Time Partition Testing

—

Systematischer Test des kontinuierlichen Verhaltens von eingebetteten Systemen

von Diplom-Informatiker
Eckard Lehmann

von der Fakultät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

- Dr.-Ing. -

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender:	Prof. Dr. Klaus Obermayer
Gutachter:	Prof. Dr. Peter Pepper
Gutachter:	Prof. Dr. Stefan Jähnichen

Tag der wissenschaftlichen Aussprache: 18. November 2003

Berlin 2004
D 83

Vorwort

Bereits am frühen Morgen des 9. September 1945 waren die Mathematikerin Dr. Grace Murray Hopper und ihre Kollegen an der Harvard University intensiv mit Mark II beschäftigt. Mark II – ein Relais-basierter, für seine Zeit hochmoderner Computer – wies an diesem Morgen unerwarteterweise Störungen auf. Das Team begann sofort mit der Suche nach der Ursache des Problems. Noch am selben Tag um 15:45Uhr wurde der Fehler schließlich mit einer sehr ungewöhnlichen Entdeckung gefunden. Niemand konnte zu diesem Zeitpunkt schon ahnen, welche Auswirkung diese Entdeckung noch Jahre später auf die Computerwelt und die Informatik haben wird.

Am Morgen des besagten Tages flog eine Motte in das Relais Nummer 70 von Mark II, blockierte es und verursachte dadurch den ersten tatsächlichen „Bug“ in der Geschichte der Informatik. Nachdem das Insekt mit einer Pinzette entfernt wurde, war das Problem behoben. Die Motte wurde in das Logbuch mit der Notiz *“First actual case of bug being found.”* (siehe Abb. 1) geklebt. Es wird heute oft behauptet, dass die Mathematikerin Grace Hopper den Bug selbst gefunden hat, obwohl es keinerlei Dokumente gibt, die belegen können, dass sie überhaupt anwesend war, als die Motte aufgespürt wurde.

Interessanterweise referenzieren die meisten Enzyklopädien der Informatik dieses Ereignis an der Harvard University als Ursprung des Begriffs *Bug* in der Bedeutung eines Fehlers. Obwohl dies eine weit verbreitete Annahme ist, ist sie in Wirklichkeit falsch. Tatsächlich gab es den Begriff *Bug* als Synonym für einen Fehler oder einen technischen Defekt bereits lange Zeit bevor an die ersten Computer überhaupt zu denken war.

Die historische Bedeutung des Begriffes *Bug* wird auf den Walisischen Ausdruck *bwg* (=bug) zurückgeführt, der *Geist* bedeutet und bereits 1707 in Lhwyd's *Archaeologia Britannica* zitiert wurde. Erst später wurde der *Bug* als Begriff für ein Insekt populär und verdrängte die ursprüngliche Bedeutung nahezu. Ein Bug im historischen Sinne stellte ein imaginäres Objekt der Angst und des Schreckens dar. *To swear by no bugs* bedeutete eine ernsthaften und aufrichtigen Schwur oder Eid zu leisten [MB+33, p.1159]. Diese ursprüngliche Bedeutung gewann bereits Ende des 19. Jahrhunderts wieder an Popularität. Am 18. November 1878 verwendete Edison den Begriff Bug in einem Brief an

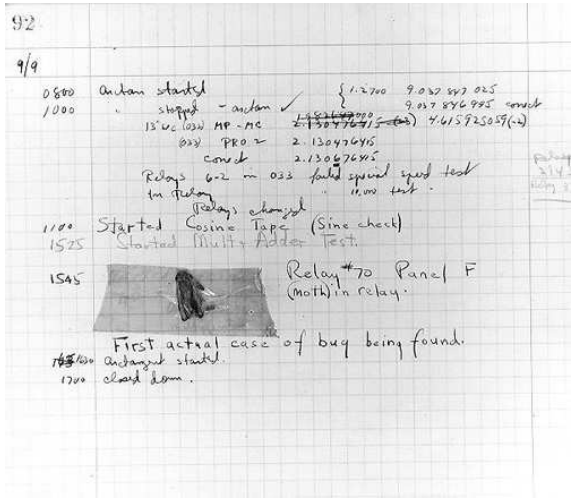

Abbildung 1: Log book entry[1]

Theodore Puskas, in dem er schrieb:

> „The first step is an intuition – and comes with a burst, then difficulties arise. This thing gives out and then that – "Bugs" – as such little faults and difficulties are called – show themselves . . . "

Diese Bedeutung von *Bug* als Fehler in Maschinen war bereits 1934 so geläufig, dass sie in Webster's New International Dictionary aufgenommen wurde: *bug, n ... 3. A defect in apparatus or its operation. . . (Slang, U.S.).*

An der Historie des Begriffes *Bug* einerseits und seiner irrtümlichen Erläuterung in modernen Nachschlagewerken andererseits erkennt man, welche Relevanz „Bugs" in der zweiten Hälfte des 20. Jahrhunderts – dem Anbruch des Computer-Zeitalters – erlangt haben. Häufigkeit und Stellenwert von technischen Fehlern in der heutigen Welt der Computer haben die ursprüngliche Bedeutung des Begriffes nahezu ersetzt. Dies führt so weit, dass der historische Ursprung eines Bugs in Vergessenheit geriet.

Jedem Software-Entwickler ist der Begriff Bug geläufig, denn nahezu jedes Programm enthält Fehler. Diese These ist sogar heute noch berechtigt, trotz jahrelanger intensiver Forschung auf dem Gebiet der Informatik. Die populären Begriffe der letzten Jahre wie „Millenium Bug" und das „Jahr 2000 Problem" kleiden diese Tatsache in Worte. Die Häufigkeit der Verwendung und die Popularität des Begriffes *Bug* in der Informatik sind ein Anzeichen

[1]Mit freundlicher Genehmigung der NSF SUCCEED Engineering Database[Tra96]

eines der kritischsten und gefährlichsten Probleme des Computer-Zeitalters: Die Komplexität von Software und die Verantwortung, die Computern übertragen wird, steigt Tag für Tag enorm an. Und mit der Komplexität steigt auch die Instabilität und die Anzahl potenzieller Fehler der Systeme. Konsequenterweise sind Bugs deshalb heute integraler, einkalkulierter Bestandteil der Informatik. Erfahrene Programmierer würden nur äußerst selten behaupten, dass ihre Software, die eine gewisse Größe und Komplexität erreicht hat, keine Fehler mehr enthält. Um so mehr war ich über eine Aussage erstaunt, die ich vor einigen Jahren in Donald Knuth's TEX-Buch "TEX: The Program" fand. Er behauptete in seinem Vorwort:

> „I believe that the final bug in TEX was discovered and removed on November 27, 1985. But if, somehow, an error still lurks in the code, I shall gladly pay a finder's fee of $20.48 to the first person who discovers it. (This is twice the previous amount, and I plan to double it again in a year; you see, I really am confident!)" [Knu86]

Solch optimistische Äußerungen sind sehr rar – vor allem unter erfahrenen Programmieren – da diese die Ernsthaftigkeit von Bugs meist realistischer einschätzen als Anfänger. Am Ende musste Knuth jedenfalls 5 Mal einen Betrag von $81.92 bezahlen, so dass er schließlich sein Angebot zurückzog, den Betrag jährlich zu verdoppeln. Momentan beläuft sich die Prämie auf $327.68 [Nol99].

Die Herausforderung und auch die Schwierigkeit qualitativ hochwertige, fehlerfreie Software zu schreiben, wurde bereits vor vielen Jahren erkannt. Im Laufe der Zeit haben Forscher an vielversprechenden Verfahren und Methoden wie formalen Programmbeweisen, Programmdeduktion und anderen fortgeschrittenen Cleanroom-Techniken gearbeitet. Die Anwendung dieser Techniken auf große praktische Projekte in der Industrie erweist sich jedoch immer noch als sehr kostspielig und deshalb häufig als inakzeptabel. Dadurch ist die älteste Methode der Fehleridentifikation für Software auch heute noch die beste in der Praxis: der Test. Obwohl das Testen eine der pragmatischsten Methoden der Qualitätssicherung ist, konnte es erstaunlicherweise seine Position als am weitesten verbreitetes Verfahren bis heute halten. Natürlich haben sich die Testverfahren über die Jahre ebenfalls enorm verbessert; der pragmatische Kern des Ansatzes blieb jedoch unberührt.

Die Idee und die Grundlage der vorliegenden Dissertation entstand während meiner täglichen Arbeit beim Software-Technologie-Forschungslabor der DaimlerChrysler AG. Durch meine berufliche Tätigkeit bekam ich detaillierten Einblick in zahlreiche Software-Entwicklungsprojekte für eingebettete softwarebasierte Fahrzeug-Steuergeräte. Hierbei wurde mir die Problematik bewusst, dass der Test in der Praxis äußerst wichtig ist, durch seinen pragmatischen Charakter in der Durchführung aber die Gefahr birgt, dass essenzielle

Fehler unerkannt bleiben. Diese Problematik inspirierte mich dazu, mich mit den Problemen des Tests eingebetteter Systeme intensiver zu beschäftigen und Lösungen zu suchen, die die Effizienz und den Wirkungsgrad des Tests durch ein gezieltes, systematisches Vorgehen verbessern.

Danksagung

Die vorliegende Arbeit wurde als Dissertation zur Erlangung des akademischen Grades Dr.-Ing. dem Fachbereich Informatik der Technischen Universität Berlin vorgelegt und genehmigt. Die wissenschaftliche Aussprache fand am 18.11.2003 statt. Herrn Prof. Peter Pepper und Herrn Prof. Stefan Jähnichen danke ich für ihre Bereitschaft zur Betreuung dieser Dissertation sowie für viele wertvolle Anregungen in Bezug auf Inhalt und Aufbau der Arbeit.

Mein besonderer Dank gilt meinem Kollegen Andreas Krämer, der die Ideen der vorliegenden Arbeit durch viele nützliche Diskussionen und Anregungen sowie durch die aktive Unterstützung bei der Entwicklung der Werkzeugumgebung positiv beeinflusst hat. Darüber hinaus möchte ich mich bei meinen Kollegen Christian Klaproth, Andreas Fett und Joachim Wegener bedanken, die ebenfalls in vielen wertvollen Diskussionen zum Gelingen dieser Arbeit beigetragen haben. Frank Lattemann danke ich für die Diskussionen im Frühstadium der Arbeit, die zur Fokussierung meines Dissertationsthemas geführt haben.

Meinen Eltern und Rahel Bringmann möchte ich für die vielen aufmunternden Worte, ihr Verständnis und die langjährige Rücksichtnahme und Geduld meinen besonderen Dank aussprechen.

