekannter Semantiken behebt.

Als Zweites möchten wir die Entwicklung der Spezifikations- und Verifikationsunterstützung für STATEMATE-Statecharts über algebraischen Spezifikationen hervorheben. Wir können damit Statecharts über Datentypen mit unendlichem Wertebereich spezifizieren. Diese Datentypen werden in den Aktivierungsbedingungen und Aktionen von Übergängen benutzt. Aktionen sind als sequentielle Programme formuliert. Über den so definierten Statecharts können wir Beweise temporallogischer Aussagen führen (Kapitel 6). Soweit bekannt, ist dies der erste Kalkül zur interaktiven Verifikation von STATEMATE-Statecharts über algebraischen Spezifikationen.

Wichtige Kriterien bei der interaktiven Verifikation sind die Verständlichkeit und Nachvollziehbarkeit der Beweis für den Benutzer. Dies erreichen wir durch das schrittweise Abarbeiten der Übergangsrelation, die ein Statechart beschreibt. Damit fügen wir uns nahtlos in die Strategie der temporallogischen Verifikation in KIV ein. Hier werden Beweise über

parallele Programme und ITL-Formeln ebenfalls durch schrittweises Abarbeiten und Induktion geführt. Es ist nun möglich, in ITL beschriebene Eigenschaften über Statechart-Spezifikationen nachzuweisen. Wir können damit die ITL-Bedingungen der Fehlerbaumgatter über Statechart-Spezifikationen interaktiv in KIV nachweisen.

Dies führt uns zum dritten Punkt. Wir haben eine Methodik für die formale FTA entwickelt (Kapitel 9), deren Anwendung wir durch eine Werkzeugumgebung unterstützen (Kapitel 11). Neben der Modellierung eingeschränkter Systeme mit Zustandsübergangssystemen und der automatischen Validierung der FTA mit Modellprüfung in CTL sieht die Methodik der formalen FTA im Allgemeinen die Spezifikation von Systemen mit Statecharts vor. Wir können dann die FTA bezüglich ihrer ITL-Semantik in KIV validieren, indem wir die ITL-Beweispflichten für das Statechart-Modell nachweisen. Damit haben wir für die interaktive Verifikation einen Rahmen für die formale FTA auf Grundlage von ITL geschaffen. Die notwendige Werkzeugunterstützung für die formale FTA haben wir in KIV integriert und an Beispielen evaluiert. Wir können Fehlerbäume erstellen, die Ereignisse formalisieren und daraus die ITL-Beweispflichten für die formale FTA automatisch generieren. Diese weisen wir dann mit dem Statechart-Kalkül nach. Als Referenzbeispiel haben wir in Kapitel 10 eine formale FTA für die Fallstudie zur dezentralen Ansteuerung von Bahnübergängen, den FFB, mit der in KIV entstandenen Werkzeugunterstützung durchgeführt.

Unsere Arbeit zur formalen FTA ist bei anderen Forschergruppen bereits auf großes Interesse gestoßen und es haben sich daraus zwei Diplomarbeiten ergeben. Die Diplomarbeit von Schäfer (Universität Oldenburg, Prof. Dr. E.-R. Olderog, Abteilung Semantik, [Sch01, Sch03]) benutzt unsere ITL-Semantik der FTA als Grundlage für die Modellprüfung von Fehlerbäumen im Duration Calculus. Die Diplomarbeit von Bäumler (Ludwig-Maximilians-Universität München, Lehrstuhl Programmierung und Softwaretechnik, Prof. M. Wirsing [Bäu03]) setzt auf unseren Statechart-Spezifikationen auf und hat eine Kalkülregel für die UML-Semantik von Statecharts in KIV entwickelt, implementiert und an einer Fallstudie validiert.

Zusammenfassend ist in dieser Arbeit die formale FTA mit einer vollständigen, methodischen und werkzeugseitigen Unterstützung entstanden.

KAPITEL 13

Ausblick

Wir möchten die weitere Entwicklung der in dieser Arbeit entstandenen Techniken von zwei Seiten beleuchten. Einerseits betrachten wir den möglichen Einsatz der formalen FTA in zukünftigen Anwendungen und andererseits interessante Weiterentwicklungsmöglichkeiten in der Statechart-Verifikation.

13.1 Formale FTA

Die Sicherheitsanalyse softwarebasierter eingebetteter Systeme mit formalen Methoden rückt immer mehr in den Vordergrund. Eine Reihe von Standards [IEC91, IEC92, ESA91] schlagen bereits die Verwendung formaler Methoden für die Entwicklung softwarebasierter eingebetteter Systeme vor. Das Britische Verteidigungsministerium verlangt für softwarebasierte Verteidigungssysteme sogar zwingend die Anwendung formaler Methoden [MoD91]. Auch in der Evaluierung und Zertifizierung nach den Kriterien der ITSEC [ITS91] bzw. den Common Criteria [CCI99] sind in den höheren Stufen formale Methoden vorgeschrieben. Ein großes Problem bei der Abnahme formal analysierter Systeme ist die Schwierigkeit, formale Modelle und Beweise verständlich zu dokumentieren und kommunizieren.

Formale Dokumentation An dieser Stelle sehen wir ein großes Potential für die formale FTA. Durch die Formalisierung der FTA haben wir eine formale Methode erhalten, mit der einerseits Sicherheitseigenschaften softwarebasierter eingebetteter Systeme nachgewiesen werden können. Andererseits stellt die formale FTA die untersuchten Zusammenhänge in ihrer graphischen Notation verständlich dar. Damit überwindet die formale FTA die Kluft zwischen präziser, formaler Beschreibung und anschaulicher, verständlicher Darstellung. Die formale FTA kann so als Kommunikationsmittel zwischen den formalen Analytikern und den Sicherheitsexperten eingesetzt werden. Außerdem genießt die FTA den Vorteil, auf dem Gebiet der Sicherheitsanalyse bereits weit verbreitet zu sein. Wir denken, dass in Zukunft Sicherheitsexperten und Zertifizierungsstellen die formale FTA als präzise Sicherheitsanalysemethode akzeptieren werden.

Im Hinblick auf den Einsatz der formalen FTA zur Dokumentation formaler Methoden besteht noch eine weitere interessante Möglichkeit: Die Semantik von Fehlerbäumen ist nach unserer Definition eine temporale Formel. Wie parallele Programme oder Statecharts beschreibt ein Fehlerbaum Sequenzen von Belegungen oder Systemabläufe. Damit können wir Fehlerbäume auch zur Spezifikation dynamischer Systeme einsetzen. Wir können Eigenschaften dieser Spezifikation nachweisen und, wie für andere Systembeschreibungen auch, die Korrektheit möglicher Implementierungen bezüglich einer FTA-Spezifikation zeigen. Eine korrekte Implementierung muss die Vollständigkeitsbedingungen aller Fehlerbaumgatter erfüllen. Wenn wir die FTA zur Spezifikation softwarebasierter eingebetteter Systeme einsetzen, können wir die graphische Darstellung der Fehlerbäume als anschauliches Mittel für formale Systembeschreibungen nutzen.

Sicherheitsmaximierung Nun möchten wir noch zwei Erweiterungsmöglichkeiten der formalen FTA erwähnen, die wir bereits in einem Forschungsprojekt untersuchen [RST01, OT02]. Die erste betrifft das systematische Finden von möglichen Hardwarefehlern oder -ausfällen. Die Methodik der formalen FTA verlangt, dass Hardwarefehler oder -ausfälle, die in Fehlerbäumen zur untersuchten Gefahr beitragen, in der formalen Spezifikation berücksichtigt werden. Werden bei der Erstellung von Fehlerbäumen weitere Ursachen oder Komponentenausfälle, die zur untersuchten Gefährdung führen, übersehen, kann die Vollständigkeitsbedingungen der Fehlerbäume nicht gezeigt werden. Nun kann es aber sein, dass bestimmte Ausfälle oder Fehler weder bei der FTA entdeckt werden, noch im formalen Modell enthalten sind. Diese Ausfälle werden bisher bei der formalen FTA nicht erkannt. In [OR04a] wird ein Ansatz präsentiert, der systematisch alle Fehler- oder Ausfallmöglichkeiten für Komponenten in formalen Spezifikationen konstruiert. Dieser orthogonale Ansatz basiert weder auf der FTA noch auf dem formalen Systemmodell. Er entdeckt Fehler- oder Ausfallmöglichkeiten, die in der bisherigen Analyse noch nicht bedacht wurden und erhöht dadurch den Sicherheitsgewinn der formalen FTA.

Die zweite Erweiterung, die *Sicherheitsoptimierung* formaler Systeme [OR04b], betrifft den Einsatz formaler FTA zur quantitativen Sicherheitsanalyse. Neben dem Vergleich von Design-Alternativen (siehe Abschnitt 2.2.3), erlaubt die Sicherheitsoptimierung, ein System mit mehreren Gefährdungen optimal zu parametrieren. Optimal bedeutet in diesem Zusammenhang, dass die Summe aller Gefährdungen, das Gesamtrisiko des untersuchten Systems, möglichst gering ist.

Quantitative FTA und Fehlerbaumerzeugung Ein anderer Weg wird in einem Projektteil des Sonderforschungsbereichs *AVACS*[1] [Pro03] beschritten. Es wird die Verknüpfung quantitativer FTA mit formalen Modellen untersucht. Systemspezifikationen werden mit stochastischen Modellen, die auf Markov-Ketten [KS76] basieren, spezifiziert und eine quantitative FTA mit einem stochastischen Modellprüfer [HKMKS03] validiert. Dazu soll die von uns entwickelte Semantik der Fehlerbäume um quantitative Aspekte erweitert und

[1] Transregio-Verbundprojekt AVACS: Automatische Verifikation und Analyse komplexer Systeme, gefördert von der Deutschen Forschungsgemeinschaft.

in CSL (*continuous stochastic logic*, [ASSB96]) formalisiert werden. Die CSL-Bedingungen werden dann über den formalen Markov-Modellen nachgewiesen. Ebenfalls wird in diesem Projektteil ein Synthesealgorithmus entwickelt, der auf Arbeiten von Damm (Universität Oldenburg, Abteilung Sicherheitskritische Eingebettete Systeme) aus dem *ESACS*-Projekt aufsetzt ([Con01, Con02], siehe Abschnitt 5.5) und aus stochastischen Modellen automatisch Fehlerbäume erzeugt.

13.2 Statechart-Verifikation

Wir haben bisher die Möglichkeit geschaffen, Beweise über einzelne Statecharts zu führen. Die Fallstudie „funkbasierter Bahnübergang“ haben wir z. B. als *ein* Statechart spezifiziert, das die drei Komponenten Zug, Kommunikation und Bahnübergang als parallele Zustände enthält. Eine naheliegende Erweiterung ist, den Kalkül für Statecharts so zu erweitern, dass bei der Verifikation mehrere Komponenten betrachtet werden können, deren Verhalten durch jeweils ein separates Statechart beschrieben wird. Dann könnten wir den Zug, die Kommunikation und den Bahnübergang in verschiedenen Komponenten spezifizieren. Die komponentenbasierte Spezifikation gliedert sich gut in die Methodik der FTA ein, die Gefährdungen auf Ausfälle und Defekte einzelner Komponenten herunterbricht und diese Komponenten dann weiter untersucht. Beweispflichten, die bei der formalen FTA entstehen, können dann auf die einzelnen Komponenten verteilt und nachgewiesen werden.

Komponentenbasierte Verifikation In STATEMATE gibt es sogenannte *Activities*, um eine komponentenbasierte Entwicklung zu unterstützen. Activities beschreiben die Struktur eines Systems. Das Verhalten der Activities wird durch Statecharts beschrieben. Sehen

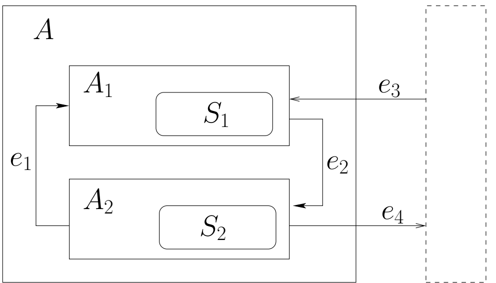


Abbildung 13.1: Activities in STATEMATE

wir uns dazu Abbildung 13.1 an. Die Systembeschreibung A besteht aus den beiden, durch Rechtecke dargestellten, Activities A_1 und A_2, deren Verhalten durch entsprechende Statecharts definiert wird. Der Datenaustausch zwischen den Komponenten erfolgt über definierte Schnittstellen, die durch Pfeile dargestellt werden. Die Pfeile sind mit den Daten beschriftet, die ausgetauscht werden, und geben die Kommunikationsrichtung der Daten an.

Die Komponente A_1 sendet das Ereignis e_2 an A_2 und A_2 das Ereignis e_1 an A_1. Die Umgebung wird durch ein gestricheltes Rechteck dargestellt, d. h. es wird das Umgebungsereignis e_3 eingelesen und e_4 an die Umgebung gesendet. Damit hat A_1 den Eingabeparameter e_1, e_3 und den Ausgabeparameter e_2. Im Allgemeinen können nicht nur Ereignisse, sondern auch Werte von Datenvariablen ausgetauscht werden. Im Unterschied zu parallelen Zuständen definieren Komponenten also eine Schnittstelle für den Austausch relevanter Daten, während parallele Zustände ihre gesamten Daten über Broadcast-Kommunikation austauschen.

Die FFB-Fallstudie können wir mit Komponenten wie in Abbildung 13.2 darstellen. Durch die spezifizierten Schnittstellen würde der Zug *Train* nur das Ereignis ack_{cr} empfangen, anstatt alle Ereignisse (auch *close*, *status*, err_{op}, ...), wie es bei einer Spezifikation mit parallelen Zuständen geschieht.

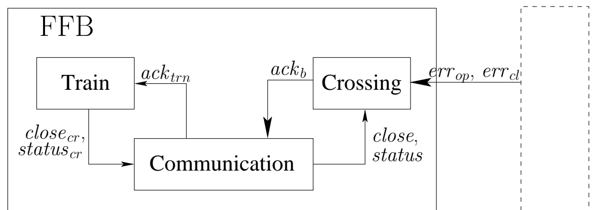


Abbildung 13.2: Komponentenbasierter FFB

Damm et al. beschreiben in [DJHP98] eine kompositionale Semantik für Statecharts. Die Semantik erlaubt Eigenschaften des Gesamtsystems über einzelnen Systemkomponenten nachzuweisen. Mit dem *assumption commitment* Paradigma nach [AL89] werden Abhängigkeiten zwischen Systemkomponenten aufgelöst und damit die Komplexität der Beweise erheblich reduziert. Die Schnittstellen kennzeichnen externen Ereignisse, die innerhalb eines Statechart betrachtet werden müssen. Dabei muss zwischen globalen Umgebungsereignissen und Ereignissen, die nur für die betrachtete Komponente aus der Umgebung stammen, jedoch für das Gesamtsystem lokal sind, unterschieden werden. Wir nennen ein Ereignis des zweiten Typs lokales Umgebungsereignis.

Zur Veranschaulichung betrachten wir uns hierzu nochmals die Komponente A_1, die in dem System A die Eigenschaft φ erfüllen soll. Wir möchten einen Beweis der Art $[\mathit{chart}\ A_1\ \mathit{SUD}\ A] \vdash \varphi$[2] führen, also einen Beweis für die Komponenten A_1 im Kontext von A. In jedem Mikro-Schritt könnte e_1 von A_2 geändert werden. Da A_1 von e_1 abhängt, muss die Möglichkeit der Änderung in der Semantik berücksichtigt werden. Globale Umgebungsereignisse können nach Definition aber nur in Makro-Schritten geändert werden. Für die komponentenbasierte Verifikation ist also eine Erweiterung derart notwendig, dass zwischen lokalen und globalen Umgebungsereignissen unterschieden werden kann.

Eine zweite Erweiterung ist im Hinblick auf die Spezifikation des Gesamtsystems notwendig. Ein Statechart bietet bereits eine Schnittstelle (siehe **import** Slot einer Statechart-Spezifikation, z. B. Abbildung 6.1, Seite 79) und kann als Komponente betrachtet werden.

[2]SUD: system under design.

Um aus einzelnen Komponenten ein Gesamtsystem zu spezifizieren, benötigen wir einen neuen Kompositionsoperator || mit entsprechender Semantik, der sich vom Paralleloperator | für die Spezifikation von *Und*-Zuständen in Statecharts unterscheidet. Das Activtiy *FFB* des funkbasierten Bahnübergangs würde dann folgendermaßen definiert werden:

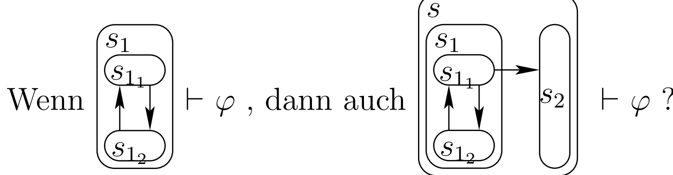
$$FFB := Train \,||\, Communication \,||\, Crossing$$

Die Semantik von || muss den Datenaustausch zwischen den Unterkomponenten und der Umgebung entsprechend wiedergeben.

Die semantischen Grundlagen für beide Erweiterungen wurden bereits in [DJHP98] gelegt. In unserem Kalkül basiert die Ausführung einzelner Statecharts bereits auf der Semantik aus [DJHP98] und für die komponentenbasierte Verifikation muss im Wesentlichen zusätzlich der Datenaustausch zwischen den Komponenten und ein entsprechender Kompositionsoperator definiert werden. Unserer Ansicht nach stellt die Erweiterung der Statechart-Verifikation auf die Verifikation über Komponenten kein semantisches oder technisches Problem dar. Die Kodierung der Schrittberechnung in sequentielle Programme scheint flexibel genug zu sein, um auch die zusätzlichen Berechnungen für den Datenaustausch zwischen den Komponenten zu integrieren (durch *Monitore* in [DJHP98] realisiert).

Als Vorarbeit für die komponentenbasierte Verifikation haben wir bereits einige Erweiterungen von Statecharts in KIV evaluiert. Wir können globale und lokale Umgebungsereignisse unterscheiden. Im Kalkül beachten wir, dass sich lokale Umgebungsereignisse zu Mikro-Schritten, globale Umgebungsereignisse nur zu Makro-Schritten ändern. Mit dieser Erweiterung können wir Beweise über einzelne Komponenten A_1 führen, die in ein komplexeres System A eingebunden sind ($[chart\ A_1\ SUD\ A] \models \varphi$). Erste Versuche zur kompositionalen Verifikation sind vielversprechend und unterstützen unsere These, dass die Erweiterung der Statechart-Verifikation auf Komponenten, die zu komplexen Systemen kombiniert werden können, mit unserem Ansatz durchführbar ist.

Modulare Statechart-Verifikation Eine naheliegende Idee ist, die komponentenbasierte Verifikation auch auf einzelne Statecharts oder Zustände zu übertragen. Erfüllt ein Unterzustand s_1 eines Zustandes s die Eigenschaft φ, so soll diese Eigenschaft auf den Zustand s übertragen werden können. Die Frage ist:

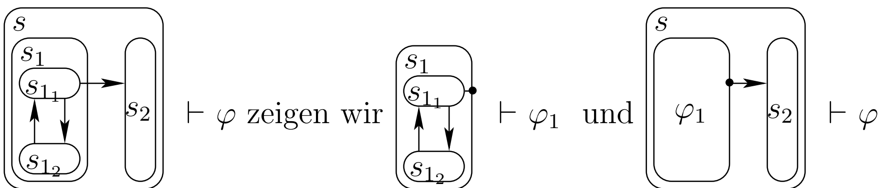


Ein Problem sind Übergänge, deren Ausgangs- und Endzustand in verschiedenen Regionen liegen. Können wir z. B. für den Zustand s_1 zeigen, dass er sich in jedem Makro-Schritt im Zustand s_{1_1} befindet, kann diese Eigenschaft nicht notwendigerweise auf den Zustand s, mit dem Unterzustand s_1, übertragen werden. Die besprochene Eigenschaft kann durch den Übergang von s_{1_1} nach s_2 verletzt werden (wir können uns nun auch in s_2 befinden, wenn

ein Makro-Schritt ausgeführt wird). Wie schon in [HN96] bemerkt, ist nicht einmal die Syntax für Zustände kompositional, denn durch den zusätzlichen Übergang von s_{1_1} nach s_2 in s ändert sich auch die Struktur von s_1. Um trotzdem eine kompositionale Semantik für Zustände zu erhalten, werden häufig nur eingeschränkte Zustandsdiagramme betrachtet. In [LvdBC00] werden beispielsweise Übergänge über verschiedene Regionen hinweg verboten.

Eine Lösung für Zustände mit Übergängen über verschiedene Regionen könnte die Strategie des modularen Beweisens sein, wie sie in der Programmverifikation nach Balser [Bal04] angewendet wird. In der Semantik von Balser ist es möglich, für ein *interleaved paralleles* Programm $\alpha_1 \mid\mid \alpha_2$ statt $\alpha_1 \mid\mid \alpha_2 \vdash \varphi$ die beiden Aussagen i) $\alpha_1 \vdash \varphi_1$ und ii) $\varphi_1 \mid\mid \alpha_2 \vdash \varphi$ zu zeigen. Damit aus i) und ii) die Aussage $\alpha_1 \mid\mid \alpha_2 \vdash \varphi$ folgt, müssen in der Semantik offene Systeme betrachtet werden, die beliebig von der Umgebung beeinflusst werden können. Beim Beweis von i) ist nicht bekannt, in welcher Umgebung das Programm α_1 abläuft (hier α_2) und wie diese Umgebung gemeinsame Variablen beeinflusst. Deshalb folgt in der Semantikdefinition nach einem Programmschritt von α_1, der Variablen entsprechend der aktuellen Programmanweisung ändert, ein Umgebungsschritt, der gemeinsame Variablen noch beliebig abändern kann (siehe Diskussion über 1-fach und 2-fach gestrichene Variablen in Abschnitt 6.1.4). Das Programm garantiert so in allen Umgebungen φ_1. In ii) kann nun das Programm α_1 durch die Eigenschaft φ_1 ersetzt werden.

Dieser Ansatz kann wie folgt auf Zustände übertragen werden. Statt



Ein Konnektor im Zustand s_1 zeigt an, dass s_{1_1} nicht nur mit dem Übergang nach s_{1_2} verlassen werden kann. Wie für parallele Programme kann nun eine Semantik für Statecharts definiert werden, die zwischen Programm- bzw. Statechart-Schritten und Umgebungsschritten unterscheidet (beachte: ein Makro-Schritt ist ein Statechart-Schritt und kein Umgebungsschritt). Nach einem Statechart-Schritt kann ein Umgebungsschritt bestimmte Variablen beliebig ändern. Die Semantik von Statecharts muss dann so definiert werden, dass ein Umgebungsschritt den Zustand s_1 verlassen und inaktiv setzen kann (dies entspricht dem möglichen Übergang am Konnektor), wenn in s_1 der Zustand s_{1_1} aktiv ist. Der Umgebungsschritt simuliert so alle möglichen Übergänge, die in s_1 möglich sind. Wir haben diese Idee bisher noch nicht weiter verfolgt, es scheint aber naheliegend, die Ideen aus der Programmverifikation auf die Statechart-Verifikation zu übertragen.

ANHANG A

Software FTA

Die Software-FTA [LC91, LH83] analysiert, unter welchen Bedingungen ein Programm fehlerhafte Ergebnisse liefert. Schrittweise wird für jede Programmanweisung untersucht, welche Bedingungen vor einer Programmanweisung gelten müssen, damit das fehlerhafte Ergebnis erzeugt wird. Dazu gibt es für jede Programmanweisung ein sogenanntes Fehlerbaummuster, das die Änderungen der Programmanweisung beschreibt. So arbeitet man sich Anweisung für Anweisung vom Programmende zum Anfang des Programms vor.

Im Folgenden stellen wir exemplarisch einige Muster für die Software FTA (SFTA) vor und veranschaulichen die Anwendung der SFTA-Muster mit einem Beispiel aus [HL83]. Für das Programm

```
if a > b
  then z := f(z)
  else z := 10
while b > x
   begin
      b := b - 1;
      z := z + 10;
   end
```

wird mittels der SFTA untersucht, wie das fehlerhafte Ereignis $z > 100$ zustande kommt. Dazu werden SFTA-Muster für *while*-Schleifen, *if*-Anweisungen und für *Zuweisungen* benötigt.

In Abbildung A.1 sehen wir, wie eine *while*-Schleife mit der SFTA untersucht wird. Das unerwünschte Ereignis tritt entweder ein, wenn der Schleifenrumpf nicht betreten wird und es schon vorher vorhanden war, oder wenn der Schleifenrumpf n-mal durchlaufen wird und es dann gilt. Im ersten Fall muss die Schleifenbedingung zu false ausgewertet werden, im zweiten n-mal zu true. Die Schwierigkeit bei der Analyse besteht darin, die Anzahl der Schleifendurchläufe zu bestimmen.

Ähnlich lässt sich eine *if*-Anweisung untersuchen (siehe Abbildung A.2). Entweder tritt das Ereignis auf, wenn die Bedingung zu true ausgewertet wird und der *then*-Zweig das

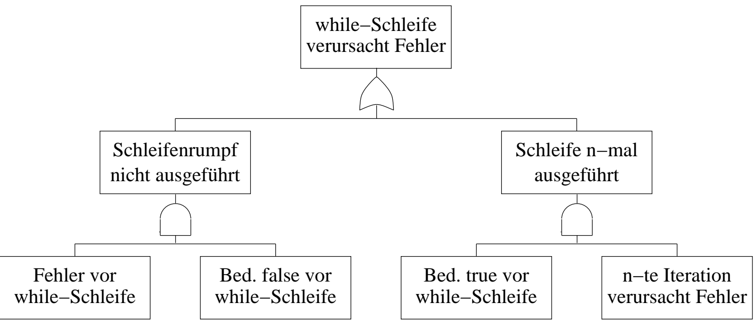


Abbildung A.1: Muster für while-Schleife

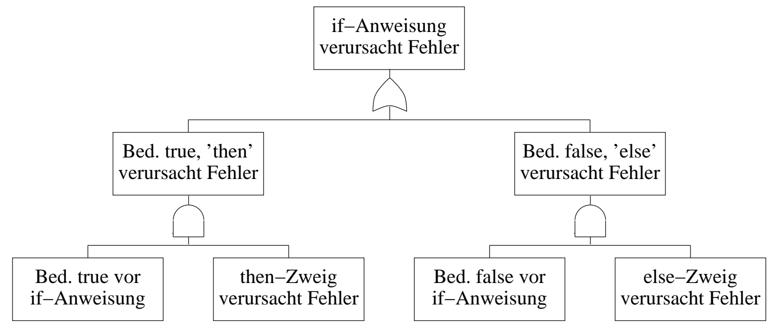


Abbildung A.2: Muster für if-Schleife

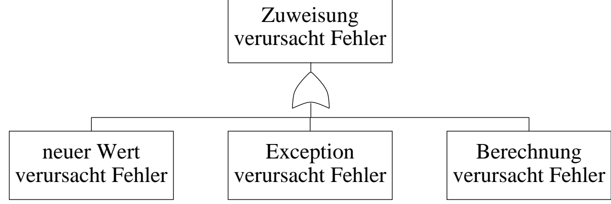


Abbildung A.3: Muster für die Zuweisung

Ereignis verursacht, oder die Bedingung wird zu false ausgewertet und der *else*-Zweig verursacht den Fehler. Beim Muster für die Zuweisung nach Abbildung A.3 wird eine Programmiersprache mit *Exceptions* behandelt. Für Programmiersprachen ohne *Exceptions* sind nur die Fälle „neuer Wert verursacht Fehler“ bzw. „Berechnung verursacht Fehler“ zu betrachten.

Mit diesen SFTA-Muster kann der Fehlerbaum für das obige Programmfragment erstellt werden. Dazu wird das Programm von der Ausgabe ($z > 100$) her, d. h. am Programmende beginnend, analysiert.

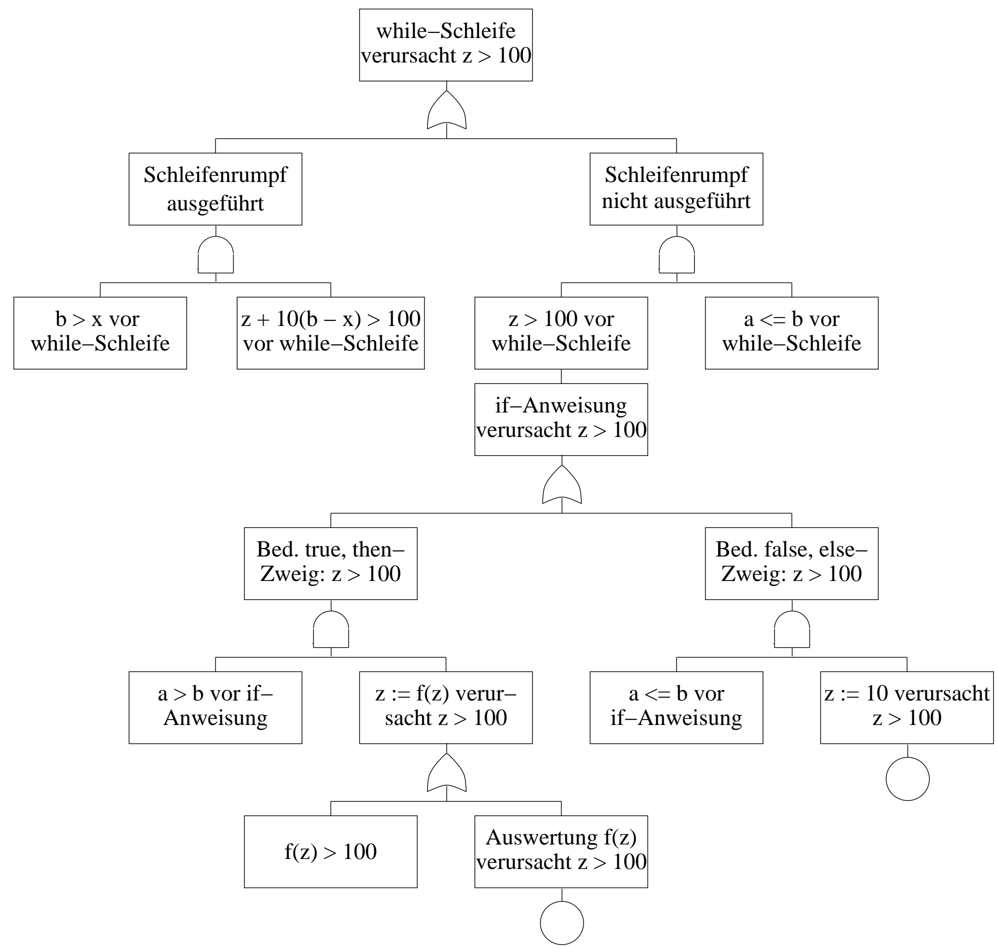


Abbildung A.4: SFTA des Beispiel-Programms

Zunächst muss untersucht werden, wie die *while*-Schleife das Ereignis $z > 100$ liefern kann. Im linken Ast wird die *while*-Schleife nie durchlaufen. Dann muss zuvor $z > 100$ gewesen sein und b kleiner-gleich x sein. Der rechte Ast untersucht, wie die Anweisungen innerhalb der *while*-Schleife das Ereignis erzeugen können. Wie bei der Programmverifikation ist dazu eine Invariante notwendig. Sei n die Anzahl der Schleifendurchläufe und z_n der Wert von z nach der n-ten Iteration. Dann muss $z_n > 100$ untersucht werden. Es muss nun

ein Ausdruck für n und z_n unter dem Ausgangswert z_0 (vor Schleifenbeginn) gefunden werden. Die zweite Anweisung in der Schleife ergibt, dass $z_n = z_0 + 10 * n$ (1) und nach n Schleifeniterationen ist $b_n = b_0 - n$. Da die Schleife aber nach n Iterationen beendet wird, muss $b_n \geq x$ gelten. Daraus ergibt sich dann $b_0 - n \geq x$ und $b_0 - x \geq n$ (2). Zusammengenommen ergeben die Gleichungen $100 < z_n \leq z_0 + 10 * (b_0 - x)$.

Betrachten wir den rechten Teilast weiter, in dem das fehlerhafte Ereignis $z > 100$ vor der Schleife gilt. Als nächstes ist die *if*-Anweisung zu untersuchen. Entweder ist $a > b$ und $z := f(z)$ verursacht das unerwünschte Ereignis, oder $a \leq b$ und die Zuweisung $z := 10$. Da keine *Exception* auftreten kann und die rechte Seite der Zuweisung nicht ausgewertet werden muss, kann nur die Änderung des Wertes z das Ereignis verursachen. Da aber $10 \not> 100$ ist, erhält man einen Widerspruch. Im anderen Fall kann die Zuweisung $z := f(z)$ den Fehler verursachen. Entweder ist das Ergebnis von $f(z)$ größer 100, oder die Auswertung von $f(z)$ verursacht einen Wert von z, der größer 100 ist. Dies kann aber nicht sein, da selbst bei einem Seiteneffekt der Wert von z mit dem Ergebnis von $f(z)$ überschrieben werden würde. Dieser Ast des Fehlerbaums kann also auch nicht das unerwünschte Ergebnis verursachen. Die Ursache muss in einem der anderen Äste auftreten, oder kann überhaupt nicht vom Programm erzeugt werden. Dann müsste das Basisereignis aus der System-FTA gestrichen werden.

Auf diese Art und Weise müssen die weiteren Äste im Fehlerbaum untersucht werden, bis alle am Funktionsanfang angekommen sind oder einen Widerspruch ergeben haben.

ANHANG B

KIV-Fallstudie: Minimale Schnittmengen

Für den Nachweis des minimalen Schnittmengen-Theorems 5.1 und des Lemmas 5.2 über die Korrektheit von Fehlerbäumen haben wir in KIV Fehlerbäume und deren formale Semantik algebraisch spezifiziert. Einen Überblick über die Spezifikation gibt der Spezifikationsgraph im Anhang C.