Inhaltsverzeichnis

1 Einleitung **1**
- 1.1 Ursachen von Fehlern . . . 3
- 1.2 Über die Popularität des Tests . . . 8
- 1.3 Nachteile des Tests . . . 9
- 1.4 Ein einfaches Testmodell . . . 11

2 Einführung zu TPT **19**
- 2.1 Kontinuierliches Verhalten . . . 19
- 2.2 Anwendung von TPT: Ein Beispiel . . . 21
- 2.3 Testprozess mit TPT . . . 24

3 Kontinuierliches Verhalten **27**
- 3.1 Modellierung der Zeit . . . 27
- 3.2 Kontinuierliche Modellierung . . . 29
- 3.3 Stromverarbeitende Funktionen . . . 31
  - 3.3.1 Komponenten . . . 33
  - 3.3.2 Parallelisierung . . . 36
  - 3.3.3 Der Feedback-Operator . . . 37
  - 3.3.4 Ausdrücke und Gleichungen . . . 38
  - 3.3.5 Temporale Prädikate . . . 39
  - 3.3.6 Sequenzialisierung . . . 39
  - 3.3.7 Verkettung und Schaltfolgen . . . 41
- 3.4 Hybride Systeme . . . 42
  - 3.4.1 Hybride Systeme mit Aktionen . . . 45
  - 3.4.2 Termination . . . 46
- 3.5 Diskretisiertes Verhalten . . . 47
- 3.6 Zusammenfassung . . . 50

4 Testfallmodellierung **53**
- 4.1 Ausführbare Testfälle . . . 54
- 4.2 Modellierungstechniken . . . 55
- 4.3 Szenarien . . . 58

4.3.1 Schnittstelle zum System . . . 58
4.3.2 Deklarationen von Kanälen . . . 59
4.3.3 Signatur von Szenarien . . . 61
4.3.4 Intuitive Semantik . . . 62
4.3.5 Ausführbare Semantik . . . 62
4.4 Direkte Definition . . . 64
4.4.1 Intuitive Semantik . . . 65
4.4.2 Ausführbare Semantik . . . 65
4.4.3 Ausdrücke für Gleichungen . . . 67
4.4.4 Grafische Definition unabhängiger Verläufe . . . 74
4.5 Time Partitioning . . . 77
4.5.1 TP-Diagramme . . . 77
4.5.2 Intuitive Semantik . . . 86
4.5.3 Ausführbare Semantik . . . 87
4.6 Zusammenfassung . . . 90

5 Systematische Auswahl 93
5.1 Variation des Verhaltens . . . 95
5.1.1 Formen der Variation . . . 98
5.1.2 Testlets . . . 102
5.1.3 Beispielanwendung . . . 105
5.2 Kombination von Szenarien . . . 109
5.2.1 Klassifikationsbaum-Methode . . . 110
5.2.2 Testlets und Klassifikationsbäume . . . 112
5.2.3 Szenarien in der Kombinationstabelle . . . 115
5.2.4 Automatisierung der Kombinatorik . . . 115
5.3 Zusammenfassung . . . 117

6 Testdurchführung 119
6.1 Architektur . . . 120
6.2 Repräsentation von Testfällen . . . 122
6.3 Testengines . . . 123
6.4 Repräsentation der Testrecords . . . 124
6.5 Zusammenfassung . . . 125

7 Testauswertung 127
7.1 Auswertung als Abstraktion . . . 128
7.2 Auswertungssprache . . . 129
7.2.1 Intervalle der Auswertung . . . 131
7.2.2 Integration in die Testmodellierung . . . 132
7.3 Online- vs. Offline-Auswertung . . . 133
7.4 Zusammenfassung . . . 133

8 Das Testwerkzeug TPT **135**
8.1 Testfallmodellierung 135
8.2 Systematische Auswahl 138
8.3 Testdurchführung 138
8.4 Testauswertung 139

9 Zusammenfassung und Ausblick **141**

A Beweise **147**

B Sprachbeschreibung **155**
B.1 Ausdrücke 155
B.2 Testfalldefinition 161

Kapitel 1

Einleitung

In der heutigen Welt gibt es Millionen von Computern in verschiedensten Bereichen des täglichen Lebens. Hierzu zählen Weichensteuerungen, Ampeln, Flugzeuge, Fahrstühle, Registrierkassen, Atomkraftwerke, Autos, Satelliten, Operationssäle und Banken, um nur einige zu nennen. So gesehen sind Computer sehr nützlich und dienen dazu, das Leben einfacher und sicherer zu machen. Sie übernehmen unangenehme Aufgaben oder Aufgaben, die Menschen ohne Computer nicht bewältigen könnten. Die neuen Möglichkeiten, die Computer der Menschheit bieten, erklären die ungebremste Beliebtheit und den ständig steigenden Einsatz von Computern. Sie sind ein völlig neuartiges Hilfsmittel, das sowohl die wirtschaftliche als auch die kulturelle Entwicklung stark beschleunigt.

Die Akzeptanz und die Popularität von Computern wächst ständig, und auf der Basis dieses Optimismus wachsen auch die Aufgaben und Verantwortlichkeiten, die Computern anvertraut und zugebilligt werden. Optimistisch betrachtet, ist diese Entwicklung sehr begrüßenswert.

Von einem kritischen Standpunkt aus gesehen, ist die Entwicklung von Computersystemen und Software sowie generell von komplexen Systemen vom menschlichen Gehirn jedoch nicht mehr beherrschbar. Die wachsende Komplexität der Systeme und die Unfähigkeit des Gehirns, diese Komplexität korrekt zu erfassen und zu beherrschen, verursacht zwangsläufig Fehler. Ein Softwarefehler beruht nach dieser Auffassung immer auf einem fehlerhaften Verständnis des Programmierers, Testers oder Reviewers vom Programmverhalten: die menschliche Interpretation weicht also von der tatsächlichen Bedeutung, wie sie die Maschine berechnet, ab.

Die um die Jahrtausendwende populär gewordenen Begriffe „Millennium Bug“ und „Jahr 2000 Problem“ sind ein treffendes Beispiel für dieses Fehlerphänomen: Die Analyse bestehender Software hinsichtlich der Frage, ob sie im Jahr 2000 reibungslos funktioniert, kostete Milliarden. Und trotz dieser

hohen Summen, die für die Minimierung des Fehlerrisikos ausgegeben wurden, konnten selbst große Unternehmen nicht garantieren, dass ihre Dienste am Neujahrstag uneingeschränkt funktionieren. Einige von ihnen verkündeten offiziell, dass die Kunden mit Verspätungen, Ausfällen und Störungen rechnen sollten.

Mit steigender Komplexität, Funktionalität und Verantwortung von Computern wächst auch die Abhängigkeit von ihrer präzisen Arbeitsweise und ihrer Verfügbarkeit. Arbeitet das Datenbanksystem einer Bank fehlerhaft oder fällt es aus, verursacht dies einen enormen wirtschaftlichen Schaden. Die Rolle von Computern in Operationssälen, Autos, Flugzeugen und Atomkraftwerken macht deutlich, dass Fehler in solchen Systemen eine Gefahr für Leben, Gesundheit, Wirtschaft und Gesellschaft darstellen können. Zukünftige Systeme wie autonome Automobile, führerlose Züge, vollständig autonome Flugzeuge machen deutlich, dass die Vermeidung von Fehlern in den kommenden Jahren eine noch höhere Bedeutung erlangen wird.

Softwarefehler sind heute und in Zukunft teuer und zum Teil gefährlich, vor allem, wenn sie „zu spät“ erkannt werden. Die Fehlervermeidung und die Fehlererkennung bereits während der Entwicklung von Software sind aus diesem Grund eine der größten Herausforderungen der modernen Softwareentwicklung. Aus dem Software-Engineering sind viele verschiedene Design-, Analyse- und Verifikationsmethoden bekannt, die der Fehlervermeidung – zum Teil bereits in sehr frühen Entwicklungsphasen – dienen. Trotz dieser Fortschritte bleiben Fehler ein ernsthaftes Problem, so dass die Forschung auf diesem Gebiet auch in Zukunft unbedingt notwendig ist.

Ziel der Arbeit. Ziel der vorliegenden Arbeit ist es, zur Fehlerreduktion in komplexen, vorrangig eingebetteten softwarebasierten Systemen durch ein neues, systematisches Testverfahren – das Time Partition Testing – beizutragen. Eingebettete Systeme beeinflussen reale Umgebungen und kommunizieren mit dieser Umgebung in der Regel über kontinuierliche Größen. Für diese Systemklasse gibt es gegenwärtig keine spezialisierten systematischen Ansätze, die einen sorgfältigen Funktionstest unterstützen. Der wachsenden Bedeutung und Komplexität eingebetteter Systeme muss jedoch auch beim Test Rechnung getragen werden. Durch meine berufliche Tätigkeit bei der Software-Technologie-Forschung der DaimlerChrysler AG hatte ich die Möglichkeit, die praktischen Probleme beim Test des kontinuierlichen Verhaltens von komplexen eingebetteten Systemen detailliert kennenzulernen. Ein zentrales Anliegen bei der Entwicklung des Time Partition Testing war es deshalb, eine Lösung der Probleme aus der Praxis zu liefern.

In der Praxis fordert das Testen häufig einen hohen Anteil von 50% des Projekt-Budgets oder mehr. Durch verbesserte Vorgehensweisen lässt sich die Effizienz des Tests bezüglich der Ressourcen deutlich verbessern. Aus diesem

Grund versucht das Time Partition Testing neben der Definition eines systematischen Vorgehens zur Testdatenauswahl auch den Testaufwand durch ein vernünftiges Maß an Testautomatisierung zu verringern. Automatisierung und Toolunterstützung vereinfachen den Test und fördern damit auch die Akzeptanz neuer Ansätze in der Praxis.

Aufbau der Arbeit. Im weiteren Verlauf des Kapitels 1 wird zunächst ein Überblick über das Testen als wichtige Maßnahme zur Qualitätssicherung gegeben. Zu diesem Zweck werden die inhärenten Vor- und Nachteile des Tests diskutiert und ein sehr einfaches Modell aufgestellt, mit dessen Hilfe das Grundprinzip des Tests erläutert wird.

Im anschließenden Kapitel 2 wird eine kurze Einführung in die Idee und die Wirkungsweise von TPT gegeben. Hierzu wird zunächst der Begriff des kontinuierlichen Verhaltens erläutert. Anschließend wird anhand eines kurzen Beispiels ein durchgängiges Anwendungsszenario von TPT beschrieben. Schließlich wird, ausgehend von dem Beispiel, der Testprozess mit TPT erläutert.

Das Kapitel 3 enthält die Definition der semantischen Grundlagen der Modellierung von kontinuierlichem Verhalten. Hierbei spielen vor allem die stromverarbeitenden Funktionen sowie hybride Systeme eine zentrale Rolle, da diese die theoretische Basis der Modellierung von Testfällen mit dem Time Partition Testing sind. Im Kapitel 4 wird diese Modellierungstechnik für Testfälle ausführlich erläutert. Neben einer Technik zur Modellierung eines Testfalls bietet das Time Partition Testing eine systematische Unterstützung bei der strukturierten Definition einer großen Anzahl von Testfällen. Diese Systematik wird im Kapitel 5 eingeführt.