In der Spezifikation *faulttree* auf Seite 231 sind Fehlerbäume analog zur Definition 5.12 spezifiziert. Ein Fehlerbaum ist entweder ein Blatt, oder ein fehlerhaftes Ereignis φ, das über eines der 7 Fehlerbaumgatter, die wir in Abschnitt 5.1 besprochen haben, mit einer Menge von Unterbäumen verbunden wird.[1] Das fehlerhafte Ereignis φ wird durch die Funktion f (für Fehler) selektiert. Die Korrektheits- und Vollständigkeitsbedingungen der Fehlerbaumgatter sind in der Spezifikation *fta*, Seite 228, über einer Intervalltemporallogik spezifiziert. Im Gegensatz zur Definition der ITL in Abschnitt 5.2, die auf einem diskreten Zeitmodell basiert, wurde in KIV die ITL über einem kontinuierlichem Zeitmodell spezifiziert (siehe Spezifikation *interval* im Abschnitt C, Seite 240). Dieser Unterschied hat für den Nachweis der beiden Theoreme jedoch keine Auswirkung. Die Berechnung der minimalen Schnittmengen ist ebenfalls in der Spezifikation *fta* spezifiziert. Dazu konstruiert die Funktion *ec* nach der Struktur des Fehlerbaums eine Formel τ, die genau dann gilt, wenn alle Ereignisse einer minimalen Schnittmenge eintreten. Die disjunktive Normalform von τ entspricht den minimalen Schnittmengen. Jedes Disjunktionsglied entspricht einer minimalen Schnittmenge mit den Konjunktionsgliedern als Elemente der Schnittmenge. Das Theorem 5.1 und das Lemma 5.2 sind als Beweispflichten in der Spezifikation *fta* folgendermaßen formalisiert:

Theorem 5.1: $complete(IS, t) \vdash IS \models_i \ (\neg_{tl}\ ec(t)) \rightarrow_{tl}\ \neg_{tl} \Diamond_x\ f(t)$

Lemma 5.2: $correct(IS, t) \vdash IS \models_i \ (\Diamond_x\ ec(t)) \rightarrow_{tl}\ (\Diamond_x\ f(t))$

IS ist eine Menge von Intervallen und t ein Fehlerbaum. Erfüllen die Intervalle in IS die lokalen Vollständigkeitsbedingungen der Fehlerbaumgatter $complete(IS, t)$ aus der Spezi-

[1] In der Spezifikation *faulttree* sind zusätzlich die beiden Gatter *hdis* und *hcon* spezifiziert, die die Ursache/Wirkungs-Beziehung für das *Oder*- bzw. *Und*-Gatter nach Hansen et al. [HRS94] beschreiben. Die beiden Theoreme gelten auch für diese Gatter.

fikation *fta*, dann erfüllen ($\models_i$, aus der Spezifikation *init-valid*, Seite230) alle Intervalle in IS, wenn keines der minimalen Schnittmengen wahr wird ($\neg_{tl}\ ec(t)$), dann tritt auch das Top-Ereignis des Fehlerbaums t nicht ein ($\neg_{tl}\Diamond_x\ f(t)$). Der Sequenzenhaken $\vdash$ ist ein Operator des Sequenzenkalküls, wobei $\varphi \vdash \psi$ der Implikation $\varphi \rightarrow \psi$ entspricht (siehe Abschnitt 6.1.2 für Details zum Sequenzenkalkül). Das Theorem für die Korrektheit ist analog formuliert. Wenn die Intervalle IS die lokalen Korrektheitsbedingungen $correct(IS, t)$ erfüllen und irgendwann eine der minimalen Schnittmengen erfüllt ist ($\Diamond_x\ ec(t)$), dann tritt auch die Wirkung des Fehlerbaums t ein ($\Diamond_x\ f(t)$). Die Definition von $correct(IS, t)$ findet sich ebenfalls in der Spezifikation *fta*.

B.1 Beweis: Vollständigkeit

Wir präsentieren für das Theorem 5.1 nicht den detaillierten KIV Beweis im Sequenzenkalkül, sondern einen äquivalenten, abstrakteren Beweis. Deshalb verwenden wir auch die gewohnte Notation der ITL-Operatoren und nicht die Notation aus der algebraischen Spezifikation. Wir definieren die Vollständigkeit

$$comp(t) \wedge \neg\ ec(t) \rightarrow \neg\ \Diamond_x f(t)$$

oder äquivalent

$$comp(t) \wedge \Diamond_x f(t) \rightarrow ec(t).$$

Den Beweis führen wir mit struktureller Induktion, die

$$comp(t) \wedge \Diamond_x f(t) \rightarrow ec(t) \qquad (1)$$

als Induktionsvoraussetzung liefert. Die Blätter eines Fehlerbaums etablieren die Induktionsbasis.

Blätter zu zeigen ist

$$comp(leaf(\varphi)) \wedge \Diamond_x f(leaf(\varphi)) \rightarrow ec(leaf(\varphi))$$

Durch Einsetzen der Definitionen von ec, f und der Vollständigkeitsbedingung erhalten wir

$$\text{true} \wedge \Diamond_x \varphi \rightarrow \Diamond_x \varphi,$$

was trivial wahr ist.

synchrones *UW-Und*-Gatter zu zeigen ist

$$comp(scon(\varphi, t_1, t_2)) \wedge \Diamond_x f(scon(\varphi, t_1, t_2)) \rightarrow ec(scon(\varphi, t_1, t_2))$$

Wir setzen zuerst die Definitionen von f, ec und der Vollständigkeit des synchronen *UW-Und*-Gatters ein

$$\neg\ (\neg\ \Diamond_x(f(t_1) \wedge f(t_2))\ \boldsymbol{;}\ \Diamond_i\varphi) \wedge comp(t_1) \wedge comp(t_2) \wedge \Diamond_x\varphi$$
$$\rightarrow ec(t_1) \wedge ec(t_2)$$

und erhalten

$$\begin{aligned} & comp(t_1) \wedge comp(t_2) \wedge \Diamond_{\mathrm{x}}\varphi \\ \rightarrow\ & (ec(t_1) \wedge ec(t_2)) \vee (\neg \Diamond_{\mathrm{x}}(f(t_1) \wedge f(t_2))\ ;\ \Diamond_{\mathrm{i}}\varphi). \end{aligned}$$

In der Voraussetzung der Implikation gilt $\Diamond_{\mathrm{x}}\varphi$. Deshalb gibt es einen Zeitpunkt c, an dem auch $\Diamond_{\mathrm{i}}\varphi$ gilt. An diesem Zeitpunkt c lösen wir den **;**-Operator auf und erhalten

$$\begin{aligned} & comp(t_1) \wedge comp(t_2) \wedge \Diamond_{\mathrm{x}}\varphi \\ \rightarrow\ & (ec(t_1) \wedge ec(t_2)) \vee \neg \Diamond_{\mathrm{x}}(f(t_1) \wedge f(t_2)), \end{aligned}$$

wobei $\neg \Diamond_{\mathrm{x}}(f(t_1) \wedge f(t_2))$ vor c gelten muss. Daraus folgt

$$\begin{aligned} & \Diamond_{\mathrm{x}}(f(t_1) \wedge f(t_2)) \wedge comp(t_1) \wedge comp(t_2) \wedge \Diamond_{\mathrm{x}}\varphi \\ \rightarrow\ & ec(t_1) \wedge ec(t_2) \end{aligned}$$

mit $\Diamond_{\mathrm{x}}(f(t_1) \wedge f(t_2))$ vor c und auch

$$\begin{aligned} & \Diamond_{\mathrm{x}}f(t_1) \wedge \Diamond_{\mathrm{x}}f(t_2) \wedge comp(t_1) \wedge comp(t_2) \wedge \Diamond_{\mathrm{x}}\varphi \\ \rightarrow\ & ec(t_1) \wedge ec(t_2) \end{aligned}$$

mit $\Diamond_{\mathrm{x}}f(t_1)$ und $\Diamond_{\mathrm{x}}f(t_2)$ vor c. Mit der Induktionsvoraussetzung (1) für die Unterbäume t_1 und t_2 folgt aus der Voraussetzung der Implikation $ec(t_1)$ und $ec(t_2)$.

asynchrones *UW-Und*-Gatter zu zeigen ist

$$comp(acon(\varphi, t_1, t_2)) \wedge \Diamond_{\mathrm{x}}f(acon(\varphi, t_1, t_2)) \rightarrow ec(acon(\varphi, t_1, t_2))$$

Wir setzen zuerst die Definitionen von f, ec und der Vollständigkeit des asynchronen *UW-Und*-Gatters ein

$$\begin{aligned} & \neg (\neg \Diamond_{\mathrm{x}}f(t_1)\ ;\ \Diamond_{\mathrm{i}}\varphi) \wedge comp(t_1) \wedge \\ & \neg (\neg \Diamond_{\mathrm{x}}f(t_2)\ ;\ \Diamond_{\mathrm{i}}\varphi) \wedge comp(t_2) \wedge \Diamond_{\mathrm{x}}\varphi \\ \rightarrow\ & ec(t_1) \wedge ec(t_2) \end{aligned}$$

und erhalten

$$\begin{aligned} & comp(t_1) \wedge comp(t_2) \wedge \Diamond_{\mathrm{x}}\varphi \\ \rightarrow\ & (ec(t_1) \wedge ec(t_2)) \vee (\neg \Diamond_{\mathrm{x}}f(t_1)\ ;\ \Diamond_{\mathrm{i}}\varphi) \vee (\neg \Diamond_{\mathrm{x}}f(t_2)\ ;\ \Diamond_{\mathrm{i}}\varphi). \end{aligned}$$

Wegen $\Diamond_{\mathrm{x}}\varphi$ gibt es einen Zeitpunkt c, an dem $\Diamond_{\mathrm{i}}\varphi$ gilt und wir können bei c beide **;**-Operatoren auflösen

$$\begin{aligned} & comp(t_1) \wedge comp(t_2) \wedge \Diamond_{\mathrm{x}}\varphi \\ \rightarrow\ & (ec(t_1) \wedge ec(t_2)) \vee (\neg \Diamond_{\mathrm{x}}f(t_1)) \vee (\neg \Diamond_{\mathrm{x}}f(t_2)), \end{aligned}$$

wobei $(\neg \Diamond_{\mathrm{x}}f(t_1)) \vee (\neg \Diamond_{\mathrm{x}}f(t_2))$ vor c gelten muss. Wie im Beweis für das synchrone *UW-Und*-Gatter bringen wir $\Diamond_{\mathrm{x}}f(t_1)$ und $\Diamond_{\mathrm{x}}f(t_2)$ auf die linke Seite und schließen den Fall per Induktionsvoraussetzung (1).

***UW-Oder*-Gatter** zu zeigen ist

$$comp(dis(\varphi, t_1, t_2)) \wedge \Diamond_{\mathrm{x}} f(dis(\varphi, t_1, t_2)) \rightarrow ec(dis(\varphi, t_1, t_2))$$

Wir setzen zuerst die Definitionen von f, ec und der Vollständigkeit des asynchronen *UW-Oder*-Gatters ein.

$$\begin{aligned} &\neg\, (\neg\, \Diamond_{\mathrm{x}} f(t_1) \vee f(t_2)\ ;\ \Diamond_{\mathrm{i}}\varphi) \wedge comp(t_1) \wedge comp(t_2) \wedge \Diamond_{\mathrm{x}}\varphi \\ \rightarrow\ & ec(t_1) \vee ec(t_2) \end{aligned}$$

Der Beweisverlauf ist analog zu den Beweisen der *UW-Und*-Gatter, wobei nur ein Eintreten von $ec(t_1)$ oder $ec(t_2)$ zu zeigen ist.

***UW-Block*-Gatter** zu zeigen ist

$$comp(inh(\varphi, t, \chi)) \wedge \Diamond_{\mathrm{x}} f(inh(\varphi, t, \chi)) \rightarrow ec(inh(\varphi, t, \chi))$$

Wir setzen zuerst die Definitionen von f, ec und der Vollständigkeit des asynchronen *UW-Block*-Gatters ein.

$$\begin{aligned} &\neg\, (\neg\, \Diamond_{\mathrm{x}} f(t)\ ;\ \Diamond_{\mathrm{i}}\varphi) \wedge \neg\, (\neg\, \Diamond_{\mathrm{x}}\chi\ ;\ \Diamond_{\mathrm{i}}\varphi) \wedge comp(t) \wedge \Diamond_{\mathrm{x}}\varphi \\ \rightarrow\ & ec(t) \end{aligned}$$

Wir erhalten dieselbe Aussage wir für das asynchrone *UW-Und*-Gatter und schließen den Beweis analog.

***Z-Und*-Gatter** zu zeigen ist

$$comp(zcon(\varphi, t_1, t_2)) \wedge \Diamond_{\mathrm{x}} f(zcon(\varphi, t_1, t_2)) \rightarrow ec(zcon(\varphi, t_1, t_2))$$

Wir setzen zuerst die Definitionen von f, ec und der Vollständigkeit des *Z-Und*-Gatters ein.

$$\begin{aligned} &\boxtimes(\varphi \rightarrow f(t_1) \wedge f(t_2)) \wedge comp(t_1) \wedge comp(t_2) \wedge \Diamond_{\mathrm{x}}\varphi \\ \rightarrow\ & ec(t_1) \wedge ec(t_2) \end{aligned}$$

In jedem Sub-Intervall ($\boxtimes$) impliziert φ die Ursachen $f(t_1) \wedge f(t_2)$ und wegen der Voraussetzung $\Diamond_{\mathrm{x}}\varphi$ gibt es ein Sub-Intervall, in dem φ gilt. In diesem Sub-Intervall gilt auch $f(t_1) \wedge f(t_2)$. Für dieses Sub-Intervall erhalten wir

$$f(t_1) \wedge f(t_2) \wedge comp(t_1) \wedge comp(t_2) \rightarrow ec(t_1) \wedge ec(t_2)$$

und schließen den Beweis durch Anwenden der Induktionsvoraussetzung (1) für t_1 und t_2.

***Z-Oder*-Gatter** zu zeigen ist

$$comp(zdis(\varphi, t_1, t_2)) \wedge \diamondplus f(zdis(\varphi, t_1, t_2)) \rightarrow ec(zdis(\varphi, t_1, t_2))$$

Wir setzen zuerst die Definitionen von f, ec und der Vollständigkeit des *Z-Oder*-Gatters ein.

$$\begin{aligned} &\boxtimes(\varphi \rightarrow f(t_1) \vee f(t_2)) \wedge comp(t_1) \wedge comp(t_2) \wedge \diamondplus\varphi \\ \rightarrow\ & ec(t_1) \vee ec(t_2) \end{aligned}$$

Der weitere Beweis verläuft analog zum Fall für das *Z-Und*-Gatter, wobei nur ein Eintreten von $ec(t_1)$ oder $ec(t_2)$ zu zeigen ist.

***Z-Block*-Gatter** zu zeigen ist

$$comp(zinh(\varphi, t, \chi)) \wedge \diamondplus f(zinh(\varphi, t, \chi)) \rightarrow ec(zinh(\varphi, t, \chi))$$

Wir setzen zuerst die Definitionen von f, ec und der Vollständigkeit des *Z-Block*-Gatters ein.

$$\boxtimes(\varphi \rightarrow (f(t)) \wedge \chi) \wedge comp(t) \wedge \diamondplus\varphi \rightarrow ec(t)$$

Auch hier verläuft der Restbeweis analog zu den beiden obigen Beweisen für die Z-Gatter.

□

B.2 Beweis: Korrektheit

Wie für die Vollständigkeit, präsentieren wir für das Lemma 5.2 einen abstrakten Beweis für die Korrektheit minimaler Schnittmengen, der äquivalent dem Sequenzenkalkülbeweis in KIV ist. Formal definieren wir die Korrektheit

$$corr(t) \wedge ec(t) \rightarrow \diamondplus f(t).$$

Der Beweis erfolgt wieder mit struktureller Induktion. Diese liefert

$$corr(t) \wedge ec(t) \rightarrow \diamondplus f(t) \tag{2}$$

als Induktionsvoraussetzung. Die Blätter eines Fehlerbaums bilden die Induktionsbasis für die strukturelle Induktion.

Blätter zu zeigen ist

$$corr(leaf(\varphi)) \wedge ec(leaf(\varphi)) \rightarrow \diamondplus f(leaf(\varphi))$$

Durch Einsetzen der Definitionen f, ec und der Korrektheit eines Gatters erhalten wir

$$\text{true} \wedge \diamondplus\varphi \rightarrow \diamondplus\varphi$$

und der Basisfall ist geschlossen.

asynchrones *UW-Und*-Gatter zu zeigen ist

$$corr(acon(\varphi, t_1, t_2)) \land ec(acon(\varphi, t_1, t_2)) \rightarrow \diamondtimes f(acon(\varphi, t_1, t_2)$$

Nach Einsetzen der Definitionen für ec, f und der Korrektheitsbedingung des asynchronen *UW-Und*-Gatter erhalten wir

$$\begin{aligned}&(\diamondtimes f(t_1) \land \diamondtimes f(t_2) \rightarrow \diamondtimes\varphi)\\ &\land corr(t_1) \land corr(t_2) \land ec(t_1) \land ec(t_2) \rightarrow \diamondtimes\varphi.\end{aligned}$$

Mit der Induktionsvoraussetzung (2) können wir $\diamondtimes f(t_1)$ und $\diamondtimes f(t_2)$ schließen. Daraus folgt in der Voraussetzung der Implikation $\diamondtimes\varphi$ und dieser Fall ist geschlossen.

synchrones *UW-Und*-Gatter ↯

Das synchrone *UW-Und*-Gatter erfüllt nicht das Korrektheitskriterium, denn wir müssten analog zum asynchronen *UW-Und*-Gatter

$$corr(scon(\varphi, t_1, t_2)) \land ec(scon(\varphi, t_1, t_2)) \rightarrow \diamondtimes f(scon(\varphi, t_1, t_2)$$

zeigen. Nach Einsetzen der Definitionen für ec, f und der Korrektheitsbedingung des synchronen *UW-Und*-Gatter erhalten wir

$$\begin{aligned}&((\diamondtimes f(t_1) \land f(t_2)) \rightarrow \diamondtimes\varphi)\\ &\land corr(t_1) \land corr(t_2) \land ec(t_1) \land ec(t_2) \rightarrow \diamondtimes\varphi\end{aligned}$$

und müssen in der Voraussetzung der Implikation

$$(\diamondtimes f(t_1) \land f(t_2)) \rightarrow \diamondtimes\varphi$$

herleiten. Die Induktionsvoraussetzung (2) liefert aber nur $\diamondtimes f(t_1)$ und $\diamondtimes f(t_2)$, d. h. es kann nicht gezeigt werden, dass die beiden Ursachen zusammen in irgendeinem Sub-Intervall auftreten.

***UW-Block*-Gatter** ↯

Für ein *Block*-Gatter sind keine Korrektheitsbedingungen definiert.

***Z-Und*-Gatter** ↯

Analog zum synchronen *UW-Und*-Gatter können wir für das *Z-Und*-Gatter nicht zeigen, dass die beiden Ursachen zusammen auftreten. Wir müssten

$$corr(zcon(\varphi, t_1, t_2)) \land ec(zcon(\varphi, t_1, t_2)) \rightarrow \diamondtimes f(zcon(\varphi, t_1, t_2)$$

zeigen. Nach Einsetzen der Definitionen für ec, f und der Korrektheitsbedingung des *Z-Und*-Gatter erhalten wir

$$\begin{aligned}&(\boxtimes(f(t_1) \land f(t_2)) \rightarrow \varphi) \land\\ &(corr(t_1) \land corr(t_2) \land ec(t_1) \land ec(t_2)) \rightarrow \diamondtimes\varphi\end{aligned}$$

und müssten

$$(\boxtimes(f(t_1) \wedge f(t_2)) \rightarrow \varphi)$$

herleiten. Wir können mit der Induktionsvoraussetzung (2) zwar sowohl $f(t_1)$ als auch $f(t_2)$ herleiten, aber nicht, dass beide im gleichen Sub-Intervall gelten. Deshalb können wir nicht zeigen, dass es ein Sub-Intervall gibt, in dem φ gilt.

***Z-Oder*-Gatter** zu zeigen ist

$$corr(zdis(\varphi, t_1, t_2)) \wedge ec(zdis(\varphi, t_1, t_2)) \rightarrow \diamondtimes f(zdis(\varphi, t_1, t_2)$$

Nach Einsetzen der Definitionen für ec, f und der Korrektheitsbedingung des *Z-Oder*-Gatter erhalten wir

$$\begin{gathered}(\boxtimes f(t_1) \vee f(t_2) \rightarrow \varphi) \wedge \\ (corr(t_1) \wedge corr(t_2) \vee ec(t_1) \wedge ec(t_2)) \rightarrow \diamondtimes \varphi.\end{gathered}$$

Mit der Induktionsvoraussetzung (2) können wir $\diamondtimes f(t_1)$ oder $\diamondtimes f(t_2)$ schließen. In jedem Sub-Intervall ($\boxtimes$) gilt, wenn $f(t_1)$ oder $f(t_2)$ gilt, dann auch φ. Daraus folgt in der Voraussetzung der Implikation, dass in einem Sub-Intervall φ gilt. Deshalb gilt $\diamondtimes\varphi$ und dieser Fall ist geschlossen.

***Z-Block*-Gatter** ↯

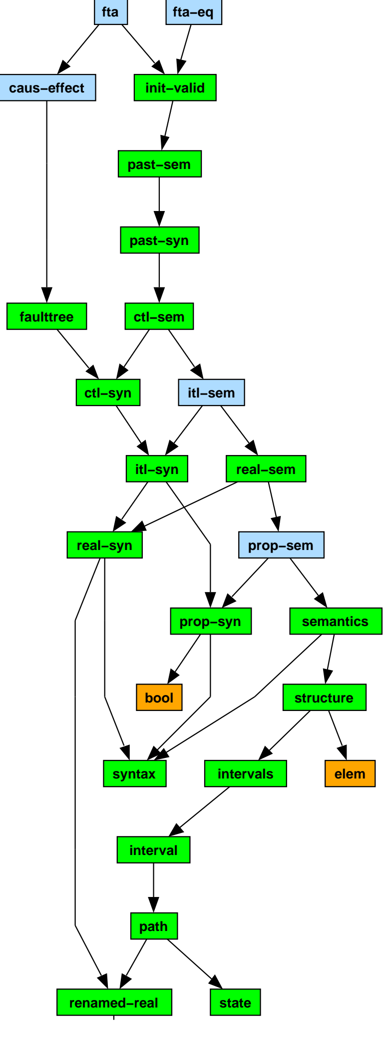
siehe *UW-Block*-Gatter

□

ANHANG C

Algebraische Spezifikation von Fehlerbäumen

Entwicklungs-Graph



Spezifikation: *fta*

fta =
enrich init-valid, caus-effect **with**
functions
ec : nft → tl ;
predicates
correct : (interval → bool) × nft;
complete : (interval → bool) × nft;

axioms

correct(is, leaf(φ));

correct(is, scon(φ, t_1, t_2))
$\leftrightarrow$ is $\models_i$ f(t_1) $\wedge_{tl}$ f(t_2) uwcorr φ $\wedge$ correct(is, t_1) $\wedge$ correct(is, t_2);

correct(is, acon(φ, t_1, t_2))
$\leftrightarrow$ is $\models_i$ $\diamond_x$ f(t_1) $\wedge_{tl}$ $\diamond_x$ f(t_2) $\rightarrow_{tl}$ $\diamond_x$ φ $\wedge$ correct(is, t_1) $\wedge$ correct(is, t_2);

correct(is, dis(φ, t_1, t_2))
$\leftrightarrow$ is $\models_i$ f(t_1) $\vee_{tl}$ f(t_2) uwcorr φ $\wedge$ correct(is, t_1) $\wedge$ correct(is, t_2);

correct(is, zcon(φ, t_1, t_2))
$\leftrightarrow$ is $\models_i$ f(t_1) $\wedge_{tl}$ f(t_2) zcorr φ $\wedge$ correct(is, t_1) $\wedge$ correct(is, t_2);

correct(is, zdis(φ, t_1, t_2))
$\leftrightarrow$ is $\models_i$ f(t_1) $\vee_{tl}$ f(t_2) zcorr φ $\wedge$ correct(is, t_1) $\wedge$ correct(is, t_2);

correct(is, hdis(φ, t_1, t_2))
$\leftrightarrow$ is $\models_i$ $\diamond_x$(f(t_1) $\vee_{tl}$ f(t_2)) $\rightarrow_{tl}$ $\diamond_x$ φ $\wedge$ correct(is, t_1) $\wedge$ correct(is, t_2);

correct(is, hcon(φ, t_1, t_2))
$\leftrightarrow$ is $\models_i$ $\diamond_x$(f(t_1) $\wedge_{tl}$ f(t_2)) $\rightarrow_{tl}$ $\diamond_x$ φ $\wedge$ correct(is, t_1) $\wedge$ correct(is, t_2);

$\neg$ correct(is, scon(φ, t_1, t_2));

$\neg$ correct(is, zcon(φ, t_1, t_2));

$\neg$ correct(is, hcon(φ, t_1, t_2));

$\neg$ correct(is, inh(φ, t, χ));

$\neg$ correct(is, zinh(φ, t, χ));

complete(is, leaf(φ));

complete(is, scon(φ, t_1, t_2))
$\leftrightarrow$ is $\models_i$ f(t_1) $\wedge_{tl}$ f(t_2) uwcomp φ $\wedge$ complete(is, t_1) $\wedge$ complete(is, t_2);

complete(is, acon(φ, t_1, t_2))
$\leftrightarrow$ is $\models_i$ f(t_1) uwcomp φ $\wedge$ is $\models_i$ f(t_2) uwcomp φ $\wedge$ complete(is, t_1) $\wedge$ complete(is, t_2);

complete(is, dis(φ, t_1, t_2))
$\leftrightarrow$ is $\models_i$ f(t_1) $\vee_{tl}$ f(t_2) uwcomp φ $\wedge$ complete(is, t_1) $\wedge$ complete(is, t_2);
complete(is, inh(φ, t_1, t_2))
$\leftrightarrow$ is $\models_i$ f(t) uwcomp φ $\wedge$ is $\models_i$ f(χ) uwcomp φ $\wedge$ complete(is, t) ;
complete(is, zcon(φ, t_1, t_2))
$\leftrightarrow$ is $\models_i$ f(t_1) $\wedge_{tl}$ f(t_2) zcomp φ $\wedge$ complete(is, t_1) $\wedge$ complete(is, t_2);
complete(is, zdis(φ, t_1, t_2))
$\leftrightarrow$ is $\models_i$ f(t_1) $\vee_{tl}$ f(t_2) zcomp φ $\wedge$ complete(is, t_1) $\wedge$ complete(is, t_2);
complete(is, zinh(φ, t, χ)) $\leftrightarrow$ is $\models_i$ f(t) $\wedge_{tl}$ f(χ) zcomp φ $\wedge$ complete(is, t);
complete(is, hdis(φ, t_1, t_2))
$\leftrightarrow$ is $\models_i$ $\diamond_x$ φ $\rightarrow_{tl}$ $\diamond_x$(f(t_1) $\vee_{tl}$ f(t_2)) $\wedge$ complete(is, t_1) $\wedge$ complete(is, t_2);
complete(is, hcon(φ, t_1, t_2))
$\leftrightarrow$ is $\models_i$ $\diamond_x$ φ $\rightarrow_{tl}$ $\diamond_x$(f(t_1) $\wedge_{tl}$ f(t_2)) $\wedge$ complete(is, t_1) $\wedge$ complete(is, t_2);
ec(leaf(φ)) = $\diamond_x$ φ;
ec(scon(φ, t_1, t_2)) = ec(t_1) $\wedge_{tl}$ ec(t_2);
ec(acon(φ, t_1, t_2)) = ec(t_1) $\wedge_{tl}$ ec(t_2);
ec(dis(φ, t_1, t_2)) = ec(t_1) $\vee_{tl}$ ec(t_2);
ec(inh(φ, t, χ)) = ec(t);
ec(zcon(φ, t_1, t_2)) = ec(t_1) $\wedge_{tl}$ ec(t_2);
ec(zdis(φ, t_1, t_2)) = ec(t_1) $\vee_{tl}$ ec(t_2);
ec(zinh(φ, t, χ)) = ec(t);
ec(hdis(φ, t_1, t_2)) = ec(t_1) $\vee_{tl}$ ec(t_2);
ec(hcon(φ, t_1, t_2)) = ec(t_1) $\wedge_{tl}$ ec(t_2);

end enrich

Spezifikation: *fta-eq*

fta-eq =
enrich init-valid **with**
predicates
point : tl;

axioms

point : point(φ) $\leftrightarrow$ ($\forall$ m, m_1. m[m.now] = m_1[m_1.now] $\wedge [\![\varphi]\!]_{bool}$m $\rightarrow [\![\varphi]\!]_{bool} m_1$);

end enrich

Spezifikation: ***cause-effect***

cause-effect =
enrich faulttree-new **with**
functions

. zcorr .	:	tl × tl	→	tl	**prio** 9;
. zcomp .	:	tl × tl	→	tl	**prio** 9;
. uwcorr .	:	tl × tl	→	tl	**prio** 9;
. uwcomp .	:	tl × tl	→	tl	**prio** 9;

axioms

zcorr : φ zcorr $\psi = \Box_x(\varphi \rightarrow_{tl} \psi)$;

zcomp : φ zcomp $\psi = \Box_x(\psi \rightarrow_{tl} \varphi)$;

uwcorr : φ uwcorr $\psi = \Diamond_x \varphi \rightarrow_{tl} \Diamond_x \psi$;

uwcomp : φ uwcomp $\psi = \neg_{tl}(\Box_x \neg_{tl} \varphi \ \hat{}\ \Diamond_i \psi)$;

end enrich

Spezifikation: ***init-valid***

init-valid =
enrich past-sem **with**
predicates

. $\models$-∞ .	:	(interval → bool) × tl **prio** 6;
. $\models_i$.	:	(interval → bool) × tl **prio** 6;
. $\models$.	:	(interval → bool) × tl **prio** 6;

axioms

init-valid : is $\models_i \varphi \leftrightarrow (\forall$ m. m.is = is $\wedge$ m.now = $0_r \rightarrow [\![\varphi]\!]_{bool}$m);

inf-valid :
is $\models$-∞ φ
$\leftrightarrow \forall$ i, x. (is)(i) $\wedge$ x $\in$ i $\wedge$ (i.low $\neq$ -∞ $\rightarrow$ x = i.low $\infty\!\rightarrow_r$) $\rightarrow [\![\varphi]\!]_{bool}$mkstruct(is, i, x);

valid : is $\models \varphi \leftrightarrow (\forall$ i, x. (is)(i) $\wedge$ x $\in$ i $\rightarrow [\![\varphi]\!]_{bool}$mkstruct(is, i, x));

end enrich

Spezifikation: ***faulttree***

faulttree =
data specification
using ctl-syn
nft = leaf (f : tl ;) **with** leafp
| scon (f : tl ; ft_1 : nft ; ft_2 : nft ;)
| acon (f : tl ; ft_1 : nft ; ft_2 : nft ;)
| dis (f : tl ; ft_1 : nft ; ft_2 : nft ;)
| zcon (f : tl ; ft_1 : nft ; ft_2 : nft ;)
| zdis (f : tl ; ft_1 : nft ; ft_2 : nft ;)
| inh (f : tl ; ft : nft ; χ : tl ;)
| zinh (f : tl ; ft : nft ; χ : tl ;)
| hdis (f : tl ; ft_1 : nft ; ft_2 : nft ;)
| hcon (f : tl ; ft_1 : nft ; ft_2 : nft ;)
;
variables t, t_0, t_1, t_2: nft;
end data specification

Spezifikation: ***past-sem***

past-sem =
enrich past-syn **with**

axioms

pallpath :
$[\![\forall' \varphi]\!]_{bool}$m
$\leftrightarrow \forall$ i, y. (m.is)(i) $\wedge$ i.low $\leq$ y $\to\infty_r$ $\wedge$ y $\to\infty_r$ $\leq$ i.high $\wedge$
i[y] = m.interval[m.now]
$\to [\![\varphi]\!]_{bool}$mkstruct(m.is, i, y);

since :
$[\![\varphi$ stl $\psi]\!]_{bool}$m
$\leftrightarrow \exists$ x. m.low $\leq$ x $\to\infty_r$
$\wedge$ x $\leq$ m.now
$\wedge [\![\psi]\!]_{bool}$m $|_{now}$ x
$\wedge$ ($\forall$ y. x < y $\wedge$ y $\leq$ m.now $\to [\![\varphi]\!]_{bool}$m $|_{now}$ y);

end enrich

Spezifikation: ***past-syn***

past-syn =
enrich ctl-sem **with**
functions

. stl .	:	tl × tl	→	tl	**prio** 9;
$\Diamond_p$.	:	tl	→	tl	;
$\Box_p$.	:	tl	→	tl	;
∀' .	:	tl	→	tl	;
∃' .	:	tl	→	tl	;

axioms

peventually : $\Diamond_p\ \varphi$ = true_tl stl φ;

palways : $\Box_p\ \varphi = \neg_{tl}\ \Diamond_p\ \neg_{tl}\ \varphi$;

pexpath : $\exists'\ \varphi = \neg_{tl}\ \forall'\ \neg_{tl}\ \varphi$;

end enrich

Spezifikation: ***ctl-sem***

ctl-sem =
enrich itl-sem, ctl-syn **with**

axioms

strong-until :
$[\![\varphi\ u_{tl}\ \psi]\!]_{bool}\mathrm{m}$
$\leftrightarrow \exists\ \mathrm{x}.\quad \mathrm{m.now} \leq \mathrm{x}$
$\wedge\ \mathrm{x} \rightarrow\infty_r\ \leq \mathrm{m.high}$
$\wedge[\![\psi]\!]_{bool}\mathrm{m}\ |_{now}\ \mathrm{x}$
$\wedge\ (\forall\ \mathrm{y}.\ \mathrm{m.now} \leq \mathrm{y} \wedge \mathrm{y} < \mathrm{x} \rightarrow [\![\varphi]\!]_{bool}\mathrm{m}\ |_{now}\ \mathrm{y})$;

allpath :
$[\![\forall_{tl}\ \varphi]\!]_{bool}\mathrm{m}$
$\leftrightarrow \forall\ \mathrm{i}.\quad (\mathrm{m.is})(\mathrm{i})$
$\wedge\ \mathrm{i.low} = \mathrm{m.low}$
$\wedge\ \mathrm{m.now} \rightarrow\infty_r\ \leq \mathrm{i.high}$
$\wedge\ (\forall\ \mathrm{y}.\ \mathrm{m.low} \leq \mathrm{y} \rightarrow\infty_r \wedge \mathrm{y} \leq \mathrm{m.now} \rightarrow \mathrm{m.path[y]} = \mathrm{i.path[y]})$
$\rightarrow[\![\varphi]\!]_{bool}\mathrm{m}\ |_i\ \mathrm{i}$;