Im Kapitel 6 wird erläutert, wie modellierte Testfälle in unterschiedlichen Testumgebungen ausgeführt werden. Die Beschreibungsmittel und Techniken zur integrierten automatischen Auswertung der Ergebnisse der Testdurchführung werden im Kapitel 7 vorgestellt. Abschließend folgt eine Darstellung einer praktischen Umsetzung der TPT-Werkzeugumgebung im Kapitel 8 sowie eine Zusammenfassung im Kapitel 9.

1.1 Ursachen von Fehlern

Die Notwendigkeit von Tests ist unter Softwareentwicklern und Projektmanagern im Allgemeinen akzeptiert, da Fehler ein einkalkulierter Faktor bei Softwareentwicklungsprojekten sind; und Testen ist die bevorzugte Methode, um Fehler zu beheben. Dennoch ist die Frage berechtigt, warum Fehler überhaupt entstehen. Die Antwort auf diese Frage ist wichtig, da die Ursachen von Fehlern auch bei der Entwicklung eines Verfahrens zur Fehlerfindung berücksichtigt werden müssen.

Das Hauptziel beim Programmieren eines Computers ist es, die Maschine

als ein Hilfsmittel zu verwenden, um ein bestimmtes Problem zu lösen. Hierfür gibt es unterschiedliche Motivationen abhängig von der Art des Problems, das es zu lösen gilt: Häufig ist der Mensch nicht in der Lage, das Problem selbst mit der notwendigen Präzision und Geschwindigkeit zu lösen. Ein Beispiel hierfür sind Motorsteuerungen, die im Bereich von Mikrosekunden präzise in die Zündung eingreifen müssen. Die andere Motivation ist, dass die Lösung des Problems eine monotone, anstrengende oder gefährliche Tätigkeit erfordert, die besser von einer Maschine erledigt werden kann, auch wenn der Mensch diese Aufgaben grundsätzlich selbst erledigen könnte.

In jedem Fall bedeutet die Programmierung eines Computers, die Lösung eines Problems vom Menschen auf die Maschine zu übertragen, und Fehler treten aus dem einfachen Grund auf, weil die Art und Weise, wie Menschen und Maschinen Probleme lösen, verschieden ist. Um diese Behauptung zu untermauern, ist eine kurze Erläuterung der Prinzipien des menschlichen Problemlösens notwendig.

Menschliches Problemlösen. Die grundsätzliche Art und Weise, wie Menschen Probleme lösen, wird in einem Spezialgebiet der Psychologie seit ungefähr 1910 untersucht [Dun59]. Ein *Problem* in der psychologischen Bedeutung muss drei Kriterien erfüllen. Es muss ein Ausgangszustand existieren (Startpunkt), es muss ein gewünschter Endzustand existieren (Ziel), und der Weg vom Startpunkt zum Ziel ist unbekannt [May79]. Problemlösen bedeutet, einen Weg vom Startpunkt zum Ziel durch kognitive Prozesse zu finden. Das Ziel kann je nach Situation mehr oder weniger präzise beschrieben sein.

Das menschliche Problemlösen ist ein recht komplexer Prozess, der heute noch nicht vollständig entschlüsselt ist. Der generelle Problemlösungsprozess kann als Wechselspiel zweier kooperierender Subprozesse aufgefasst werden, die als *Verstehen* und *Suchen* bezeichnet werden. Der Prozess des Verstehens erzeugt im Gehirn eine interne Repräsentation eines Problemzustandes (Ausgangszustand, Endzustand oder ein Zwischenzustand) während der Suchprozess eine Lösung oder einen Teilschritt der Lösung generiert. Abhängig vom Problem und von der Person kann sich der tatsächliche Ablauf des Verstehens und Suchens unterscheiden. Abhängig von der Art, wie die einzelnen Teilschritte der Lösung gefunden werden, unterscheidet man zwei Arten der Lösungssuche: Das *analytische Problemlösen* bedeutet, dass neben dem Ausgangszustand und dem Ziel auch alle anwendbaren Transformationsregeln bereits bekannt sind. Ein typisches Beispiel ist die Lösung einer Gleichung mit einer Variablen. Die algebraischen Transformationsregeln sind in diesem Fall fest definiert. Die intellektuelle Herausforderung besteht darin, die relevanten Regeln auszuwählen und in der richtigen Reihenfolge anzuwenden. Das *synthetische Problemlösen* wird immer dann verwendet, wenn die Transformationsregeln unbekannt oder vom Problembereich entkoppelt sind. (Letzteres bedeutet, dass die Regeln zwar eigentlich bekannt sind, diese aber an ein an-

deres Problem *funktional gebunden* sind, so dass das Gehirn die Anwendung der Regel nicht in Erwägung zieht.) Die Regeln, die für die Problemlösung notwendig sind, müssen in diesem Fall zunächst *synthetisiert* werden. Klassische Knobelaufgaben sind typische Beispiele für synthetisches Problemlösen: das Ziel ist meist klar, die Regeln sind jedoch unbekannt [DS96].

Es ist immer noch umstritten, ob „produktives Denken“ und „Kreativität“ wirklich existieren, d.h., ob Gedanken, die es noch nie gegeben hat, im menschlichen Gehirn neu „generiert“ werden. Mit den Worten des Problemlösens stellt sich hier also die Frage, ob der Prozess der „kreativen“ Lösungsfindung eine andere Qualität besitzt, oder ob es sich nur um eine unbewusste und komplexe Variante des synthetischen Problemlösens handelt. Die Gegner der Kreativitäts-These nehmen an, dass Menschen täglich hunderte von Problemen lösen – von der Geburt bis zum Tod – und dabei Regeln erlernen. „Kreativität“ und Intelligenz basieren nach ihrer Auffassung auf einer günstigen Kombination von diesen Regeln während des synthetischen Problemlösens, das letzten Endes auf komplexem Regelwissen beruht [Roh88, AAH83].

Maschinelles Problemlösen. Verglichen mit dem Menschen arbeiten Computer äußerst zuverlässig und erledigen die Aufgaben, für die sie programmiert wurden, mit höchster Präzision. Deshalb traut man zum Beispiel Computern in Kassen und Taschenrechnern: niemand käme auf die Idee, die Ergebnisse zu prüfen, die von diesen Maschinen errechnet wurden. Computer arbeiten auf der Basis von Algorithmen, die einfach abgearbeitet werden, um ein Problem zu lösen. Der Vorteil dieses Ansatzes liegt darin, dass der Lösungsprozess deterministisch und wiederholbar ist. So gesehen sind Kassen und Taschenrechner perfekte Anwendungsgebiete für Computer.

Auf der anderen Seite können Computer ihre Algorithmen jedoch nicht adaptieren, wenn sich der Kontext des Problems ändert. Aus diesem Grund fliegen beispielsweise Flugzeuge nicht vollständig autonom, da die Flugsteuerungssoftware nicht weiß, was in Situationen zu tun ist, für die sie nicht programmiert wurde. Computer sind zwar vergleichsweise perfekt in Präzision und Zuverlässigkeit bei der Bearbeitung ihres Problems, ihnen fehlt jedoch die grundsätzliche Fähigkeit hinsichtlich der oben beschriebenen Problemlösungsprozesse *Verstehen* und *Suchen*, die der Mensch anwendet.

Zwangsläufig lösen Computer und Menschen Probleme auf unterschiedliche Art und Weise. Beide Wege haben klare Vor- und Nachteile. Computer lösen Aufgaben, für die Menschen sie programmiert haben, verstehen jedoch niemals, *warum* sie es tun. Das übergeordnete Problem ist nur dem Menschen bekannt. Die Flugsteuerungssoftware „weiß“ nicht, dass sie ein Flugzeug steuern muss und dass sie damit die Verantwortung für das Leben der Passagiere, die Fracht und das Flugzeug selbst hat. Aus dieser Perspektive ist Software eine präzise Beschreibung *wie*, jedoch nicht *warum* eine Aufgabe gelöst werden soll.

Die Tatsache, dass Menschen kognitive Mechanismen zum Problemlösen einsetzen, während Computer algorithmische Prozeduren abarbeiten, führt zu zwei zentralen Quellen für Softwarefehler, die im Folgenden als Spezifikationsproblem und Implementierungsproblem bezeichnet werden.

Das Spezifikationsproblem. Die Behauptung, dass die Spezifikation eines Problems eine fundamentale, jedoch auch eine „natürliche“ Quelle von Fehlern ist, lässt sich leicht anhand des Beispiels „Autofahren“ beweisen: Lernt ein Mensch Auto zu fahren, lernt er/sie zunächst die Grundkonzepte wie das Lenken, Gasgeben, Bremsen, Schalten usw. Schon nach ein paar Übungen verinnerlicht das menschliche Gehirn unbewusst das Lösungsschema für das Problem „Autofahren“ basierend auf einer Reihe von Transformationsregeln. Tritt nun eine ungewöhnliche Situation – wie zum Beispiel ein geplatzter Reifen – ein, die bisher nicht trainiert wurde, so setzt automatisch das synthetische Problemlösen ein. Nahezu jeder Fahrer wird versuchen, das Auto sofort zu stoppen: Der Fahrer *synthetisiert* eine Regel, da er mit dieser Art von Problem bislang noch nicht konfrontiert war. Die unbewusste Synthese, die im Gehirn abläuft, könnte die folgende sein: *Das Auto verhält sich nicht normal.* → *Die Geschwindigkeit beizubehalten könnte gefährlich sein.* → *Geschwindigkeit reduzieren.* → *Bremsen reduziert die Geschwindigkeit.* → *Vollbremsung, um schnellstmöglich zu bremsen.* Dieser Prozess erfordert sowohl den Prozess Verstehen („Das Auto verhält sich nicht normal“) als auch Suche („Bremsen reduziert die Geschwindigkeit“). Ist der Fahrer erfahren, kann er weitere Regeln entsprechend der Situation hinzuziehen, während andere synthetisierte Regeln bereits zu seinem Erfahrungsschatz zählen, also nicht mehr synthetisiert werden müssen: *Das Auto verhält sich nicht normal.* → *Es ist glatt.* → *Vorsichtig Bremsen, ohne dass die Räder blockieren.*

Mit andern Worten ist die Regelsynthese ein komplexer Prozess, der in seiner Perfektion eine besondere Fähigkeit des Menschen ist. Tritt das Problem des geplatzten Reifens auf, gibt es noch keine vorgefertigte Regel im Gehirn des Fahrers, die sofort angewendet werden kann. Verallgemeinert bedeutet dies, dass Menschen meist synthetisch denken. Sie entwickeln keine universell vorgefertigten Lösungen, bevor ein Problem entsteht. Tritt ein Problem allerdings wiederholt auf, greift das Gehirn auf die bereits gefundene Lösung schneller und souveräner zurück.