xeventually :
$[\![\Diamond_x\ \varphi]\!]_{bool}$m
$\leftrightarrow \exists$ x, y∞. m.now $\rightarrow\infty_r \leq$ x $\rightarrow\infty_r \wedge$ x $\rightarrow\infty_r \leq$ y∞ $\wedge$ y∞ $\leq$ m.high $\wedge[\![\varphi]\!]_{bool}$m $|_{high}$ y∞ $|_{now}$ x;

ieventually :
$[\![\Diamond_i\ \varphi]\!]_{bool}$m $\leftrightarrow$ ($\exists$ y∞. m.now $\rightarrow\infty_r \leq$ y∞ $\wedge$ y∞ $\leq$ m.high $\wedge[\![\varphi]\!]_{bool}$m $|_{high}$ y∞);

end enrich

Spezifikation: *ctl-syn*

ctl-syn =
enrich itl-syn **with**
functions

$\Box_{tl}$.	:	tl	$\rightarrow$	tl	;
$\Diamond_{tl}$.	:	tl	$\rightarrow$	tl	;
$\Box_x$.	:	tl	$\rightarrow$	tl	;
$\Diamond_x$.	:	tl	$\rightarrow$	tl	;
$\Box_i$.	:	tl	$\rightarrow$	tl	;
$\Diamond_i$.	:	tl	$\rightarrow$	tl	;
. u_{tl} .	:	tl $\times$ tl	$\rightarrow$	tl	**prio** 9;
. w_{tl} .	:	tl $\times$ tl	$\rightarrow$	tl	**prio** 9;
. b_{tl} .	:	tl $\times$ tl	$\rightarrow$	tl	**prio** 9;
. wb_{tl} .	:	tl $\times$ tl	$\rightarrow$	tl	**prio** 9;
$\forall_{tl}$.	:	tl	$\rightarrow$	tl	;
$\exists_{tl}$.	:	tl	$\rightarrow$	tl	;

axioms

eventually : $\Diamond_{tl}\ \varphi =$ true_tl $u_{tl}\ \varphi$;

always : $\Box_{tl}\ \varphi = \neg_{tl}\ \Diamond_{tl}\ \neg_{tl}\ \varphi$;

weak-until : $\varphi\ w_{tl}\ \psi = (\varphi\ u_{tl}\ \psi) \vee_{tl} \Box_{tl}\ \varphi$;

expath : $\exists_{tl}\ \varphi = \neg_{tl}\ \forall_{tl}\ \neg_{tl}\ \varphi$;

xalways : $\Box_x\ \varphi = \neg_{tl}\ \Diamond_x\ \neg_{tl}\ \varphi$;

ialways : $\Box_i\ \varphi = \neg_{tl}\ \Diamond_i\ \neg_{tl}\ \varphi$;

before : $\varphi\ b_{tl}\ \psi = \neg_{tl}(\neg_{tl}\ \varphi\ u_{tl}\ \psi)$;

weak-before : $\varphi\ wb_{tl}\ \psi = (\varphi\ b_{tl}\ \psi) \vee_{tl} \Box_{tl}\ \neg_{tl}\ \psi$;

end enrich

Spezifikation: *itl-sem*

itl-sem =
enrich real-sem, itl-syn **with**

axioms

length : $[\![\text{length}]\!]_{real}$m = m.high $-_i$ m.now $\rightarrow\infty_r$;

chop :
$[\![\varphi \,\hat{}\, \psi]\!]_{bool}$m
$\leftrightarrow \exists$ y∞. m.now $\rightarrow\infty_r \leq$ y∞
$\wedge$ y$\infty \leq$ m.high
$\wedge[\![\varphi]\!]_{bool}$m $|_{high}$ y∞
$\wedge$ (y$\infty \neq \infty \rightarrow [\![\psi]\!]_{bool}$m $|_{now}$ y∞ $\infty\rightarrow_r$);

end enrich

Spezifikation: *itl-syn*

itl-syn =
enrich prop-syn, real-syn **with**

constants

length : tl;
more : tl;
inf : tl;
finite : tl;
last : tl;

functions

. ˆ .	:	tl × tl	→	tl	**prio** 12;
. $*_{tl}$	:	tl	→	tl	;
finally .	:	tl	→	tl	;

axioms

more : more = const(0_r $\rightarrow\infty_r$) $<_r$ length;

inf : inf = true_tl ˆ false_tl;

finite : finite = $\neg_{tl}$ inf;

last : last = $\neg_{tl}$ more;

finally : finally φ = true_tl ˆ (last $\wedge_{tl}$ φ);

end enrich

Spezifikation: *real-sem*

real-sem =
enrich prop-sem, real-syn **with**
functions

$. \rightarrow_{elem}$	:	real∞	$\rightarrow$	elem	;
$. \rightarrow_{real}$	:	elem	$\rightarrow$	real∞	;
$[\![\; . \;]\!]_{real} \; .$	:	tl $\times$ structure	$\rightarrow$	real∞	**prio** 1 **left**;

axioms

inverse : $x\infty \rightarrow_{elem} \rightarrow_{real} = x\infty$;

sem : $[\![\varphi]\!]_{real}\mathrm{m} = [\![\varphi]\!]_{tl}\mathrm{m} \rightarrow_{real}$;

const : $[\![\mathrm{const}(x\infty)]\!]_{real}\mathrm{m} = x\infty$;

neg : $[\![\sim_r \varphi]\!]_{real}\mathrm{m} = \sim [\![\varphi]\!]_{real}\mathrm{m}$;

add : $[\![\varphi_1 +_r \varphi_2]\!]_{real}\mathrm{m} = [\![\varphi_1]\!]_{real}\mathrm{m} + [\![\varphi_2]\!]_{real}\mathrm{m}$;

sub : $[\![\varphi_1 -_r \varphi_2]\!]_{real}\mathrm{m} = [\![\varphi_1]\!]_{real}\mathrm{m} -_i [\![\varphi_2]\!]_{real}\mathrm{m}$;

mult : $[\![\varphi_1 *_r \varphi_2]\!]_{real}\mathrm{m} = [\![\varphi_1]\!]_{real}\mathrm{m} *_i [\![\varphi_2]\!]_{real}\mathrm{m}$;

less : $[\![\varphi_1 <_r \varphi_2]\!]_{bool}\mathrm{m} \leftrightarrow [\![\varphi_1]\!]_{real}\mathrm{m} < [\![\varphi_2]\!]_{real}\mathrm{m}$;

lesseq : $[\![\varphi_1 \leq_r \varphi_2]\!]_{bool}\mathrm{m} \leftrightarrow [\![\varphi_1]\!]_{real}\mathrm{m} \leq [\![\varphi_2]\!]_{real}\mathrm{m}$;

end enrich

Spezifikation: *prop-sem*

prop-sem =
enrich semantics, prop-syn **with**
functions

$. \rightarrow_{elem}$	:	bool	$\rightarrow$	elem	;
$\neg_e \; .$	:	elem	$\rightarrow$	elem	;
$. \wedge_e \; .$	:	elem $\times$ elem	$\rightarrow$	elem	**prio** 9;

predicates

$. \rightarrow_{bool}$: elem;
$[\![\; . \;]\!]_{bool} \; .$: tl $\times$ structure **prio** 1 **left**;

variables bv, bv_1, bv_2, bv_3: bool;

axioms

inverse : bv $\rightarrow_{elem} \rightarrow_{bool} \leftrightarrow$ bv;

sem : $[\![\varphi]\!]_{bool}$m $\leftrightarrow [\![\varphi]\!]_{tl}$m $\rightarrow_{bool}$;

not-elem : $\neg_e$ bv $\rightarrow_{elem}$ = (not bv)$\rightarrow_{elem}$;

and-elem : $bv_1 \rightarrow_{elem} \wedge_e bv_2 \rightarrow_{elem}$ = (bv_1 and bv_2)$\rightarrow_{elem}$;

const : $[\![c_{bool}$ bv$]\!]_{tl}$m = bv $\rightarrow_{elem}$;

not : $[\![\neg_{tl}\ \varphi]\!]_{tl}$m = $\neg_e\ [\![\varphi]\!]_{tl}$m;

and : $[\![\varphi \wedge_{tl} \psi]\!]_{tl}$m = $[\![\varphi]\!]_{tl}$m $\wedge_e [\![\psi]\!]_{tl}$m;

end enrich

Spezifikation: ***real-syn***

real-syn =
enrich syntax, renamed-real **with**
functions

const	:	real∞	$\rightarrow$	tl	;
$\tilde{}_r$.	:	tl	$\rightarrow$	tl	;
. $+_r$.	:	tl × tl	$\rightarrow$	tl	**prio** 14;
. $-_r$.	:	tl × tl	$\rightarrow$	tl	**prio** 14 **left**;
. $*_r$.	:	tl × tl	$\rightarrow$	tl	**prio** 15;
. $<_r$.	:	tl × tl	$\rightarrow$	tl	**prio** 13;
. $\leq_r$.	:	tl × tl	$\rightarrow$	tl	**prio** 13;

end enrich

Spezifikation: ***prop-syn***

prop-syn =
enrich bool, syntax **with**
constants
true_tl : tl;
false_tl : tl;

functions

c_{bool} .	:	bool	$\rightarrow$	tl	;
$\neg_{tl}$.	:	tl	$\rightarrow$	tl	;
. $\wedge_{tl}$.	:	tl $\times$ tl	$\rightarrow$	tl	**prio** 11;
. $\vee_{tl}$.	:	tl $\times$ tl	$\rightarrow$	tl	**prio** 10;
. $\rightarrow_{tl}$.	:	tl $\times$ tl	$\rightarrow$	tl	**prio** 9;
. $\leftrightarrow_{tl}$.	:	tl $\times$ tl	$\rightarrow$	tl	**prio** 8;

variables φ, φ_1, φ_2, φ_3, ψ, ψ_1, ψ_2, ψ_3, χ, χ_1, χ_2, χ_3: tl;

axioms

true : true_tl = c_{bool} true;

false : false_tl = $\neg_{tl}$ true_tl;

or : $\varphi \vee_{tl} \psi = \neg_{tl}(\neg_{tl} \varphi \wedge_{tl} \neg_{tl} \psi)$;

imp : $\varphi \rightarrow_{tl} \psi = \neg_{tl} \varphi \vee_{tl} \psi$;

equiv : $\varphi \leftrightarrow_{tl} \psi = (\varphi \rightarrow_{tl} \psi) \wedge_{tl} (\psi \rightarrow_{tl} \varphi)$;

end enrich

Spezifikation: *semantics*

semantics =
enrich syntax, structure **with**
functions $[\![\, . \,]\!]_{tl}$. : tl $\times$ structure $\rightarrow$ elem **prio** 1 **left**;

end enrich

Spezifikation: *structure*

structure =
enrich intervals, elem **with**
sorts structure;
functions

mkstruct	:	(interval → bool) × interval × real		
			→	structure ;
. .is	:	structure	→	interval → bool ;
. .interval	:	structure	→	interval ;
. .now	:	structure	→	real ;
. .path	:	structure	→	path ;
. .low	:	structure	→	real∞ ;
. .high	:	structure	→	real∞ ;
. [.]	:	structure × real	→	state **prio** 2;
$. \|_i .$	:	structure × interval	→	structure **prio** 9 **left**;
$. \|_{now} .$	:	structure × real	→	structure **prio** 9 **left**;
$. \|_{path} .$	:	structure × path	→	structure **prio** 9 **left**;
$. \|_{low} .$	:	structure × real∞	→	structure **prio** 9 **left**;
$. \|_{high} .$	:	structure × real∞	→	structure **prio** 9 **left**;

predicates

cons : structure;

cons-params : (interval → bool) × interval × real;

variables m, m_1, m_2, m_3: structure;

axioms

structure **generated by** mkstruct;

cons-def : cons(m) ↔ cons(m.interval) ∧ m.interval $\subseteq_i$ m.is ∧
m.now ∈ m.interval;

cons : cons(m);

cons-params : cons-params(is, i, y) ↔ cons(i) ∧ i $\subseteq_i$ is ∧ y ∈ i;

intervals : cons-params(is, i, y) → mkstruct(is, i, y).is = is;

interval : cons-params(is, i, y) → mkstruct(is, i, y).interval = i;

now : cons-params(is, i, y) → mkstruct(is, i, y).now = y;

path : m.path = m.interval.path;

low : m.low = m.interval.low;

high : m.high = m.interval.high;

state : m[y] = m.interval[y];

modint : i $\subseteq_i$ m.is ∧ m.now ∈ i → m $|_i$ i = mkstruct(m.is, i, m.now);

modint-is : (m $|_i$ i).is = m.is;

modint-now : (m $|_i$ i).now = m.now;

modnow-1 : y ∈ m.interval → m $|_{now}$ y = mkstruct(m.is, m.interval, y);

modnow-2 : y $\rightarrow\infty_r$ < m.low → m $|_{now}$ y = mkstruct(m.is, m.interval, m.low $\infty\!\!\rightarrow_r$);

modnow-3 : m.high < y $\rightarrow\infty_r$ → m $|_{now}$ y = mkstruct(m.is, m.interval, m.high $\infty\rightarrow_r$);

modpath : m $|_{path}$ p = m $|_i$ (m.interval $|_{path}$ p);

modlow-1 : m.low ≤ x∞ ∧ x∞ ≤ m.now $\rightarrow\infty_r$ → m $|_{low}$ x∞ = m $|_i$ (m.interval $|_{low}$ x∞);

modlow-2 : m.now $\rightarrow\infty_r$ < x∞ → m $|_{low}$ x∞ = m $|_i$ (m.interval $|_{low}$ m.now $\rightarrow\infty_r$);

modlow-3 : x∞ < m.low → m $|_{low}$ x∞ = m $|_i$ (m.interval $|_{low}$ m.low);

modhigh-1 : m.now $\rightarrow\infty_r$ ≤ z∞ ∧ z∞ ≤ m.high → m $|_{high}$ z∞ = m $|_i$ (m.interval $|_{high}$ z∞);

modhigh-2 : z∞ < m.now $\rightarrow\infty_r$ → m $|_{high}$ z∞ = m $|_i$ (m.interval $|_{high}$ m.now $\rightarrow\infty_r$);

modhigh-3 : m.high < z∞ → m $|_{high}$ z∞ = m $|_i$ (m.interval $|_{high}$ m.high);

ext : $m_1 = m_2 \leftrightarrow m_1$.is = m_2.is ∧ m_1.interval = m_2.interval ∧ m_1.now = m_2.now;

end enrich

Spezifikation: ***syntax***

syntax =
specification
sorts tl;
variables φ, φ_1, φ_2, φ_3: tl;
end specification

Spezifikation: ***intervals***

intervals =
enrich interval **with**
predicates

. ∈ .	:	interval × (interval → bool);
. $\subseteq_i$.	:	interval × (interval → bool);
subint-closed	:	(interval → bool);

variables is, is_1, is_2, is_3: interval → bool;

axioms

elem : $i \in is \leftrightarrow (is)(i)$;

subint : subint-closed(is) $\leftrightarrow (\forall\ i_1, i_2.\ i_1 \in is \wedge i_2 \subseteq i_1 \rightarrow i_2 \in is)$;

subin : $i \subseteq_i is \leftrightarrow (\exists\ i_1.\ i \subseteq i_1 \wedge i_1 \in is)$;

end enrich

Spezifikation: *interval*

interval =
enrich path **with**
sorts interval;
functions

mkint	:	path × real∞ × real∞	→	interval	;
. .path	:	interval	→	path	;
. .low	:	interval	→	real∞	;
. .high	:	interval	→	real∞	;
. [.]	:	interval × real	→	state	**prio** 2;
$. \|_{path}\ .$	:	interval × path	→	interval	**prio** 9 **left**;
$. \|_{low}\ .$	:	interval × real∞	→	interval	**prio** 9 **left**;
$. \|_{high}\ .$	:	interval × real∞	→	interval	**prio** 9 **left**;

predicates

. ∈ .	:	real × interval;
. ⊆ .	:	interval × interval;
cons	:	interval;
cons-params	:	path × real∞ × real∞;

variables i, i_1, i_2, i_3: interval;

axioms

interval **generated by** mkint;
elem : $y \in i \leftrightarrow i.low \leq y \rightarrow\infty_r \wedge y \rightarrow\infty_r \leq i.high$;

cons-def : cons(i) $\leftrightarrow$ i.path $= \geq_{prune}(\leq_{prune}(i.path, i.low), i.high) \wedge (\exists\ y.\ y \in i)$;

cons : cons(i);

cons-params : cons-params(p, x∞, z∞) $\leftrightarrow (\exists\ y.\ x\infty \leq y \rightarrow\infty_r \wedge y \rightarrow\infty_r \leq z\infty)$;

path : cons-params(p, x∞, z∞) $\rightarrow$ mkint(p, x∞, z∞).path $= \geq_{prune}(\leq_{prune}(p, x\infty), z\infty)$;

low : cons-params(p, x∞, z∞) $\rightarrow$ mkint(p, x∞, z∞).low = x∞;

high : cons-params(p, x∞, z∞) $\rightarrow$ mkint(p, x∞, z∞).high = z∞;

state : i[y] = i.path[y];

modpath : i $|_{path}$ p = mkint(p, i.low, i.high);

modlow-1 : x∞ $\leq$ i.high $\wedge$ x∞ $\neq$ ∞ $\rightarrow$ i $|_{low}$ x∞ = mkint(i.path, x∞, i.high);

modlow-2 : i.high $<$ x∞ $\wedge$ i.high $\neq$ ∞ $\rightarrow$ i $|_{low}$ x∞ = mkint(i.path, i.high, i.high);

modlow-high : (i $|_{low}$ ∞).high = i.high;

modhigh-1 : i.low $\leq$ z∞ $\wedge$ z∞ $\neq$ -∞ $\rightarrow$ i $|_{high}$ z∞ = mkint(i.path, i.low, z∞);

modhigh-2 : z∞ $<$ i.low $\wedge$ i.low $\neq$ -∞ $\rightarrow$ i $|_{high}$ z∞ = mkint(i.path, i.low, i.low);

modhigh-low : (i $|_{high}$ -∞).low = i.low;

ext : $i_1 = i_2 \leftrightarrow i_1$.low = i_2.low $\wedge$ i_1.high = i_2.high $\wedge$ ($\forall$ y. y $\in i_1 \rightarrow i_1$[y] = i_2[y]);

subint :
$i_1 \subseteq i_2 \leftrightarrow i_2$.low $\leq i_1$.low $\wedge$ i_1.high $\leq i_2$.high $\wedge$ ($\forall$ y. y $\in i_1 \rightarrow i_1$[y] = i_2[y]);

end enrich

Spezifikation: ***path***

path =
enrich state, renamed-real **with**
sorts path;
functions

. [.]	:	path × real	$\rightarrow$	state	**prio** 2;
$\leq_{prune}$	:	path × real∞	$\rightarrow$	path	;
$\geq_{prune}$	:	path × real∞	$\rightarrow$	path	;

variables p, p_1, p_2, p_3: path;

axioms

prune-le-1 : y $\rightarrow$∞$_r$ $\leq$ x∞ $\wedge$ x∞ $\neq$ ∞ $\rightarrow$ $\leq_{prune}$(p, x∞)[y] = p[x∞ ∞$\rightarrow_r$];

prune-le-2 : x∞ $<$ y $\rightarrow$∞$_r$ $\rightarrow$ $\leq_{prune}$(p, x∞)[y] = p[y];

prune-ge-1 : z∞ $\leq$ y $\rightarrow$∞$_r$ $\wedge$ z∞ $\neq$ -∞ $\rightarrow$ $\geq_{prune}$(p, z∞)[y] = p[z∞ ∞$\rightarrow_r$];

prune-ge-2 : y $\rightarrow$∞$_r$ $<$ z∞ $\rightarrow$ $\geq_{prune}$(p, z∞)[y] = p[y];

ext : $p_1 = p_2 \leftrightarrow$ ($\forall$ y. p_1[y] = p_2[y]);

end enrich

Spezifikation: *renamed-real*

renamed-real =
rename infinity **by morphism**
  a → ar; a_0 → ar_0; b → br; c → cr; d → dr
end rename

Spezifikation: *state*

state =
specification
  sorts state;
  variables st, st_1, st_2, st_3: state;
end specification

Spezifikation: *bool*

bool =
data specification
  bool = true
    | false
    ;
  variables boolvar, $boolvar_0$: bool;
end data specification

Spezifikation: *elem*

elem =
specification
  sorts elem;
  variables a, b, c: elem;
end specification

ANHANG D

Spezifikation des funkbasierten Bahnübergangs

Im Folgenden präsentieren wir die Spezifikation des funkbasierten Bahnübergangs (FunkFahrBetrieb, FFB), so wie er in KIV eingegeben wurde. Fehlerübergänge sind wieder hinterlegt hervorgehoben. In Abbildung D.1 ist der Entwicklungsgraph dargestellt, der alle Spezifikationen enthält. Wir verwenden die Bibliotheksspezifikationen, die in Projekten immer wieder verwendet werden (*nat-basic1* ... *nat-pot*). Darauf aufbauend sind die für den FFB spezifischen Spezifikationen dargestellt. In *ffb-math* sind die Prädikate $close(pos, v, a_{brk}, gp)$ und $request(pos, v, a_{brk}, gp)$ und die Funktionen $brake_d(v, a_{brk})$, $status_d(v, a_{brk})$ und $close_d(v, a_{brk})$ mit den entsprechenden Konstanten für die Kommunikationszeiten spezifiziert (siehe Abschnitt 10.1). Die Spezifikation *train-ctrl* spezifiziert das Statechart für den Zug, *communication* für die Kommunikation, *crossing-ctrl* für den Bahnübergang und *ffb* fasst die drei *Oder*-Zustände zu dem Gesamtstatechart für die Spezifikation des funkbasierten Bahnübergangs zusammen.

Spezifikation: *ffb*

chart specification
  using using train-ctrl, communication, crossing-ctrl

  andchart ffb = train | communication | crossing;
    import variables err_cl; err_op : bool;
    end chart specification

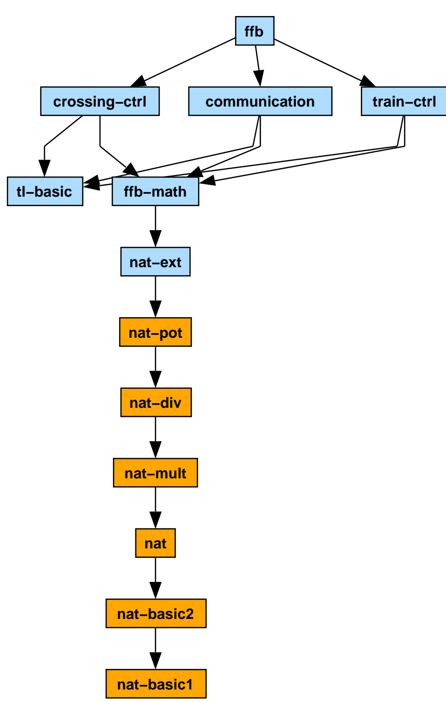


Abbildung D.1: Entwicklungsgraph für den FFB

Spezifikation: *train-ctrl*

chart specification
    using ffb-math, tl-basic

    basicchart wfs;
        variables wfs_cnt : nat;
        initial variables wfs_cnt = 0;
        static reactions tick | **begin** wfs_cnt := wfs_cnt + 1 **end**;

    orchart train = idle + wfc + brake **subcharts** wfs;
        import events ack_rcv;
        variables a, v, pos : nat;
        events save_snd ; status_snd;
        initial state idle;
        variables a = 0, pos = 0;

        transitions

```
    idle → close(pos, v, a_brk, gp) | begin save_snd := true end → wfc;
    wfc → request(pos, v, a_brk, gp) | begin status_snd := true ; wfs_cnt := 0 end
            → wfs;
    wfs → ack_rcv | → idle;
        → wfs_cnt > tc_st1 + tc_st2 | begin a := a_brk end → brake;
        → wfs_cnt > tc_st1 + tc_st2 | → brake;
  static reactions
    tick | begin pos := pos + v ; if v > a then v := v - a
                                          else v := 0 end;
end chart specification
```

Spezifikation: *communication*

```
chart specification
  using ffb-math, tl-basic

  basicchart communication;
    import events save_snd; status_snd; ack_snd;
    variables  tm1, tm2, tm3 :nat;
               act1, act2, act3 : bool;
    events ack_rcv; status_rcv; close_rcv;

    static reactions
    save_snd    | begin tm1 := tc_c ; act1 := true end;
    tick ∧ act1 | begin tm1 := tm1 - 1 end;
    tm1 = 0     | begin act1 := false; close_rcv := true end;
    tm1 = 0     | begin act1 := false end;

    status_snd  | begin tm2 := tc_st1 ; act2 := true end;
    tick ∧ act2 | begin tm2 := tm2 - 1 end;
    tm2 = 0     | begin act2 := false ; status_rcv := true end;
    tm2 = 0     | begin act2 := false end;

    ack_snd     | begin tm3 := tc_st2 ; act3 := true end;
    tick ∧ act3 | begin tm3 := tm3 - 1 end;
    tm3 = 0     | begin act3 := false ; ack_rcv := true end;
    tm3 = 0     | begin act3 := false end;
end chart specification
```

Spezifikation: ***crossing-ctrl***

chart specification
  using ffb-math, tl-basic

  basicchart closing;
    variables closing_cnt : nat;
    initial variables closing_cnt = 0;
    static reactions tick | **begin** closing_cnt := closing_cnt + 1 **end**;

  basicchart closed;
  variables closed_cnt : nat;
  initial variables closed_cnt = 0;
  static reactions tick | **begin** closed_cnt := closed_cnt + 1 **end**;

  orchart crossing = open **subcharts** closing + closed;
  import variables pos : nat; err_cl; err_op : bool;
  import events close_rcv; status_rcv;
  events ack_snd;
  initial state open;
  transitions
    open → close_rcv | → closing;
    closing → closing_cnt ≥ t_cl | → closed;
    closed → closed_cnt ≥ t_max ∨ pos = gp + 1 ∨ err_op | → open;
  static reactions
    status_rcv | **begin if** closed ∨ err_cl **then** ack_snd := true **end** ;
end chart specification

Spezifikation: ***tl-basic***

tl-basic =
specification
  predicates
    tl-dnf : bool × bool × bool × bool;
    tl-cnf : bool × bool × bool × bool;
end specification

Spezifikation: *ffb-math*

```
ffb-math =
enrich nat-ext with
  constants
      tc_c : nat;
      tc_st1 : nat;
      tc_st2 : nat;
      t_max : nat;
      t_cl : nat;
      a_brk : nat;
      gp : nat;
  functions
      close_d    :  nat × nat  →  nat  ;
      status_d   :  nat × nat  →  nat  ;
      brake_d    :  nat × nat  →  nat  ;
  predicates
      close      :  nat × nat × nat × nat;
      request    :  nat × nat × nat × nat;
  variables
      pos, v, a: nat;
      b, b1: bool;
      Ack_Sent: bool flexible;

axioms

    brake_dist : brake_d(v, a) = ((v / a + 1) ∗ (2 ∗ v - a ∗ v / a)) / 2 + v;
    status_dist : status_d(v, a) = brake_d(v, a) + (tc_st1 + tc_st2) ∗ v + v;
    close_dist : close_d(v, a) = brake_d(v, a) + (tc_c + t_cl + tc_st2) ∗ v + v;
    statusReq : request(pos, v, a, gp) ↔ pos ≥ gp - status_d(v, a);
    closeReq : close(pos, v, a, gp) ↔ pos ≥ gp - close_d(v, a);
    dec : a_brk > 0;
    tc_c : tc_c > 0;
    tc_st1 : tc_st1 > 0;
    tc_st2 : tc_st2 > 0;
    t_max : t_max > 0;
    t_cl : t_cl > 0;

end enrich
```

Spezifikation: ***nat-ext***

nat-ext =
enrich nat-pot **with**
functions

. ˆ2	:	nat	→	nat	;
. /d .	:	nat × nat	→	nat	**prio** 9;
⌈ . ⌉	:	nat	→	nat	;

axioms

square : $n\ \hat{}2 = n * n$;
DivRnd : $n \neq 0 \to m \leq \lceil m\ /d\ n \rceil * n \wedge \lceil m\ /d\ n \rceil * m < m + n$;
DivRnd-zero : $n \neq 0 \to \lceil 0\ /d\ n \rceil = 0$;

end enrich

Spezifikation: ***nat-pot***

nat-pot =
enrich nat-div **with**
functions

. ˆ .	:	nat × nat	→	nat	**prio** 12;
log	:	nat × nat	→	nat	;
2ˆ .	:	nat	→	nat	;
log2 .	:	nat	→	nat	;

axioms

Pot-zero : $n\ \hat{}\ 0 = 1$;
Pot-succ : $n\ \hat{}\ m +1 = n\ \hat{}\ m * n$;
Logdef : $m\ \hat{}\ n_0 \leq n \wedge n < m\ \hat{}\ n_0 +1 \to \log(m, n) = n_0$;
Pot2-zero : $2\hat{}\ 0 = 1$;
Pot2-succ : $2\hat{}\ n +1 = 2 * 2\hat{}\ n$;
Log2def : $2\hat{}\ m \leq n \wedge n < 2\hat{}\ m +1 \to \text{log2}\ n = m$;

end enrich

Spezifikation: ***nat-div***

nat-div =
enrich nat-mult **with**

functions

. / .	:	nat × nat	→	nat	**prio** 11;
. % .	:	nat × nat	→	nat	**prio** 11;

axioms

Divdef : n ≠ 0 → m / n ∗ n ≤ m ∧ m < (m / n)+1 ∗ n;
Moddef : n ≠ 0 → m = m / n ∗ n + m % n;

end enrich

Spezifikation: *nat-mult*

nat-mult =
enrich nat **with**
functions . ∗ . : nat × nat → nat **prio** 10;

axioms

m ∗ 0 = 0;
m ∗ n +1 = m ∗ n + m;

end enrich

Spezifikation: *nat*

nat =
enrich nat-basic2 **with**
functions . - . : nat × nat → nat **prio** 8 **left**;
predicates

. ≤ .	:	nat × nat;
. > .	:	nat × nat;
. ≥ .	:	nat × nat;

axioms

m - 0 = m;
m - n +1 = (m - n)-1;
m ≤ n ↔ ¬ n < m;
m > n ↔ n < m;
m ≥ n ↔ ¬ m < n;

end enrich

Spezifikation: ***nat-basic2***

nat-basic2 =
enrich nat-basic1 **with**
  functions . + . : nat × nat → nat **prio** 9;
  variables m, n_0: nat;

axioms

  n + 0 = n;
  m + n +1 = (m + n)+1;
  n < n_0 ∨ n = n_0 ∨ n_0 < n;
  1 = 0 +1;
  0 ≠ 1;

end enrich

Spezifikation: ***nat-basic1***

nat-basic1 =
data specification
  nat = 0
    | . +1 (. -1 : nat → nat ;)
    ;
  variables n: nat;
  order predicates . < . : nat × nat;
end data specification

Generated axioms:

  nat **freely generated by** 0, +1;
  disj : 0 ≠ n +1;
  sel : n +1 -1 = n;
  inj : n +1 = n_0 +1 ↔ n = n_0;
  case : n = 0 ∨ n = n -1 +1;
  ref : ¬ n < n;
  trans : n < n_0 ∧ n_0 < n_1 → n < n_1;
  less : n_0 < n +1 ↔ n_0 = n ∨ n_0 < n;
  less : ¬ n < 0;
  elim : n ≠ 0 → n_0 = n -1 ↔ n = n_0 +1;

ANHANG E

Syntax und Semantik sequentieller Programme

Syntax

Die Menge der Programme $PROG$ ist die kleinst mögliche Menge für die gilt:

- **skip** $\in PROG$ (leeres Programm)
- **abort** $\in PROG$ (nie terminierendes Programm)
- $\underline{x} := \underline{t} \in PROG$ (parallele/zufällige Zuweisung), mit $x_i \in X_{s_i}$ und $t_i \in T_{s_i} \cup \{?\}$
  Dabei ist X_{s_i} die Menge der Variablen und T_{s_i} die Menge der Terme der Sorte s_i.
- **var** $\underline{x} = \underline{t}$ **in** $\alpha \in PROG$ (Variablenallokation), mit $\alpha \in PROG$, $\underline{x} := \underline{t} \in PROG$
- **if** ε **then** α **else** $\beta \in PROG$ (bedingte Verzweigung), mit $\alpha, \beta \in PROG$, $\varepsilon \in BXP$
- **begin** α; β **end** $\in PROG$ (Komposition), mit $\alpha, \beta \in PROG$
- **while** ε **do** $\alpha \in PROG$ (Schleife), mit $\alpha \in PROG$ und $\varepsilon \in BXP$

Bemerkungen:

1. unnötige **begin** ... **end** Blöcke werden, wie in Pascal, weggelassen.
2. **if** ε **then** α **else skip** wird durch **if** ε **then** α abgekürzt
3. **skip** ist äquivalent zu $\underline{x} := \underline{x}$
4. **abort** ist äquivalent zu **while** true **do skip**

Hilfsfunktionen

Wir definieren die Hilfsfunktionen $read(\alpha)$ und $write(\alpha)$. Dabei verwenden wir die wie üblich definierten Funktionen $vars(t_i)$ bzw. $vars(\varepsilon)$, um die Variablen eines Terms bzw. eines Ausdrucks zu berechnen.

$$read(\alpha) := \begin{cases} \alpha \equiv \textbf{skip}, & \emptyset \\ \alpha \equiv \textbf{abort}, & \emptyset \\ \alpha \equiv \underline{x} \; := \; \underline{t}, & \bigcup_i vars(t_i) \\ \alpha \equiv \textbf{var } \underline{x} \; = \; \underline{t} \textbf{ in } \beta & (read(\beta) \backslash \bigcup_i \{x_i\}) \cup \bigcup_i vars(t_i) \\ \alpha \equiv \textbf{if } \varepsilon \textbf{ then } \beta \textbf{ else } \gamma, & vars(\varepsilon) \cup read(\beta) \cup read(\gamma) \\ \alpha \equiv \textbf{begin } \beta ; \; \gamma \textbf{ end} & read(\beta) \cup read(\gamma) \\ \alpha \equiv \textbf{while } \varepsilon \textbf{ do } \beta & vars(\varepsilon) \cup read(\beta) \end{cases}$$

$$write(\alpha) := \begin{cases} \alpha \equiv \textbf{skip}, & \emptyset \\ \alpha \equiv \textbf{abort}, & \emptyset \\ \alpha \equiv \underline{x} \; := \; \underline{t}, & \bigcup_i x_i \\ \alpha \equiv \textbf{var } \underline{x} \; = \; \underline{t} \textbf{ in } \beta & write(\beta) \backslash \bigcup_i \{x_i\} \\ \alpha \equiv \textbf{if } \varepsilon \textbf{ then } \beta \textbf{ else } \gamma, & write(\beta) \cup write(\gamma) \\ \alpha \equiv \textbf{begin } \beta ; \; \gamma \textbf{ end} & write(\beta) \cup write(\gamma) \\ \alpha \equiv \textbf{while } \varepsilon \textbf{ do } \beta & write(\beta) \end{cases}$$

Semantik

Die Semantik der Programme ist implizit über eine Algebra $\mathcal{A}$ definiert, die die Trägermenge der Sorten und die Funktionen und Prädikate definiert. Sei σ eine Belegung, dann ist die Semantik eines Programms $\sigma[\![\alpha]\!]\sigma'$ wie folgt definiert (die zugrundeliegende Algebra lassen wir wie üblich weg).[1]

- $\sigma[\![\textbf{skip}]\!]\sigma'$ genau dann, wenn $\sigma = \sigma'$
- $[\![\textbf{abort}]\!]$ ist für jedes Belegungspaar falsch (leere Relation)
- $\sigma[\![\underline{x} \; := \; \underline{t}]\!]\sigma'$ genau dann, wenn $\sigma' = \sigma[\underline{x} \leftarrow [\![\underline{t}]\!]_\sigma]$, wobei $[\![?]\!]_\sigma$ einen zufälligen Wert repräsentiert
- $\sigma[\![\textbf{var } \underline{x} \; = \; \underline{t} \textbf{ in } \alpha]\!]\sigma'$ genau dann, wenn es eine Belegung σ'' gibt, so dass: $\sigma[\![\underline{x} \; := \; \underline{t}; \; \alpha]\!]\sigma''$ und $\sigma' = \sigma''[\underline{x} \leftarrow [\![\underline{x}]\!]_\sigma]$
- $\sigma[\![\textbf{begin } \alpha; \; \beta \textbf{ end}]\!]\sigma'$ genau dann, wenn es eine Belegung σ'' gibt, so dass $\sigma[\![\alpha]\!]\sigma''$ und $\sigma''[\![\beta]\!]\sigma'$
- $\sigma[\![\textbf{if } \varepsilon \textbf{ then } \alpha \textbf{ else } \beta]\!]\sigma'$ genau dann, wenn $[\![\varepsilon]\!]_\sigma = tt$ und $\sigma[\![\alpha]\!]\sigma'$ oder $[\![\varepsilon]\!]_\sigma \neq tt$ und $\sigma[\![\beta]\!]\sigma'$

[1] $\sigma[\![\alpha]\!]\sigma'$ ist eine Relation in Inifx-Notation.