Soll ein Computer ein Fahrzeug autonom steuern, muss es selbstverständlich zunächst dazu programmiert werden, Straßen, Verkehrszeichen, Autos u.ä. zu erkennen sowie Lenkwinkel, Bremsdruck, Motormoment einstellen zu können. Seine programmierte Aufgabe ist es, die Aktuatoren in Abhängigkeit von der Analyse der Umgebungsdaten zu beeinflussen. Der Fall eines geplatzten Reifens ist von einem Computer nur dann beherrschbar, wenn er hierfür bereits programmiert ist. Der Computer ist ansonsten nicht in der Lage eine Schlussfolgerung wie *Der Reifen ist geplatzt.* → *Geschwindigkeit reduzieren.*

zu deduzieren, da er die übergeordnete Aufgabe und die drohende Gefahr nicht versteht. Wenn Computer komplexe Probleme lösen müssen, erfordert dies immer die Existenz eines abgeschlossenen Algorithmus, der eine perfekte Lösung für alle möglichen vorstellbaren Problemvarianten liefert. Diese Algorithmen müssen jedoch von Menschen entwickelt werden, und für Menschen ist es – wie oben erläutert – vollkommen unnatürlich über Probleme in dieser Detaillierung und Abgeschlossenheit im Voraus nachzudenken. Der Algorithmus einer autonomen Fahrzeugsteuerung muss die Behandlung von geplatzten Reifen, Glatteis, Kindern, die plötzlich auf die Straße springen und alle anderen möglichen Situationen bereits beinhalten. Dies ist der Grund, warum autonome Fahrzeugsteuerungen äußerst schwierig zu implementieren sind. In Tabelle 1.1 sind die Unterschiede zwischen menschlichem und maschinellem Problemlösen noch einmal zusammengefasst.

Problemlösen ...	... durch Menschen	... durch Computer
vor Eintreten des Problems	Kontextwissen und Regeln zu Teil- oder verwandten Problemen	vollständiger Algorithmus für alle möglichen Probleminstanzen
bei Eintreten des Problems	Problemlösen basierend auf den bekannten Regeln	Anwendung des Algorithmus

Tabelle 1.1: Menschliches und maschinelles Problemlösen

Das Implementierungsproblem. Häufig treten Fehler in Programmen auf, obwohl das zu lösende Problem klar und präzise spezifiziert ist. Das einfachste Beispiel hierfür ist die aufsteigende, alphabetische Sortierung einer Liste von Worten mit beliebiger Länge (ohne Berücksichtigung von Groß-/Kleinschreibung). Ein Mensch kann diese Aufgabe ohne Probleme bewältigen: die alphabetische Ordnung erlernen Kinder im Alter von etwa 7-8 Jahren. Die Spezifikation ist eindeutig und unmissverständlich, dennoch enthalten Softwareimplementierungen dieses Problems oft Fehler. Software wird von Menschen programmiert. Die gewünschte Funktionalität muss zu diesem Zweck in einer passenden, formalen, ausführbaren Programmiersprache ausgedrückt werden. Für niemanden ist eine Programmiersprache eine „Muttersprache“, und es gibt keine menschliche Sprache, die mit einer Programmiersprache verwandt ist. So gesehen ist das Entwickeln von Software mit dem Sprechen in einer Fremdsprache vergleichbar, was das Auftreten von Fehlern erklärt.

Mit der Erfahrung im Umgang mit einer Programmiersprache wächst die Sicherheit und Souveränität in der Anwendung. Ein erfahrener Programmie-

rer erkennt einen Sortieralgorithmus durch bloßes Hinsehen. Programmiersprachen stellen die Verbindung zwischen dem menschlichen Denken und der maschinellen Ausführung dar. Die Entwicklung der Programmiersprachen in den letzten 20 Jahren zeigt, wie wichtig Programmiersprachen als Kommunikationsmittel sind. Moderne Hochsprachen verfolgen immer mehr das Ziel, die menschliche Art zu denken mit einer effizienten Ausführungssemantik zu kombinieren.

1.2 Über die Popularität des Tests

Die Entwicklung komplexer Software ist ohne Anwendung von Qualitätssicherungsmaßnahmen, zu denen der Test gehört, praktisch nicht möglich. Obwohl der Test das wohl bekannteste und am häufigsten eingesetzte Verfahren zur Qualitätssicherung ist, gibt es viele andere Methoden, die für die Entwicklung hochwertiger Software angewendet werden müssen. Während der Test ein analytisches Verfahren zur Verbesserung der *funktionalen Korrektheit* von Software ist, dienen andere Methoden der Verbesserung von Qualitäts-Kriterien wie Abstraktion, Effizienz, Wiederverwendbarkeit, Lesbarkeit, Robustheit, Anwendbarkeit usw. Ohne im Kontext dieser Arbeit auf andere Methoden näher einzugehen, sei dennoch darauf hingewiesen, dass eine qualitativ hochwertige Software nur durch die gezielte Kombination unterschiedlicher Verfahren möglich ist. Selbst für die zentrale Aufgabe des Tests – die Verbesserung der funktionalen Korrektheit – gibt es diverse alternative Verfahren wie Model Checking, Korrektheitsbeweise, formale Programmsynthese, Erreichbarkeitsanalysen sowie informelle Verfahren wie Reviews, Inspektionen und Walkthroughs. Viele dieser Verfahren liefern deutlich präzisere und aussagekräftigere Ergebnisse als der Test. Trotzdem hat bislang aus den im Folgenden aufgeführten Gründen keines der Verfahren die Popularität des Test erreicht.

Die Idee des Tests ist sehr einfach: Wenn ein System im weitesten Sinne ausführbar ist, dann wird es während des Testens ausgeführt, um zu beobachten, ob es sich den Erwartungen gemäß verhält. Ein Test stellt ein Experiment dar, dessen Ergebnis im Voraus nicht bekannt ist.

Wie bereits im Abschnitt 1.1 erläutert, treten Fehler in komplexen Systemen unter anderem dadurch auf, dass der Programmierer gezwungen ist, über die gesamte Komplexität aller möglichen Instanzen und Varianten eines Problems im Voraus nachzudenken, was für Menschen keine natürliche Herangehensweise ist. Beim Testen kehrt der Entwicklungsprozess zurück zur natürlichen Denkweise des Menschen: Während des Testens wird das Verhalten eines Systems mit Hilfe von einigen wenigen individuellen Probleminstanzen – den Testfällen – untersucht. Für jeden Testfall kann ein Tester leicht das gewünschte Verhalten mit dem Verhalten des Systems als zwei Lösungen

derselben Probleminstanz vergleichen. Dieser Vorgang ist vom Gehirn leicht beherrschbar und bewirkt, dass das Testen ein sehr natürlicher Analyseprozess ist, der nicht trainiert werden muss.

Der Test betrachtet ein System also nicht als Ganzes. Stattdessen wird eine Stichprobe von ausgewählten Testdaten verwendet, mit denen das Systemverhalten evaluiert wird. Offensichtlich hängt die Anzahl der aufgedeckten Fehler und damit die Qualität des Tests von der Wahl der Testfälle ab. Aus diesem Grund ist die Selektion von Testfälle Hauptschwerpunkt der meisten Testmethoden.

Neben der einfachen Erlernbarkeit und Anwendbarkeit ist ein wesentlicher Vorteil des Testens, dass das System in seiner realen Einsatzumgebung geprüft werden kann [Gri95]. Wechselwirkungen, die zur Spezifikationszeit nicht bedacht wurden, können nur hier aufgedeckt werden: Einflüsse von Stackgrößen, Cache-Optimierungen, Memory-Alignments, Laufzeitumgebung und Betriebssystem sind ohne den Test der Software in der realen Einsatzumgebung nicht analysierbar. In diesem Punkt ist der Test den anderen Methoden überlegen.

Jedes ausführbare System kann (im einfachsten Sinne) ohne weiteres getestet werden. Testen benötigt keine speziellen Vorbereitungsarbeiten, die durchgeführt werden müssen (mit Ausnahme der Bereitstellung einer angemessenen Testumgebung), oder Formalismen, deren Verwendung für die Methode obligatorisch ist. Dieser Punkt ist in der Praxis ein entscheidender Pluspunkt für den Test. Viele andere, insbesondere formale Verfahren erfordern meist eine formale Systembeschreibung in spezifischen formalen Spezifikationssprachen und stoßen in der Praxis deshalb oft auf Ablehnung. Testen hingegen ist für jedes ausführbare System – auch für Nicht-Software-Systeme – anwendbar und erfordert keinerlei formale Systembeschreibung.

1.3 Nachteile des Tests

Der Hauptnachteil des Tests ist die zu Grunde liegende Pragmatik. In den meisten Fällen erlaubt der Test keinerlei verbindliche Aussage über die Qualität des Systems. Selbst beim Einsatz statistischer Testverfahren kann nur teilweise eine praktisch nützliche Aussage über die Fehlerwahrscheinlichkeit angegeben werden. Nach der Testdurchführung enthalten Systeme für gewöhnlich noch Fehler, die auf Grund des Strichproben-Ansatzes beim Testen nicht gefunden werden konnten.

In vielen praktischen Fällen wird der Test sehr unsystematisch und ineffizient durchgeführt. Häufig handelt es sich beim Testvorgehen eher um die Durchführung von Experimenten als um einen wohlgeplanten, gezielt durchgeführten Schritt der Software-Produktentwicklung. Die Konsequenz sind fehlerhafte Programme, die sich als sehr kostspielig herausstellen können, falls

sie wirtschaftliche oder gesundheitliche Schäden verursachen [IPL99]. Andererseits ist das Testen ein sehr altes Verfahren im Vergleich zu anderen Bereichen des Software-Engineerings. Millionen von Tests sind bereits durchgeführt worden, und Hunderte von Forschern haben sich mit der Entwicklung und Verbesserung von Testverfahren beschäftigt. In diesem Abschnitt soll diskutiert werden, wie es zu dem scheinbaren Widerspruch zwischen der langjährigen Erfahrung und Forschung einerseits und der gegenwärtigen Testpraxis andererseits kommt.

Testen bedeutet Raten. Testen ist ein Stichprobenverfahren. Der Test ist nicht in der Lage zu beweisen, dass ein System *keine* Fehler mehr enthält. Er kann lediglich nachweisen, *dass* ein System Fehler enthält. Diese Tatsache inspirierte Dijkstra bereits 1972 zu seiner berühmten Äußerung: „Testing can only reveal the presence of errors, never their absence.“ [Dij72].

Genau genommen kann der Test mit dem bloßen Raten verglichen werden. Nachdem Tausende von Testfällen untersucht wurden und kein Fehler gefunden wurde, kann es theoretisch immer noch sein, dass alle verbleibenden Fälle fehlerhaft sind. Aus theoretischer Sicht ist der Test deshalb ein sehr schwaches Verfahren, vor allem weil die Größe der Stichprobe in der Regel nicht einmal für statistische Analysen ausreicht. In der Praxis führt diese Tatsache dazu, dass die Wichtigkeit der Testfallauswahl unterschätzt wird. Der Test besteht oft in der Anwendung eines spezialisierten Testtools, dem das Hauptaugenmerk gilt, ohne die Auswahl der relevanten Testfälle systematisch durchzuführen. Hinzu kommt, dass es keine Abbruchbedingungen für den Test gibt, die sich aus der erreichten Softwarequalität ableiten lassen. Zwangsläufig wird der Testaufwand deshalb oft als willkürlich empfunden, und der Testabbruch definiert sich über Zeit- und Budgetgrenzen.