- $\sigma[\![\textbf{while } \varepsilon \textbf{ do } \alpha]\!]\sigma'$ genau dann, wenn eine endliche Folge von Belegungen $\sigma_0, \ldots, \sigma_n$ $(n \geq 0)$ existiert, so dass
  - $\sigma = \sigma_0$, $\sigma' = \sigma_n$,
  - $[\![\varepsilon]\!]_{\sigma_0} = tt, \ldots, [\![\varepsilon]\!]_{\sigma_{n-1}} = tt$, $[\![\varepsilon]\!]_{\sigma_n} = f\!f$, und
  - $\sigma_i[\![\alpha]\!]\sigma_{i+1}$ für jedes i mit $0 \leq i < n$

  (σ_i ist die Belegung nach der i-ten Schleifenwiederholung)

Semantik von Formeln in Dynamischer Logik

Die Semantik von Formeln in Dynamischer Logik (DL-Formeln) entspricht der Semantik von Formeln in Prädikatenlogik (PL-Formeln) mit zwei zusätzlichen Operatoren.

- $[\![[\alpha]\ \varphi]\!]_\sigma = tt$ genau dann, wenn für alle Belegungen σ' mit $\sigma[\![\alpha]\!]\sigma'$ gilt: $[\![\varphi]\!]_{\sigma'} = tt$ (Die Belegung σ' repräsentiert den erreichbaren Endzustand des Programms α.)
- $[\![\langle\alpha\rangle\ \varphi]\!]_\sigma = tt$ genau dann, wenn es eine Belegung σ' mit $\sigma[\![\alpha]\!]\sigma'$ gibt und $[\![\varphi]\!]_{\sigma'} = tt$. (Dies impliziert, dass das Programm α terminiert, d. h. es gibt einen erreichbaren Endzustand σ'.)

Programm Äquivalenz

Zwei Programme α, β sind *äquivalent*, kurz $\alpha \equiv \beta$, genau dann wenn, ihre Semantik in allen Modellen gleich ist, d. h. für alle $\mathcal{A} \in Alg(\Sigma)$ gilt $[\![\alpha_\mathcal{A}]\!] = [\![\beta_\mathcal{A}]\!]$.

Bemerkungen zur Dynamischen Logik

- Die Semantik der Dynamischen Logik wertet Programmformeln nicht „statisch“, mit einer fixen Variablenbelegung, sondern „dynamisch“, abhängig vom Zustand innerhalb von DL-Formeln, aus. Initiale Belegungen können Resultate einer Ausführung anderer Programme sein. Deshalb die Bezeichnung: „Dynamische Logik“.
- $[\![[\alpha]\ \varphi]\!]_\sigma = tt$ ist immer wahr, wenn α nicht terminiert. $[\alpha]\ \varphi$ modelliert also *partielle Korrektheit*. Im Gegensatz dazu ist $\langle\alpha\rangle\ \varphi$ immer falsch, wenn α nicht terminiert.
- Es gilt: $\langle\alpha\rangle\ \varphi \ \leftrightarrow\ \neg\ [\alpha]\ \neg\ \varphi$.
- Wenn $\alpha \equiv \beta$, dann gilt für alle φ, $\langle\alpha\rangle\ \varphi \leftrightarrow \langle\beta\rangle\ \varphi$.
- Wenn $x_1, \ldots, x_n$ alle Variablen der Programme α und β sind und $y_1, \ldots, y_n$ andere Variablen sind, dann gilt:

  wenn $\langle\alpha\rangle\ (x_1 = y_1 \wedge\ \ldots\ \wedge x_n = y_n) \leftrightarrow \langle\beta\rangle\ (x_1 = y_1 \wedge\ \ldots\ \wedge x_n = y_n)$, dann $\alpha \equiv \beta$
- Die beiden letzten Bemerkungen zeigen, dass Programmäquivalenz in Dynamischer Logik ausgedrückt werden kann.

ANHANG F

Grammatik von Statechart-Spezifikationen

```
chartspec         : T_CHART_SPECIFICATION optcomment
                    usedspeclist
                    chartlist
                    T_END_CHART_SPECIFICATION ;

chartlist         : chart
                  | chartlist chart ;

chart             : orchart
                  | andchart
                  | basicchart ;

orchart           : T_OR_CHART symbol T_GLEICH orcharts
                    T_STRICHPUNKT optcomment
                    chartsignature
                    initlist
                    translist
                    srlist ;
chartsignature    : importvardeflist
                    importevsymlist
                    vardeflist
                    evsymlist ;

andchart          : T_AND_CHART symbol T_GLEICH andchartsymlist
                    T_STRICHPUNKT optcomment chartsignature ;
```

```
basicchart              : T_BASIC_CHART symbol T_STRICHPUNKT optcomment
                          chartsignature
                          initvardecllistbasic
                          srlist ;

andchartsymlist         : symbol
                        | andchartsymlist T_STRICH symbol ;

srlist                  : /* empty */
                        | T_STATIC_REACTIONS srlist_nonempty ;

srlist_nonempty         : staticreaction
                        | srlist_nonempty staticreaction ;

staticreaction          : eventandaction T_STRICHPUNKT optcomment ;


translist               : /* empty */
                        | T_TRANSITIONS transitionslist_nonempty ;

transitionslist_nonempty
                        : transitionslist_nonempty transitions
                        | transitions ;

transitions             : xov shorttransitions ;

shorttransitions        : /* empty */
                        | shorttransitions T_IMPL shorttrans ;

eventandaction          : iboolexpr T_STRICH basicprog ;

eventnoaction           : basicprog ;

noeventaction           : iboolexpr T_STRICH ;

shorttrans              : eventandaction T_IMPL xov
                          T_STRICHPUNKT optcomment
                        | eventnoaction T_IMPL xov
```

```
                        T_STRICHPUNKT optcomment
                      | noeventaction T_IMPL xov
                        T_STRICHPUNKT optcomment ;

commsymlist_nonempty
                      : boolsym
                      | commsymlist_nonempty boolsym ;

boolsym               : symbol T_STRICHPUNKT optcomment

initlist              : T_INITIAL_STATE xov T_STRICHPUNKT
                        initvardecllist ;

initvardecllistbasic  : /*empty*/
                      | T_INITIAL T_VARIABLES vardecllist_nonempty
                        T_STRICHPUNKT ;

initvardecllist       : /*empty*/
                      | T_VARIABLES vardecllist_nonempty
                        T_STRICHPUNKT ;

importvardeflist      : /*empty*/
                      | T_IMPORT_VARIABLES vardeflist_aux ;

importevsymlist       : /*empty*/
                      | T_IMPORT_EVENTS commsymlist_nonempty ;

evsymlist             : /*empty*/
                      | T_EVENTS commsymlist_nonempty ;

orcharts              : orcharts_nonempty
                        T_SUBCHARTS orcharts_nonempty
                      | orcharts_nonempty
                      | T_SUBCHARTS orcharts_nonempty ;

orcharts_nonempty     : symbol
                      | orcharts_nonempty T_PLUS symbol ;
```

symbol : ... /*any symbol (not in token list)*/

Literaturverzeichnis

[AL89] M. Abadi and L. Lamport. Composing Specifications. In G. Rozenberg, J.W. de Bakker, and W.-P. de Roever, editors, *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, volume 430 of *LNCS*, pages 1–41, Berlin, Germany, 1989. Springer-Verlag.

[And86] Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof.* Academic Press, 1986.

[ASSB96] Aziz, K. Sanwal, V. Singhal, and R. K. Brayton. Verifying continuous time markov chains. In R. Alur and T. A. Henzinger, editors, *CAV'96: 8th International Conference on Computer Aided Verification*, volume 1102 of *LNCS*, pages 269–276, New Brunswick, NJ, USA, 1996. Springer-Verlag.

[B+03] M. Bozzano et al. ESACS: An integrated methodology for design and safety analysis of complex systems. In *Proceedings of European Safety and Reliability Conference (ESREL'03)*, pages 237–245, Maastricht, The Netherlands, 2003. Balkema.

[BA93] G. Bruns and S. Anderson. Validating safety models with fault trees. In J. Górski, editor, *SafeComp'93: 12th International Conference on Computer Safety, Reliability, and Security*, pages 21 – 30. Springer-Verlag, 1993.

[Bal04] M. Balser. *Verifying Concurrent System with Symbolic Execution – Temporal Reasoning is Symbolic Execution with a Little Induction.* PhD thesis, University of Augsburg, Augsburg, Gemany, 2004. (to appear).

[BBC+00] N. Bjørner, A. Brown, M. Colon, B. Finkbeiner, Z. Manna, H. Sipma, and T. Uribe. Verifying temporal properties of reactive systems: A step tutorial. In *Formal Methods in System Design*, pages 227–270, 2000.

[BCG91] R. E. Bloomfield, J. H. Cheng, and J. Górski. Towards a common safety description model. In J. Lindeberg, editor, *SafeComp'91: 10th International Conference on Computer Safety, Reliability, and Security*, pages 1–6, Trondheim, Norway, 1991. Pergamon Press.

[BCS02] P. Bieber, C. Castel, and C. Seguin. Combination of fault tree analysis and model checking for safety assessment of complex systems. In *Dependable Computing EDCC-4: 4th European Dependable Computing Conference*, volume 2485 of *LNCS*, pages 19–31, Toulouse, France, 2002. Springer-Verlag.

[BDK92] S. Biundo, D. Dengler, and J. Koehler. Deductive planning and plan reuse in a command language environment. In *Proceedings of the 10th European Conference on Artificial Intelligence*, pages 628–632. Wiley, 1992.

[BDRS02] M. Balser, C. Duelli, W. Reif, and G. Schellhorn. Verifying concurrent systems with symbolic execution. *Journal of Logic and Computation*, 12(4):549–560, 2002.

[BDW00] T. Bienmöller, W. Damm, and H. Wittke. The STATEMATE verification environment – making it real. In E. A. Emerson and A. P. Sistla, editors, *CAV'00: 12th international Conference on Computer Aided Verification*, number 1855 in LNCS, pages 561–567, Chicago, IL, USA, 2000. Springer-Verlag.

[BG92] G. Berry and G. Gonthier. The ESTEREL synchronous programming language – design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, November 1992.

[BRS+00] M. Balser, W. Reif, G. Schellhorn, K. Stenzel, and A. Thums. Formal system development with KIV. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering*, number 1783 in LNCS, pages 363–366. Springer-Verlag, 2000.

[BRSS99] M. Balser, W. Reif, G. Schellhorn, and K. Stenzel. KIV 3.0 for Provably Correct Systems. In *Current Trends in Applied Formal Methods*, LNCS 1641. Boppard, Germany, Springer-Verlag, 1999.

[BT02] M. Balser and A. Thums. Interactive verification of statecharts. In *Integration of Software Specification Techniques (INT'02)*, 2002. http://tfs.cs.tu-berlin.de/~mgr/int02/proceedings.html.

[Bäu03] S. Bäumler. Verification of UML state machines with KIV. Master's thesis, Ludwig-Maximilians-Universität München, 2003.

[BV03a] M. Bozzano and A. Villafiorita. Integrating fault tree analysis with event ordering information. In *Proceedings of ESREL'03*, pages 247–254. Balkema, 2003.

[BV03b] M. Bozzano and Adolfo Villafiorita. Improving system reliability via model checking: theFSAP/NuSMV-SA safety analysis platform. In *Proceedings of SafeComp'03*, pages 49–62. Springer-Verlag, 2003.

[CAB+01] W. Chan, R. J. Anderson, P. Beame, D. H. Jones, D. Notkin, and W. E. Warner. Optimizing symoblic model checking for statecharts. *IEEE Transactions on Software Engineering*, 27(2):170 – 190, 2001.

[CCI99] CCIB. *Common Criteria for Information Technology Security Evaluation, Version 2.1 (ISO 15408)*. Common Criteria Implementation Board, October 1999. Available at http://csrc.nist.gov/cc.

[ČdLM+97] M. Čepin, R. de Lemos, B. Mavko, S. Riddle, and A. Saeed. An object-based approach to modelling and analysis of failure properties. In P. Daniel, editor, *SafeComp'97: 16th International Conference on Computer Safety, Reliability, and Security*, pages 281 – 294. Springer-Verlag London, 1997.

[CE81] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logics of Programs*, number 131 in LNCS. Springer-Verlag, 1981.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-State Concurrent Systems using Temporal Logic. *ACM Transactions on Programming Languages and Systems*, 8:244–263, 1986.

[CHR91] Zhou Chaochen, C. A. R. Hoare, and Anders P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, December 1991.

[CMZ02] A. Cau, B. Moszkowski, and H. Zedan. *ITL – Interval Temporal Logic*. Software Technology Research Laboratory, SERCentre, De Montfort University, The Gateway, Leicester LE1 9BH, UK, 2002. www.cms.dmu.ac.uk/~cau/itlhomepage.

[Con01] The ESACS Consortium. ESACS – Enhanced safety assessment for complex systems. Technical report, ESACS, 2001.

[Con02] The ESACS Consortium. D6: Safety techniques. Technical report, ESACS, 2002.

[DH98] W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. Technical Report CS98-09, The Weizmann Institute of Science, Rehovot, Israel, 1998.

[DH99] Werner Damm and David Harel. LSC's: breathing life into message sequence charts. In *Proceedings conference on formal methods for open object-based distributed systems (FMOODS '99)*, pages 293–311. ACM Press, 1999.

[Dij76] E. W. Dijkstra. *A Discipline of Programming*, chapter 14. Prentice-Hall, Englewood Cliffs, N. J., 1976.

[DIN90] *Zuverlässigkeit: Begriffe*, 1990.

[DJHP98] W. Damm, B. Josko, H. Hungar, and A. Pnueli. A compositional real-time semantics of STATEMATE designs. In W.-P. de Roever, H. Langmaack, and A. Pnueli, editors, *COMPOS' 97*, volume 1536 of *LNCS*, pages 186–238. Springer-Verlag, 1998.

[Dug96] J. B. Dugana. *Handbook of Software Reliability Engineering*, chapter Software System Analysis Using Fault Trees. McGraw-Hill, 1996.

[ESA91] European Space Agency. *ESA Software Engineering Standards*, 1991.

[FFB96] Betriebliches Lastenheft für FunkFahrBetrieb, 1996. Stand 1.10.1996.

[FMNP95] P. Fenelon, J. McDermid, A. Nicholson, and D. Pumfrey. Experience with the application of HAZOP to computer-based systems. In *Proceedings of the 10th Annual Conference on Computer Assurance*, Gaithersburg, MD, 1995. IEEE.

[FSS+94] Th. Filkorn, H.A. Schneider, A. Scholz, A. Strasser, and P. Warkentin. SVE user's guide. Technical Report ZFE BT SE 1-SVE-1, Siemens AG, Corporate Research and Development, Munich, 1994.

[Gei99] R. Geisler. *Formal Semantics for the Integration of Statecharts and Z in a Metamodel-Based Framework*. PhD thesis, Technical University of Berlin, 1999.