Die Grenzen des menschlichen Gehirns. Der einzige Zweck des Tests ist es, Fehler zu finden. Gäbe es keine Fehler, wären auch keine Tests notwendig. Fehler sind jedoch auf Grund der Komplexität der meisten Systeme unvermeidlich, weil das menschliche Gehirn die Komplexität nicht beherrscht (vgl. Abschn. 1.1). Durch das Stichprobenverfahren reduziert sich diese Komplexität zwar wieder auf ein Maß, das vom Menschen beherrschbar ist; dennoch bleibt das Komplexitätsproblem während der Festlegung der relevanten Testfälle erhalten, da hierbei das System als Ganzes betrachtet werden muss. Aus diesem Grund ist die Testfallauswahl auch der wichtigste Schritt für den Test und Hauptschwerpunkt der meisten Testmethoden.

Testen ist destruktiv. Testen ist ein „ungeliebtes Kind“ der Softwareentwickler, da es destruktiv ist: Ein gründlicher Test deckt Fehler auf, und Fehler bedeuten, dass während der Konstruktion des Systems etwas falsch gemacht wurde, das jetzt einen zusätzlichen Aufwand erfordert. Liegt der Fehler in der

Verantwortung eines einzelnen Entwicklers oder einer Gruppe von Entwicklern wächst die Gefahr der sozialen Spannungen zwischen Entwicklern und Testern.

Testen erfordert einen enormen zusätzlichen Aufwand, der teilweise sogar höher als der Konstruktionsaufwand ist, aber nichts zu den funktionalen Fähigkeiten eines Systems beiträgt. Der kommerzielle Erfolg eines Systems und das öffentliche Interesse wird aber mehr durch funktionale als durch qualitative Faktoren eines Systems bestimmt. Die Qualität spielt erst dann eine vordergründige Rolle, wenn sie unter ein gewisses Toleranzmaß fällt. Testen findet dadurch kaum Anerkennung und ist zwangsläufig eine relativ unbeliebte Aufgabe; auch innerhalb von Projekten ist diese Rollenverteilung häufig zu beobachten.

Die hauptsächliche Arbeit beim Testen entsteht nach klassischen Vorgehensmodellen, wenn die eigentliche Systemkonstruktion bereits abgeschlossen ist. Gerät ein Entwicklungsprojekt unter Zeitdruck, ist der Test leider der „natürliche Kandidat“ für Kürzungen, um die Projekt-Deadlines und die zur Verfügung stehenden Ressourcen einzuhalten.

Zusätzlich wird das Testen paradoxerweise zu einer „destruktiven“ Tätigkeit, wenn das System eine gewisse Qualität erreicht hat und kaum noch Fehler auftreten. In diesem Fall wächst das intuitive Vertrauen in das System, und die Notwendigkeit, die Tests fortzusetzen, lässt sich immer schlechter motivieren. Der Test läuft hier Gefahr als „Ressourcenkiller“ abgestempelt zu werden, dem nur durch eine sorgfältige Testplanung und ein klar definiertes Abbruchkriterium begegnet werden kann.

1.4 Ein einfaches Testmodell

Im folgenden Abschnitt wird ein einfaches Modell des Tests entwickelt, das zur Abgrenzung des Testbegriffs und als Grundlage der nachfolgenden Diskussionen sehr nützlich ist. In den vergangenen 20 Jahren sind bereits mehrere Testtheorien entwickelt worden (vgl. [GG75, WO80, Gou83, HT90]). Diese dienen jedoch vorrangig als Basis formaler Beweise, mit denen die Notwendigkeit von Tests gerechtfertigt werden soll. Diese Theorien sind aus diesem Grund nur aus theoretischer Sicht von Interesse. Mit dem folgenden Testmodell soll vielmehr versucht werden, ein einfaches, allgemeines und abstraktes Modell für den Test zu definieren, das sich an der Testpraxis orientiert und den essenziellen Kern des Tests herausarbeitet. Dabei ist vor allem interessant, welche Kriterien die Qualität eines Tests beeinflussen und wie man diese Qualität objektiv bewerten kann. Es sei jedoch an dieser Stelle betont, dass das Testmodell die Qualität des Tests nicht verbessern kann; es kann lediglich zum besseren Verständnis beitragen und damit ggf. das Vorgehen beim Test verbessern.

Testobjekte und Testorakel

Jeder Test hat das Ziel, ein System auf seine Korrektheit zu prüfen. Aus der Sicht des Tests wird ein solches System als *Testobjekt* bezeichnet. Ein Testobjekt muss bestimmte Eigenschaften erfüllen, um testbar im allgemeinsten Sinne zu sein: Das Testobjekt muss zunächst eine definierte Eingabeschnittstelle $\mathfrak{I}$ sowie eine definierte Ausgabeschnittstelle $\mathfrak{O}$ besitzen. Die beiden Menge $\mathfrak{I}$ und $\mathfrak{O}$ werden auch *Input-Domain* bzw. *Output-Domain* genannt. Jedes Testobjekt liefert für einen gegebenen Eingabewert $i \in \mathfrak{I}$ einen entsprechenden Ausgabewert $o \in \mathfrak{O}$. Ein Eingabewert i kann hierbei nicht nur ein elementares Datum im Sinne von Programmiersprachen, sondern auch – vor allem im Zusammenhang mit Realzeitsystemen – ein komplexes Szenario sein, das über eine definierte Zeitspanne das Testobjekt stimuliert. Entsprechendes gilt auch für den Ausgangswert o. Formal wird ein Testobjekt als eine totale, berechenbare Funktion $s : \mathfrak{I} \rightarrow \mathfrak{O}$ charakterisiert. Die drei wesentlichen Einschränkungen dieser Charakterisierung (1. Berechenbarkeit, 2. Totalität und 3. Determinismus) werden im Folgenden begründet.

Die Berechnung des Ausgabewertes für einen gegebenen Eingabewert muss nach endlicher Zeit terminieren. Testen ist ein praktischer Prozess der Korrektheitsprüfung eines Testobjekts. Aus diesem Grund darf er nicht unendlich lange dauern. Die Endlichkeit der Testdurchführungszeit wird durch die *Berechenbarkeitseigenschaft* von s ausgedrückt, auch wenn dieses Kriterium nur die theoretische Randbedingung bezüglich der Testeffizienz ist. In der Praxis ist die zumutbare Dauer der Testdurchführung weit stärker eingeschränkt.

Die Forderung, dass Testobjekte *total* sind, scheint auf den ersten Blick eine starke Einschränkung zu sein. Genauer betrachtet, werden partielle Systeme beim Test jedoch immer „totalisiert“: Läuft eine Berechnung also beispielsweise in eine Endlosschleife, wird das Testobjekt beim Testen nach endlicher Zeit mit dem speziellen Ausgabewert „Überschreitung der zulässigen Berechnungszeit“ abgebrochen. Die Berechnung ist aus der Sicht des Testobjekts damit total ($s(i) = o_{timeout}$ mit $o_{timeout} \in \mathfrak{O}$). Es gibt keine Möglichkeit, Ausgaben wie „Endlosrekursion“ zu erhalten – jedenfalls nicht durch den Test. Goodenough und Gerhart [GG75], Weyuker und Ostrand [WO80] haben Testobjekte („Programme“) als partielle Funktionen mit einer Spezialbehandlung der Werte definiert, an denen das Programm undefiniert ist. Die obige Diskussion zeigt, dass diese Spezialbehandlung nicht notwendig ist. Echte „Undefiniertheit“ kann vom Test nicht behandelt werden, und Undefiniertheit im Sinne des Tests bedeutet den Abbruch der Ausführung mit einem speziellen $o_{timeout}$-Wert. Dieser Ansatz wurde auch von Hamlet und Taylor [HT90] vorgeschlagen.

Testobjekte wurden als (totale) *Funktionen* anstelle von Relationen definiert, d.h., für jeden Eingabewert i liefert ein Testobjekt s genau einen eindeutigen Ausgabewert $o = s(i)$. Diese *Determinismus*-Eigenschaft wur-

de auch von anderen Testmodellen angenommen, sie ist allerdings problematisch, da sie voraussetzt, dass bei der Betrachtung eines Testobjekts die Eingabeschnittstelle vollständig abgedeckt ist: Betrachtet man beispielsweise die Funktionalität eines Taschenrechners, so wird $\mathfrak{I}$ Aspekte wie die Batteriekapazität, die Luftfeuchtigkeit und die Umgebungstemperatur in der Regel nicht enthalten, obwohl diese die Funktionalität erheblich beeinflussen können. Die Wahl eines Inputs $i \in \mathfrak{I}$ liefert demnach kein deterministisches Ergebnis. Ein weiteres Beispiel sind Race Conditions bei nebenläufigen Prozessen: Wird ein Prozess getestet, kann es Wechselwirkungen mit dem anderen Prozess geben, die teilweise nicht offensichtlich sind und somit beim Test nicht berücksichtigt werden.

Die Lösung des Problems liegt in der Einschränkung auf bestimmte Voraussetzungen zu der Systemumgebung: Sind Temperatur, Luftfeuchte und Batteriekapazität bzw. das Verhalten aller nebenläufigen Prozesse für den Test festgelegt, sind die Ergebnisse deterministisch. Dies ist die praktische Lösung des Problems, die im Sinne eines einfachen Testmodells auch hier zu Grunde gelegt wird. Es sei jedoch darauf hingewiesen, dass durch diese Einschränkung die Wechselwirkung zwischen den Umgebungsbedingungen und den Eingabegrößen an anderer Stelle unter dem Stichwort der Robustheit betrachtet werden muss.

Neben dem Testobjekt benötigt jeder Test eine Instanz, die entscheidet, ob ein berechnetes Testergebnis richtig oder falsch ist. Diese Instanz wird gewöhnlich als *Testorakel* oder als *Testreferenz* bezeichnet [Gri95]. Ein Testorakel wird in den meisten Fällen aus der Requirements-Spezifikation abgeleitet, da dieses Dokument das korrekte gewünschte Verhalten des Testobjekts beschreibt. Dies gilt auch für den Strukturtest (White-Box-Test), da das Testorakel nur für die Bewertung der Testergebnisse und nicht für die Testfallermittlung verantwortlich ist.

Ein Testorakel kann als entscheidbare Relation $r : \mathfrak{I} \leftrightarrow \mathfrak{O}$ charakterisiert werden, für die $\mathsf{dom}\ r = \mathfrak{I}$ gilt. Durch diese Charakterisierung ist festgelegt, dass der Test in der Lage sein muss, für jedes einzelne Wertepaar (i, o) deterministisch zu entscheiden, ob das Ergebnis $o = s(i)$ richtig oder falsch ist. Es kann also keine unscharfen Resultate wie „nahezu richtig“, „beinahe falsch“ oder „unklar, ob wahr oder falsch“ geben.

Weiterhin fordert die Charakterisierung des Testorakels mit der Eigenschaft $\mathsf{dom}\ r = \mathfrak{I}$, dass das Testorakel *fair* sein muss: Zu jedem Eingabewert muss das Orakel mindestens einen Ausgabewert akzeptieren. Ohne diese Einschränkung wäre bei $\mathsf{dom}\ r \subset \mathfrak{I}$ *jedes* Testobjekt per Definition fehlerhaft, was nicht der Idee des Tests entspricht.

Die dritte charakterisierende Eigenschaft eines Testorakels ist die *Entscheidbarkeit*. Hier gilt das Gleiche wie bei der Charakterisierung eines Testobjekts: Testen ist ein endlicher Prozess und eine Entscheidung, ob ein Ergebnis

richtig oder falsch ist, muss in jedem Fall terminieren.

In Abbildung 1.1 sind theoretische Randfälle von Testobjekten und Testorakeln dargestellt, die nach obigen Definitionen nicht testbar sind.

Beispiel 1. Das Testorakel fordert, dass das Testobjekt an einem bestimmten Eingabewert a nicht terminiert (Endlosschleife). In diesem Fall kann es kein Testobjekt geben, das diese Forderung erfüllt, da jedes Testobjekt nach obiger Charakterisierung terminiert. Nur ein Testobjekt, das für a fehlerhaft ist (also terminiert), kann getestet werden. Aus diesem Grund ist die Korrektheit an der Stelle a mit dem Test nur halbentscheidbar.
Beispiel 2. Das Testorakel fordert, dass das Testobjekt an der Stelle a ein Ergebnis b nach endlicher, aber beliebiger Zeit liefert. Auch in diesem Fall ist die Korrektheit an der Stelle a nur entscheidbar, wenn das Testobjekt tatsächlich terminiert.

Abbildung 1.1: Beispiele nicht testbarer Testobjekte und Testorakel

Mit der Kenntnis eines Testobjekts und eines Testorakels kann die Menge der *Fehler* als $\mathfrak{F} := \{i \mid (i, s(i)) \notin r\}$ definiert werden. $\mathfrak{F}$ entspricht der Menge aller Eingangswerte, an denen die zugehörigen, von s gelieferten Ausgangswerte vom Testorakel als *falsch* erkannt wurden. Von entscheidender Bedeutung für den Test ist die Tatsache, dass zwar für jedes individuelle i die Eigenschaft $i \in \mathfrak{F}$ entscheidbar ist (wegen der Eigenschaften von s und r), die Menge $\mathfrak{F}$ im Allgemeinen aber nicht algorithmisch aufgezählt werden kann. Anders ausgedrückt: Es gibt keinen Algorithmus, der alle Fehler in einem Testobjekt bei Kenntnis von s und r automatisch finden kann.

Aus diesem Grund ist es notwendig, Testobjekte zu *testen*. Der Test ist ein Stichprobenverfahren, bei dem aus der Gesamtmenge $\mathfrak{I}$ eine *endliche* Teilmenge I ausgewählt wird. Für jeden Wert $i \in I$ wird beim Test die Eigenschaft $i \in \mathfrak{F}$ untersucht. Dieser Stichprobenansatz hat den wesentlichen Vorteil, dass die Menge der so aufgedeckten Fehler $I \cap \mathfrak{F}$ immer berechenbar ist (wegen der Endlichkeit von I ist). Der Nachteil ist jedoch, dass nur solche Fehler aufgedeckt werden können, die in I enthalten sind. Dies macht deutlich, dass die Qualität des Tests entscheidend von der Auswahl der Stichprobe I beeinflusst wird. Die *Fehleraufdeckungsrate* R_I ist das Verhältnis

$$R_I := \begin{cases} \frac{\mathsf{card}\ (I \cap \mathfrak{F})}{\mathsf{card}\ \mathfrak{F}} & \text{falls } \mathfrak{F} \neq \emptyset \\ 1 & \text{sonst} \end{cases}$$

zwischen den mit I aufgedeckten Fehlern und den im Testobjekt enthaltenen Fehlern. Bei ungünstigster Auswahl von I gilt $R_I = 0$, d.h., es sind keine der im Testobjekt enthaltenen Fehler durch die Stichprobe I gefunden worden.

Der günstigste Fall ist $R_I = 1$. In diesem Fall sind alle enthaltenen Fehler aufgedeckt.

Anmerkung: Das Maß R_I ist im Allgemeinen nicht berechenbar, da die Gesamtanzahl der Fehler $\mathsf{card}\ \mathfrak{F}$ aus den oben genannten Gründen nicht berechnet werden kann. R_I kann also nur durch Schätzung von $\mathsf{card}\ \mathfrak{F}$ approximiert werden. Der Schätzung von $\mathsf{card}\ \mathfrak{F}$ können statistische Abschätzungen oder Erfahrungswerte aus vergleichbaren Entwicklungsprojekten zu Grunde liegen.

Fehlerursachen und Fehlerkosten

Der relative Anteil der mit einer Stichprobe I aufgedeckten Fehler R_I stellt bereits ein einfaches Qualitätsmaß dar. Für praktische Tests ist dieses Maß jedoch im Allgemeinen nicht ausreichend, um die Güte eines Tests realistisch zu beurteilen, wie das folgende Beispiel zeigt.

Sei $r : \mathbb{N} \leftrightarrow \mathbb{N}$ ein Testorakel, das die Nachfolgerfunktion der natürlichen Zahlen[1] beschreibt und $s : \mathbb{N} \to \mathbb{N}$ ein Testobjekt, das fälschlicherweise die Identitätsfunktion implementiert, so dass demnach $\mathfrak{F} = \mathbb{N}$ gilt. Das Testobjekt s kann auf verschiedene Arten implementiert worden sein. Folgende zwei Alternativen sollen betrachtet werden:

Implementierung A	Implementierung B

`y := x`	`if (x == 0) y := 0`
	`elsif (x == 1) y := 1`
	`elsif (x == 2) y := 2`
	...

In der Implementierung A deckt der Test mit einem einzigen (beliebigen) Wert $i \in \mathbb{N}$ bereits den Fehler des vergessenen „`+1`“ auf. Das Programm mit weiteren Eingaben zu testen macht wenig Sinn, da der einzige Fehler des fehlenden „`+1`“ schon aufgedeckt wurde. Die ideale Testdatenmenge ist deshalb die Menge $I = \{i\}$ mit einem beliebigen Wert $i \in \mathbb{N}$.

Anders in der Implementierung B: Hier ist es notwendig, jeden Wert $i \in \mathbb{N}$ zu testen, da in jeder Zeile des Programms ein anderer Programmierfehler[2] enthalten ist. Die ideale Testdatenmenge ist in diesem Beispiel die Menge $I = \mathbb{N}$.

Interessant ist, dass sich beide Testobjekte aus der Sicht des Tests offensichtlich stark unterscheiden. Die Ursache liegt darin begründet, dass beide Implementierungen, obwohl sie sich dasselbe Testobjekt und dieselbe Fehlermenge $\mathfrak{F} = \mathbb{N}$ teilen, nicht die *Fehlerursache* – im Folgenden auch als *Bug*[3]

[1] Formal: $r = \{(x, x+1) | x \in \mathbb{N}\}$

[2] Hier sei unterstellt, dass bei der Aufdeckung des Fehlers einer Zeile nicht automatisch auf die anderen Fehler geschlossen wird.

[3] Im Englischen wird der Fehler meist als *failure* und die Fehlerursache als *fault* oder *bug* bezeichnet. Der einfacheren Sprechweise wegen werden die Fehlerursachen im weiteren Verlauf der Arbeit als *Bugs* bezeichnet.

bezeichnet – teilen, die das Fehlverhalten hervorruft.

Fehlverhalten tritt immer als Folge von Bugs auf. Der Test hat die Aufgabe, experimentell herauszufinden, welche Bugs im Testobjekt enthalten sind. Zusätzlich wird beim Test aber auch überprüft, welche Bugs *nicht* vorliegen. Da zum Zeitpunkt des Tests das Systemverhalten noch unbekannt ist, müssen zwangsläufig alle *potenziellen* Bugs betrachtet werden, die im Testobjekt eventuell zu einem Fehlverhalten führen könnten. Dies bedeutet, dass ein Test genau genommen experimentell untersucht, welche potenziellen Bugs in einem Testobjekt tatsächlich vorliegen.

Zur Modellierung der potenziellen Bugs wird eine Menge $\mathfrak{B}$ verwendet. Jeder potenzielle Bug $b \in \mathfrak{B}$ wirkt sich auf eine nichtleere Menge $b_\downarrow \subseteq \mathfrak{I}$ von Eingängen aus, die *Wirkungen* von b genannt werden. Umgekehrt wird die einem Eingang $i \in \mathfrak{I}$ zugeordnete Menge der potenziellen Bugs als $i_\uparrow := \{b \in \mathfrak{B} \mid i \in b_\downarrow\}$ notiert. Diese Notation wird auch auf Mengen von Eingaben erweitert: $I_\uparrow := \bigcup_{i \in I} i_\uparrow$.

Wie bereits erwähnt, treten nicht alle potenziellen Bugs in einem Testobjekt tatsächlich auf. Deshalb sind die tatsächlichen Bugs eines Testobjekts s eine Teilmenge von $\mathfrak{B}$, die als $\mathfrak{B}_s$ bezeichnet wird. Die Menge $\mathfrak{B} \setminus \mathfrak{B}_s$ ist dementsprechend die Menge der in s nicht auftretenden Bugs. Ein fehlerhafter Eingangswert i tritt dabei genau dann auf, wenn es mindestens einen in s auftretenden Bug b gibt, der sich auf i auswirkt. Formal bedeutet dies folgende Eigenschaft: $i \in \mathfrak{F} \iff \mathfrak{B}_s \cap i_\uparrow \neq \emptyset$. (Es ist zu beachten, dass bei dieser Charakterisierung der Beziehung zwischen Fehlern und Bugs das Phänomen einander auflösender Doppelfehler im Interesse eines einfachen Modells ausgeschlossen wurde.)

Eine ideale Stichprobe wäre offensichtlich eine Menge I, die sämtliche Bugs überprüft ($I_\uparrow = \mathfrak{B}$), aber kein Experiment „zu viel“ enthält: D.h., es darf für die ideale Stichprobe keine andere Stichprobe I' mit $I'_\uparrow = \mathfrak{B}$ und $\mathsf{card}\ I' < \mathsf{card}\ I$ geben.