[Gór94] J. Górski. Extending safety analysis techniques with formal semantics. In F. J. Redmill and T. Anderson, editors, *Technology and Assessment of Safety Critical Systems*, pages 147 – 163, London, 1994. Springer-Verlag.

[GW95] J. Górski and A. Wardziński. Formalising fault trees. In F. Redmill and T. Anderson, editors, *Achievement and Assurance of Safety*. Springer-Verlag, 1995.

[GW97] J. Górski and A. Wardziński. Timing aspects of fault tree analysis of safety critical systems. In F. Redmill and T. Anderson, editors, *Safer Systems*. Springer-Verlag, 1997.

[Han96] K. Hansen. *Linking Safety Analysis to Safety Requirements*. PhD thesis, Danmarks Tekniske Universitet, Lyngby, August 1996.

[Har84] D. Harel. Dynamic logic. In D. Gabbay and F. Guenther, editors, *Handbook of Philosophical Logic*, volume 2, pages 496–604. Reidel, 1984.

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[HKMKS03] H. Hermanns, J.-K. Katoen, J. Meyer-Kayser, and M. Siegle. A tool for model-checking markov chains. *Int. Journal on Software Tools for Technilogy Transfer*, 4(3):153–172, 2003.

[HL83] P. Harvey and N. Leveson. Analyzing software safety. *IEEE Transactions on Software Engineering*, 9(5), September 1983.

[HLN+90] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4), 1990.

[HN96] D. Harel and A. Naamad. The statemate semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, October 1996.

[HO02a] J. Hoenicke and E.-R. Olderog. Combining specification techniques for processes data and time. In M. Butler, L. Petre, and K. Sere, editors, *Integrated Formal Methods*, volume 2335 of *LNCS*, pages 245–266. Springer-Verlag, 2002.

[HO02b] J. Hoenicke and E.-R. Olderog. CSP-OZ-DC: A combination of specification techniques for processes, data and time. *Nordic Journal of Computing*, 9(3):301–334, 2002.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[Hoe99] J. Hoenicke. Specification of radio based railway crossings with the combination of CSP, OZ, and DC. In G. Smith and I. Hayes, editors, *Towards real-time Object-Z*, LNCS, pages 49–65. Springer-Verlag, 1999.

[HP98] D. Harel and M. Politi. *Modeling Reactive Systems With Statecharts: The Statemate Approach.* McGraw Hill, 1998.

[HPSS87] D. Harel, A. Pnueli, J. P. Schmidt, and R. Sherman. On the formal semantics of statecharts. In *2nd IEEE Symposium on Logic in Computer Science*, pages 54–64. IEEE Press, 1987.

[HRS94] K. M. Hansen, A. P. Ravn, and V. Stavridou. From safety analysis to formal specification. ProCoS II document [ID/DTH KMH 1/1], Technical University of Denmark, 1994.

[HRS98] K. Hansen, A. Ravn, and V. Stavridou. From safety analysis to software requirements. *IEEE Transactions on Software Engineering*, 24(7):573 – 584, July 1998.

[IEC90] Geneva: International Electronical Comission. *International Standard IEC 1025* Fault Tree Analysis (FTA), 1990.

[IEC91] International Electrotechnical Commission. *Software for Computers in the Application of Industrial Safety Related Systems*, 1991.

[IEC92] International Electrotechnical Commission. *Functional Safety of Programmable Electronic Systems: Generic Aspects*, 1992.

[IT00] ITU-T. Recommendation z.120: Message sequence charts (MSC), 2000.

[ITS91] ITSEC. *Information Technology Security Evaluation Criteria, Version 1.2.* Office for Official Publications of the European Communities, June 1991.

[JS99] L. Jansen and E. Schnieder. Referenzfallstudie Bahnübergang – Referenzfallstudie im Bereich Verkehrsleittechnik des DFG-SPP Softwarespezifikation. Technical report, Institut für Regelungs- und Automatisierungstechnik, http://www.ifra.ing.tu-bs.de/m33/spezi/, 1999. (in German).

[JW98] J. Joyce and K. Wong. Generating verification conditions through fault tree analysis and rigorous reasoning. In *Proceedings of the 16th International System Safety Conference*, 1998.

[Klo03] J. Klose. *Live Sequence Charts: A Graphical Formalism for the Specification of Communication Behavior.* PhD thesis, University of Oldenburg, Oldenburg, Germany, 2003.

[Koz83] D. Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 17(3):333–354, December 1983.

[KS76] J. G. Kemeny and J. L. Snell. *Finite Markov Chains.* Springer-Verlag, 1976.

[KT00] J. Klose and A. Thums. Das Statemate-Referenzmodell der Referenzfallstudie Verkehrsleittechnik. Technical report, Universität Oldenburg, Universität Augsburg, 2000. to appear (in German).

[KT02] J. Klose and A. Thums. The STATEMATE reference model of the reference case study 'Verkehrsleittechnik'. Technical Report 2002-01, Universität Augsburg, 2002.

[Lam94] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3), 1994.

[LC91] N. Leveson and S. Cha. Safety verifikation of ADA programms using software fault trees. *IEEE Software*, pages 48 – 59, July 1991.

[Lei95] R. D. Leitsch. *Reliability Analysis for Engineers: An Introduction.* Oxford Science Publications, 1995.

[Lev95] N. Leveson. *Safeware: System Safety and Computers.* Addison Wesley, 1995.

[LH83] N. Leveson and P. Harvey. Analyzing software safety. *IEEE Transactions on Software Engineering*, 9(5):569 – 579, September 1983.

[Lio96] J. L. Lions. Ariane 5 – flight 501 failure. Technical Report 33, ESA, 1996. available at http://www.esa.int/export/esaCP/Pr_33_1996_p_EN.html.

[LMB92] J. R. Levine, T. Mason, and D. Brown. *Lex* & *Yacc*. O'Reilly & Associates, 2nd/updated edition, 1992.

[LMS97] Y. Lakhnech, E. Mikk, and M. Siegel. Hierarchical automata as model for statecharts. In *Asian Computing Science Conference (ASIAN'97)*, volume 1345 of *LNCS*. Springer-Verlag, 1997.

[LP99] J. Lilius and I. Porres Paltor. Formalising UML state machines for model checking. In *Second International Conference on the Unified Modeling Language: UML'99*, 1999.

[LT93] N. G. Leveson and C. S. Turner. An investigation of the therac-25 accidents. *IEEE Computer*, 26(7), 1993.

[LvdBC00] Gerald Luettgen, Michael von der Beeck, and Rance Cleaveland. A compositional approach to statecharts semantics. In *Technical Report of ICASE, NASA Langley Research Center, Hampton, VA*. ICASE Report No. 2000-12, NASA/CR-2000-210086, 2000.

[Mai97] T. Maier. FMEA and FTA to support safe design of embedded software in safety-critical systems. In *Safety and reliability of software based systems*, number 12 in CSR. Springer-Verlag, 1997.

[McM90] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1990.

[MoD91] Ministry of Defence. *The Procurement of Safety Critical Software in Defence Equipment (Part 1: Requirements, Part 2: Guidance). Interim Defence Standard 00-55*, 1991.

[Mos85] B. Moszkowski. A temporal logic for multilevel reasoning about hardware. *IEEE Computer*, 18(2):10–19, 1985.

[MTH89] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1989.

[OMG03] Object Management Group. *Unified Modelling Language Specification, Version 1.5*, March 2003. http://www.omg.org.cgi-bin/apps/doc?formal/03-03-01.pdf.

[OR04a] F. Ortmeier and W. Reif. Failure-sensitive specification: A formal method for finding failure modes. Technical Report 3, Institut für Informatik, Universität Augsburg, 2004.

[OR04b] F. Ortmeier and W. Reif. Safety optimization: A combination of fault tree analysis and optimization techniques. Technical Report 5, Institut für Informatik, Universität Augsburg, 2004.

[ORS+02] F. Ortmeier, W. Reif, G. Schellhorn, A. Thums, B. Hering, and H. Trappschuh. Safety analysis of the height control system for the Elbtunnel. In *SafeComp 2002*, pages 296 – 308, Catania, Italy, 2002. Springer LNCS 2434.

[ORS+03] F. Ortmeier, W. Reif, G. Schellhorn, A. Thums, B. Hering, and H. Trappschuh. Safety analysis of the height control system for the Elbtunnel. *Reliability Engineering and System Safety*, 81(3):259–268, 2003.

[OT02] F. Ortmeier and A. Thums. Formale Methoden und Sicherheitsanalyse. Technical Report 15, Universität Augsburg, 2002. (in German).

[Pan01a] P. K. Pandya. Model checking CTL*[DC]. In T. Margaria and W. Yi, editors, *Prodeedings of TACAS 2001*, LNCS 2031, Genova, Italy, 2001. Springer-Verlag Berlin Heidelberg.

[Pan01b] P. K. Pandya. Specifying and deciding qauntified discrete-time duration calculus formulae using DCVALID. In *Proceedings of Workshop on Real-Time Tools RT-TOOL 2001*, Aalborg, Denmark, August 2001.

[Pap03] Y. Papadopoulos. Model-based system monitoring and diagnosis of failures using statecharts and fault trees. *RESS Reliability Engineering and System Safety*, 81:325–341, 2003.

[Par93] H. Partsch. Formal problem specification on an algebraic basis. In S. Schumann B. Möller, H. Partsch, editor, *Formal Program Development*, Lecture Notes in Computer Science 755. Springer-Verlag, 1993.

[PM01] Y. Papadopoulos and M. Maruhn. Model-based automated synthesis of fault trees from simulink models. In *Proceedings of DSN'2001, Int. Conference of Distributed System Networks*, pages 77–82. IEEE Computer Society, 2001.

[PMM+01] Y. Papadopoulos, J. McDermid, A. Mavrides, C. Scheidler, and M. Maruhn. Model-based semiautomatic safety analysis of programmable systems in automotive applications. In *Proceedings of ADAS 2001, Int. Conference on Advanced Driver Assistance Systems*, pages 53–57. IEE publications, 2001.

[Pro03] AVACS Funding Proposal. AVACS – Automatic verification and analysis of complex systems. Technical report, University of Oldenburg, 2003.

[PS91] A. Pnueli and M. Shalev. What is in a step: On the semantics of statecharts. In *Symposium on Theoretical Aspects of Computer Software*, volume 526 of *LNCS*, pages 244–264. Springer-Verlag, 1991.

[PS98] J. Philipps and P. Scholz. Formal verification and hardware design with statecharts. In B. Möller and J. V. Tucker, editors, *Prospects for Hardware Foundations*, volume 1546 of *LNCS*. Springer-Verlag, 1998.

[Rau02] Antoine Rauzy. Mode automata and their compilation into fault trees. *Reliability Engineering and System Safety*, 78:1–12, 2002.

[Rei79] D. Reifer. Software failure modes and effects analysis. *IEEE Transactions on Reliability*, 28(3):147 – 249, August 1979.

[Rei95] W. Reif. The KIV-approach to Software Verification. In M. Broy and S. Jähnichen, editors, *KORSO: Methods, Languages, and Tools for the Construction of Correct Software – Final Report*, LNCS 1009. Springer-Verlag, Berlin, 1995.

[RK97] Jürgen Ruf and Thomas Kropf. Symbolic Model Checking for a Discrete Clocked Temporal Logic with Intervals. In E. Cerny and D.K. Probst, editors, *Conference on Correct Hardware Design and Verification Methods (CHARME)*, pages 146–166, Montreal, 1997. IFIP WG 10.5, Chapman and Hall.

[RK99] J. Ruf and T. Kropf. Modeling real-time systems with I/O-interval structures. In *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen ud Systemen*. Shaker Verlag, 1999.

[RR00] G. Reggio and L. Repetto. Casl-Chart: A combination of statecharts and of the algebraic specification language Casl. Technical report, DISI Universitá di Genova, 2000.

[RSS95] W. Reif, G. Schellhorn, and K. Stenzel. Interactive Correctness Proofs for Software Modules Using KIV. In *COMPASS'95 – Tenth Annual Conference on Computer Assurance*, Gaithersburg (MD), USA, 1995. IEEE press.

[RSSB98] W. Reif, G. Schellhorn, K. Stenzel, and M. Balser. Structured specifications and interactive proofs with KIV. In W. Bibel and P. Schmitt, editors, *Automated Deduction—A Basis for Applications*, pages 13 – 39. Kluwer Academic Publishers, Dordrecht, 1998.

[RST00a] W. Reif, G. Schellhorn, and A. Thums. Formale Sicherheitsanalyse einer funkbasierten Bahnübergangsteuerung. In E. Schnieder, editor, *Forms2000 – Formale Techniken für die Eisenbahnsicherung*, volume Reihe 12, Nr. 441 of *Fortschritt-Bericht VDI*, 2000.

[RST00b] W. Reif, G. Schellhorn, and A. Thums. Safety analysis of a radio-based crossing control system using formal methods. In E. Schnieder and U. Becker, editors, *9th IFAC Symposium Control in Transportation Systems 2000*, June 2000.

[RST01] W. Reif, G. Schellhorn, and A. Thums. Integration formaler Spezifikation und Sicherheitsanalyse. Technical Report 6, Universität Augsburg, 2001. (in German).

[Ruf99] J. Ruf. *Techniken zur Modellierung und Verifikation von Echtzeitsystemen*. PhD thesis, Universität Karlsruhe, Germany, 1999. (in German).

[Ruf00] J. Ruf. RAVEN: Real-time analyzing and verification environment. Technical Report WSI 2000-3, University of Tübingen, Wilhelm-Schickard-Institute, January 2000.

[SA91] V. Sperschneider and G. Antoniou. *Logic: A Foundation for Computer Science*. Addison Wesley, 1991.

[SA97] G. Schellhorn and W. Ahrendt. Reasoning about Abstract State Machines: The WAM Case Study. *Journal of Universal Computer Science (J.UCS)*, 3(4):377–413, 1997. available at http://hyperg.iicm.tu-graz.ac.at/jucs/.

[Sch99] G. Schellhorn. *Verifikation abstrakter Zustandsmaschinen*. PhD thesis, Universität Ulm, Fakultät für Informatik, 1999. (in German).

[Sch01] Andreas Schäfer. Fehlerbaumanalyse und Model-Checking. Master's thesis, Universität Oldenburg, 2001. (in German).

[Sch03] A. Schäfer. Combining real-time model-checking and fault tree analysis. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003: Formal Methods*, LNCS 2805, pages 522–541. Springer-Verlag, 2003.

[Sim87] I-Logix. *Simulation Reference Manual*, 1987.

[Smi00] G. Smith. *The Object-Z Specification Language*. Kluwer Academic Publisher, 2000.

[Sof98] Integration von Techniken der Softwarespezifikation für ingenieurwissenschaftliche Anwendungen. http://tfs.cs.tu-berlin.de/projekte/indspec/SPP/, 1998.

[Ste01] Kurt Stenzel. Verification of JavaCard Programs. Technical report 2001-5, Institut für Informatik, Universität Augsburg, Germany, 2001. Available at http://www.Informatik.Uni-Augsburg.DE/swt/fmg/papers/.

[Sti92] C. Stirling. *Handbook of Logic in Computer Science*, volume 2, chapter Modal and Temporal Logics. Oxford University Press, 1992.

[Sto96] N. Storey. *Safety-Critical Computer Systems*. Addison-Wesley, 1996.

[STR02] G. Schellhorn, A. Thums, and W. Reif. Formal fault tree semantics. In *Proceedings of The Sixth World Conference on Integrated Design & Process Technology*, Pasadena, CA, 2002.

[Tap01] J. Tapken. Model-checking of duration calculus specifications. Master's thesis, University of Oldenburg, June 2001. http://semantik.informatik.uni-oldenburg.de/projects/.

[TS02] A. Thums and G. Schellhorn. Formal safety analysis in transportation control. In R. Slovak and E. Schnieder, editors, *Proceedings of the Workshop on Software Specification of Safety Relevant Transportation Control Tasks*, Braunschweig, Germany, 2002. VDI-Verlag.

[TS03] A. Thums and G. Schellhorn. Model checking FTA. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003: Formal Methods*, LNCS 2805, pages 739–757. Springer-Verlag, 2003.

[TSR02] A. Thums, G. Schellhorn, and W. Reif. Comparing fault tree semantics. In D. Haneberg, G. Schellhorn, and W. Reif, editors, *FM-TOOLS 2002*, Technical Report 2002-11, pages 25 – 32. Universität Augsburg, 2002.

[vdB94] M. von der Beeck. A comparison of statecharts variants. In H. Langmaak, W.-P. de Roever, and J. Vytopil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *LNCS*. Springer-Verlag, 1994.

[VDI95] Sicherheitstechnische Begriffe für Automatisierungssysteme –Blatt 4. VDI-Verlag, 1995.

[VGRH81] W. E. Vesely, F. F. Goldberg, N. H. Roberts, and D. F. Haasl. *Fault Tree Handbook*. Washington, D.C., 1981. NUREG-0492.

[Wir90] M. Wirsing. *Algebraic Specification*, volume B of *Handbook of Theoretical Computer Science*, chapter 13, pages 675 – 788. Elsevier, Oxford, 1990.

[ZH97] Zhou Chaochen and M. R. Hansen. Duration calculus: Logical foundations. In *Formal Aspects of Computing*, pages 283–330, 1997.

[ZHR91] Zhou Chaochen, C. A. R. Hoare, and A. P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, December 1991.