Bei praktischen Tests ist dieses Ideal jedoch faktisch nicht erreichbar. Häufig bleiben Bugs beim Test unentdeckt. Ursache hierfür ist, dass bei der Bewertung der Testqualität die *Kosten* eine Rolle spielen, die durch potenzielle Bugs verursacht werden können. Vorrangig sind solche Bugs zu prüfen, deren Kosten besonders hoch sind. In der Praxis können Bugs extrem unterschiedliche Relevanz entsprechend ihrer Auswirkungen haben. So gibt es Bugs, die nahezu vernachlässigbar sind (z.B. falsch programmierte Farben bei Benutzeroberflächen), und solche, die katastrophale Auswirkungen haben (z.B. nicht erkannte Überhitzung eines nuklearen Brennelementes). Die meisten Testtheorien erwähnen das Kostenproblem ohne nähere Behandlung, mit Ausnahme von Tsoukalas, Duran und Ntafos [TDN93], die ein einfaches Kostenmodell definiert haben. Die Kosten eines potenziellen Bugs $b \in \mathfrak{B}$ werden im Folgenden durch einen Wert $c_b \in \mathbb{R}^+$ definiert, deren Summe über alle

$b \in \mathfrak{B}$ gleich 1 sein soll. Die Summe

$$C_I := \sum_{b \in I_\uparrow} c_b$$

beschreibt die Kosten der Bugs, die durch die Stichprobe I geprüft wurden und ist damit ein weiteres Maß, mit dem die Testqualität beurteilt werden kann. Interessant an dieser Definition ist vor allem, dass eine Stichprobe bzgl. C_I auch dann eine hohe Qualität haben kann, wenn kein einziger Fehler gefunden wurde. Dies ist durchaus sinnvoll. Gerade in den letzten Testdurchläufen, in denen die Anzahl der im Testobjekt unentdeckt gebliebenen Bugs sehr gering ist, nimmt bei diesem Maß die Testqualität deshalb nicht ab.

Das Maß C_I kann in praktischen Fällen nur geschätzt werden, da die Menge $\mathfrak{B}$ für ein System im Allgemeinen nur geschätzt werden kann. Außerdem existiert für die Angabe der Kostenverteilung kein objektives Maß. Vielmehr obliegt es der Qualitätssicherung, die Kosten abzuschätzen und damit den Schwerpunkt für den Test zu bestimmen. Diese Abschätzung ist in der Praxis intuitiv durchaus üblich, wenn die Schwerpunkte für die Tests geplant werden. Die Kostenverteilung wird aber in der Regel nicht explizit angegeben.

Testaufwand

Die beiden bisher definierten Maße für die Fehleraufdeckung R_I und C_I sind jeweils maximal für den vollständigen Test mit $I = \mathfrak{I}$. Ursache hierfür ist, dass die Maße nur die Qualität aus der Sicht der Fehleraufdeckung, nicht aber den notwendigen Aufwand der Testdurchführung betrachten. In der Praxis stellt der Test immer einen Kompromiss zwischen Fehleraufdeckung und notwendigem Aufwand dar.

Der Testaufwand soll vereinfachend als $E_I := \mathsf{card}\ I$ definiert werden. Diese Definition abstrahiert von der Tatsache, dass der Aufwand einzelner Tests in der Regel nicht für alle $i \in \mathfrak{I}$ gleich groß ist.

Für die zusammenfassende Bewertung der Qualität eines Tests sind immer beide Faktoren – Fehleraufdeckung und Aufwand – gemeinsam zu betrachten. Die Fehleraufdeckung verhält sich monoton steigend zum Aufwand: Werden zu einer gegebenen Stichprobe I weitere Eingaben hinzugefügt, so wird durch die neue Stichprobe I' eine mindestens gleich große Fehleraufdeckung (gemessen mit einem der Maße C_I bzw. R_I) erzielt. Die entscheidende Frage ist hierbei, in welchem Grad die Fehleraufdeckung relativ zum wachsenden Aufwand zunimmt. Dieses Verhältnis hängt einzig von der verwendeten Testmethode ab. In Abbildung 1.2 ist dieser Zusammenhang veranschaulicht.

Das beschriebene Testmodell macht deutlich, dass die Stichprobe I die einzige Größe ist, mit der der Tester Einfluss auf die Fehleraufdeckung und den Aufwand des Tests nehmen kann. Sobald I fest steht, ist der weitere Ablauf

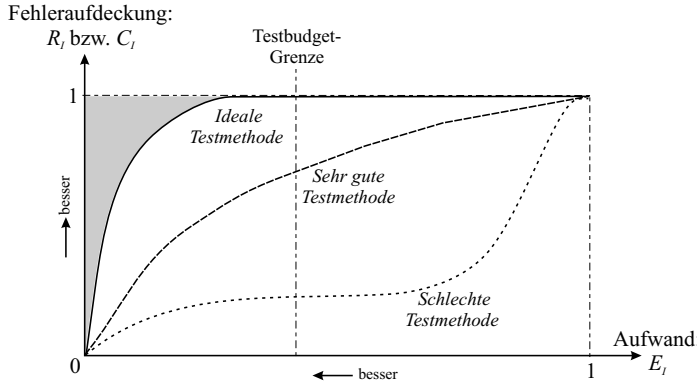

Abbildung 1.2: Maß für die Testqualität

des Tests intellektuell nicht mehr anspruchsvoll. Die nachfolgenden Testschritte (Testdurchführung, Monitoring, Testauswertung usw.) beeinflussen nur die Effizienz der Testumsetzung, nicht aber die Testqualität.

Kapitel 2

Einführung zu TPT

Das in dieser Arbeit vorgestellte Time Partition Testing (kurz: TPT) definiert eine durchgängige Vorgehensweise für den Test des kontinuierlichen Verhaltens eingebetteter Systeme. Das Ziel bei der Entwicklung des Time Partition Testing war es, in jeder Testphase die jeweiligen Anforderungen und Problemfelder aus der gegenwärtigen Praxis zu adressieren.

Im folgenden Kapitel soll zunächst ein kurzer Überblick über das Time Partition Testing anhand eines kurzen Beispiels gegeben werden, bevor in den darauffolgenden Kapiteln detaillierter auf die einzelnen Bereiche eingegangen wird.

2.1 Kontinuierliches Verhalten

Im Rahmen dieser Arbeit wird der Test der speziellen Klasse von Systemverhalten betrachtet, das *kontinuierliches Verhalten* genannt wird. Da es sich hierbei nicht um einen Standardbegriff der Informatik handelt, soll zunächst erläutert werden, was unter kontinuierlichem Verhalten zu verstehen ist.

Grundsätzlich kann für jedes reaktive System das kontinuierliche Verhalten betrachtet werden. Sei $S : I \times Z \times T \rightarrow O \times Z \times T$ ein solches System, wobei I, O und Z die üblichen Eingänge, Ausgänge und internen Zustände des Systems sind. Die Menge T symbolisiert die Zeit (Realzeit oder ein anderes Zeitmodell). Gilt $S(i, z, t) = (o, z', t')$, so bedeutet dies, dass S zum Zeitpunkt t' den Ausgangswert o und den internen Zustand z' berechnet hat, wenn zum Zeitpunkt t der Eingangswert i und der interne Zustand z vorgelegen hat. Die Zeitspanne $t \ldots t'$ ist die Berechnungsdauer von S für einen Schritt.

Bei klassischen Black-Box-Testverfahren wird für einzelne vorgegebene Eingangswerte $i \in I$ der Ausgangswert $o \in O$ ermittelt und ausgewertet. Hat ein System einen für das Verhalten relevanten internen Zustandsraum Z und hängt das Verhalten von der Zeit des Auftretens der Eingabe ab,

müssen diese Faktoren zwangsläufig auch beim Test berücksichtigt werden. Beim klassischen Black-Box-Test wird dies meist dadurch gelöst, dass teilweise auch Zustände als Eingänge aufgefasst werden. Für die Testdurchführung werden die Zustände dann entweder künstlich manipuliert oder das System wird vor der eigentlichen Testdurchführung mit bestimmten Eingangswerten in den gewünschten Zustand gebracht. Aus dieser Sicht bilden die Mengen $\mathfrak{I} = I \times Z \times T$ und $\mathfrak{O} = O \times Z \times T$ die Input- und Output-Domains im Sinne des Testmodells aus Abschnitt 1.4. Hat ein System beispielsweise drei Betriebsmodi, in denen es sich jeweils unterschiedlich verhält, so wird für einen Testfall beschrieben, in welchem Betriebsmodus dieser Testfall durchzuführen ist. Ist die Anzahl interner Zustände gering und haben die Zustände auch aus Black-Box-Sicht eine logische Bedeutung, ist diese Herangehensweise durchaus vertretbar. Hingegen wird sie schier unbrauchbar, wenn die Anzahl der internen Zustände sehr groß wird und jeder einzelne interne Zustand nicht mehr intuitiv verständlich ist. Dies trifft insbesondere für reaktive Systeme zu, da reaktive Systeme meist die Historie einer Interaktion mit der Systemumgebung im internen Zustand abbilden müssen.

Die Problematik wird bereits bei einem sehr einfachen Beispiel deutlich: Es soll ein Delay-Glied getestet werden, das einen einkommenden Wert der Eingangsgröße `p` um eine feste Zeitdauer `dT` verzögert an der Ausgangsgröße `q` ausgeben soll (vgl. Abb. 2.1). Obwohl das Verhalten dieses Systems relativ einfach zu beschreiben ist, ist sein Test mit klassischen Verfahren recht kompliziert, da der interne Zustand immer den kompletten Verlauf des Signals `p` der letzten Zeitspanne `dT` repräsentieren muss.

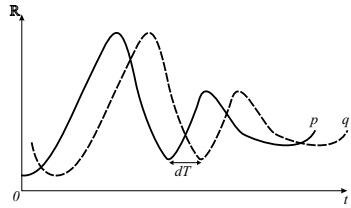

Abbildung 2.1: Kontinuierliches Verhalten eines Delay-Gliedes

Wesentlich einfacher ist hingegen die Sichtweise, dass das System immer über eine gewisse Zeitspanne hinweg mit einem kompletten kontinuierlichen Verlauf von Eingangsdaten $t : T \rightarrow I$ stimuliert wird. Entsprechend wird im selben Zeitfenster der Verlauf der Ausgangswerte beobachtet. Geht man weiterhin davon aus, dass zu einem definierten Anfangszeitpunkt t_0 immer ein definierter interner Anfangszustand z_0 vorliegt, kann das Verhalten über der Zeit vollständig *ohne* Berücksichtigung der internen Zustände – also als reine Black-Box – analysiert werden.

Formal betrachtet heißt das, dass das System S in eine Funktion $\vec{S} : Z \times T \to (T \to I) \to (T \to O)$ überführt wird: Seien der Anfangszustand z_0, der Anfangszeitpunkt t_0 sowie ein Verlauf von Eingangsdaten $\vec{\iota} : T \to I$ gegeben. Durch iterativen Aufruf von S sind dadurch bereits die Folgen $\{o_n\}$, $\{z_n\}$ und $\{t_n\}$ durch $(o_{n+1}, z_{n+1}, t_{n+1}) = S(\vec{\iota}(t_n), z_n, t_n)$ definiert. Mit diesen Folgen lässt sich $\vec{S}$ als $\vec{S}_{(z_0,t_0)}(\vec{\iota})(t_n) = o_n$ definieren.[1]

Diese abstrahierte Verhaltensbeschreibung von S in Form von $\vec{S}$, bei der Verläufe der Ein- und Ausgänge betrachtet werden und die internen Zustände von S (mit Ausnahme des Anfangszustands) irrelevant sind, wird als *kontinuierliches Verhalten* bezeichnet. Unter der Voraussetzung, dass z_0 und t_0 für alle später durchgeführten Tests fix sind, bilden die Mengen $\mathfrak{I} = (T \to I)$ und $\mathfrak{O} = (T \to O)$ die Input- und Output-Domains im Sinne des Testmodells aus Abschnitt 1.4. Das Time Partition Testing ist ein spezialisiertes Testverfahren, das reaktive Systeme aus dieser Sicht betrachtet.

Vor allem für eingebettete Realzeitsysteme ist das kontinuierliche Verhalten eine sehr natürliche Betrachtungsweise. Dies gilt sowohl für Systeme mit harten als auch für solche mit weichen Realzeitanforderungen. Eingebettete Systeme interoperieren mit einer realen Umgebung, z.B. mit physikalischen Prozessen. Hierbei spielt die Zeit immer eine zentrale Rolle. Auch für eingebettete Systeme, die eine Schnittstelle zum Anwender haben, ist das kontinuierliche Verhalten wichtig, da hier Zeitaspekte wie Reaktionszeiten, Anzeigedauern u.ä. Berücksichtigung finden. Wie im nächsten Kapitel ausführlicher diskutiert wird, basiert das kontinuierliche Verhalten auf der realen, dichten Zeit. Systeme mit zeitdiskretem Verhalten sind Grenzfälle für das kontinuierliche Verhalten. Beispiele sind Systemschnittstellen, die bestimmte Protokolle für die Kommunikation mit anderen Komponenten definieren. Für solche Systeme *kann* das kontinuierliche Verhalten betrachtet werden, wenngleich es Testtechniken gibt, die für diese Art von Systemen teilweise besser geeignet sind.

2.2 Anwendung von TPT: Ein Beispiel

Im folgenden Abschnitt wird ein stark vereinfachtes Anwendungsbeispiel von TPT erläutert, um einen groben Überblick über die enthaltenen Konzepte zu geben.

Eine (fiktive) Motorsteuerung hat bei 6500min^{-1} ihren kritischen Arbeitsbereich. Die Motorsteuerung liest permanent den aktuellen Drehzahlwert und die Gaspedalstellung ein und regelt in Abhängigkeit dieser Werte die Einspritzmenge. Der Drehzahlfühler hat bei Drehzahlen über 5000min^{-1} zeitwei-

[1] Diese Charakterisierung von $\vec{S}$ ist für alle Zwischenzeitpunkte ($t \notin \{t_n\}$) unspezifiziert. Die Behandlung der Diskretisierung bei kontinuierlichem Verhalten wird erst im Abschn. 3.5 diskutiert.

se ein spezifisches Schwingungsverhalten, so dass das Drehzahlsignal entsprechend verfälscht an der Motorsteuerung ankommt (vgl. Abb. 2.2a).

Testziel. Ziel des Tests ist es, die Auswirkungen des Drehzahlfehlers auf die Funktionalität der Motorsteuerung zu überprüfen. Um den Test präzise beschreiben und automatisch ablaufen zu lassen, müssen Drehzahlsensor und Fahrer durch einen Testfalltreiber ersetzt werden, der den genauen Ablauf des Tests festlegt (vgl. Abb. 2.2b).

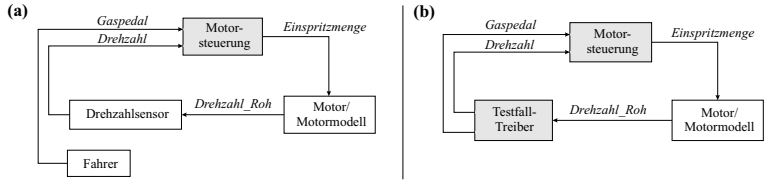

Abbildung 2.2: Kontext der Testdurchführung

Testmodellierung. Zunächst muss beschrieben werden, wie Testfälle ablaufen sollen. Hierzu unterstützt TPT eine Automatennotation, mit der der Gesamtablauf in eine Folge von Phasen zerlegt wird (vgl. Abb. 2.3).

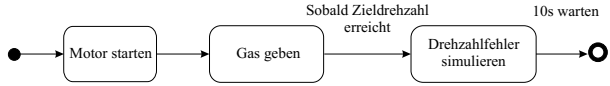

Abbildung 2.3: Phasenablauf in Form eines Automaten

Dieser Ablauf beschreibt, dass jeder Testfall initial zunächst den Motor starten muss, um die Motorsteuerung zu initialisieren. Dann muss solange Gas gegeben werden, bis die gewünschte Drehzahl erreicht ist. Anschließend wird das Gaspedal konstant gehalten und der Drehzahlfehler simuliert. Nach weiteren 10 Sekunden ist der Test beendet.

Dieser Automat beschreibt das Verhalten in den Zuständen sowie die Übergangsbedingungen informell. Jedem Element wird zusätzlich eine formale Definition zugeordnet, damit Testfälle automatisch durchführbar sind. Transitionsbedingungen werden mit einer C-ähnlichen Sprache formalisiert. Die formale Bedingung von „10s warten“ ist beispielsweise `t >= 10` (`t` entspricht der aktuellen, relativen Zeit seit Betreten des letzten Zustandes). Zustände werden ggf. durch Sub-Automaten weiter in Teilphasen zerlegt und auf unterster Ebene durch Gleichungen in C-ähnlicher Sprache formalisiert. Der Zustand

„Gas geben bis Drehzahl erreicht“ ist durch die Gleichung `Gaspedal(t) = 80.0` definiert (`Gaspedal` entspricht der Stellung des Gaspedals in Prozent). Auf diese Weise lässt sich ein Testfall formal präzise beschreiben.

Da *alle* Testfälle für das Testproblem immer nach dem obigen Schema aus Abb. 2.3 ablaufen, soll der Graph für alle Testfälle wiederverwendet werden. Wenn der Graph für alle Testfälle gleich ist, muss es zwangsläufig Unterschiede innerhalb der Definition einzelner Phasen und Übergangsbedingungen geben. Der nächste Schritt ist deshalb, die Elemente des Automaten herauszuarbeiten, die von Testfall zu Testfall variieren. Hinter der Variation steckt die Annahme, dass sich jede Variante hinsichtlich möglicher Fehler im System anders auswirken könnte. Im Beispiel werden die Definitionen von „Sobald Zieldrehzahl erreicht“ und „Drehzahlfehler simulieren“ variiert, d.h., der Test soll bei unterschiedlichen Drehzahlen und mit verschiedenen simulierten Messfehlern erfolgen. Die jeweiligen Varianten werden den Aspekten mit Hilfe der Klassifikationsbaum-Methode [Gro94] zugeordnet (vgl. Abb. 2.4). Anschließend werden aus den $3 \times 3 = 9$ möglichen Kombinationen in der Kombinationstabelle vier Testfälle ausgewählt, von denen angenommen wird, dass sie das Testproblem hinreichend abdecken (vgl. Abb. 2.4).

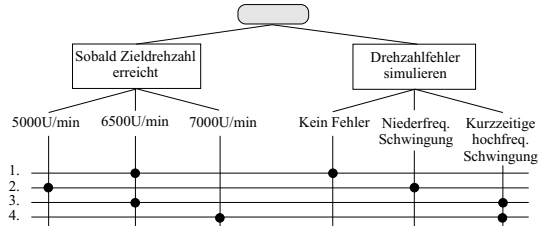

Abbildung 2.4: Klassifikationsbaum mit kombinierten Testfällen

Für den Zustand „Drehzahlfehler simulieren“ und die Transition „Sobald Zieldrehzahl erreicht“ existieren jeweils drei konkrete Varianten der Definition. Für diese Elemente muss demnach jeder Variante eine formale Beschreibung zugeordnet werden, damit die Testfälle automatisch durchführbar sind. Mit dieser formalen Definition sind alle im Klassifikationsbaum kombinierten Testfälle vollständig formal beschrieben und somit ausführbar.

Testdurchführung. Für die Durchführung der vier definierten Testfälle wird eine so genannte *Testengine* benötigt, die jeden Testfall in Wechselwirkung mit der Motorsteuerung ausführen kann. TPT repräsentiert alle definierten Testfälle in einem XML-basierten Zwischencode, der von der Testengine automatisch abgearbeitet wird. Durch die Verwendung eines abstrakten Zwischencodes kann ein und derselbe Testfall auf mehreren Plattformen

ausgeführt werden. Dies ist in der Praxis wünschenswert, wenn ein System zunächst auf dem Entwicklungsrechner und anschließend auf der Zielplattform getestet werden soll. Während der Ausführung werden alle beteiligten Signale (`Gaspedal`, `Drehzahl` usw.) aufgezeichnet, damit sie später ausgewertet werden können.

Testauswertung. Um das Ergebnis der Testdurchfühung automatisiert überprüfen zu können, müssen Eigenschaften spezifiziert werden, die das System erfüllen muss. Im Beispiel ist es unter anderem sinnvoll, die zulässigen Grenzen der Motordrehzahl zu überprüfen. Da diese Analyse jedoch nur in der Phase „Drehzahlfehler simulieren“ erfolgen muss, weil nur in dieser Phase das Verhalten des Motors relevant ist, ist es mit TPT möglich, formale Auswertungseigenschaften direkt an den betroffenen Zustand zu annotieren und die Analyse damit auf den entsprechenden Zeitabschnitt zu beschränken. Für eine genauere Erläuterung des Konzeptes der Testauswertung sei auf Kapitel 7 verwiesen.

Testdokumentation. Für die abschließende Bewertung des Testergebnisses durch den Tester und als Nachweis der durchgeführten Tests wird anschließend eine Testdokumentation generiert (als HTML, PDF o.ä.). Jeder modellierte Testfall kann jederzeit wiederholt durchgeführt und mit den Ergebnissen verglichen werden.

2.3 Testprozess mit TPT

Das einfache Anwendungsszenario aus dem letzten Abschnitt bildet bereits exemplarisch den Prozess ab, der sich aus der Anwendung von TPT ergibt. Dieser Testprozess basiert auf dem allgemeineren Testprozess von [Gri95] und besteht aus den folgenden Schritten (vgl. Abb. 2.5).

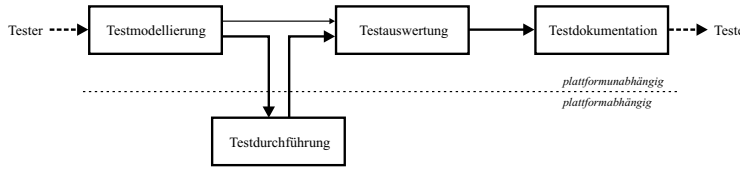

Abbildung 2.5: Schema des Testprozesses

Testmodellierung. Während der Testmodellierung werden alle Testfälle mit Hilfe einer speziellen grafischen, formalen Automatennotation durch den Tester definiert. Für jeden Testfall werden die Zustände und Transitionen solange hierarchisch verfeinert, bis sie auf unterster Ebene formal präzise definiert sind, so dass der Testfall eine ausführbare Semantik hat. Eine Besonderheit von TPT ist, dass alle Testfälle eines Testproblems gemeinsam modelliert werden und sich nur durch Variantenbildung im Detail unterscheiden.

Die Testmodellierung fasst damit die beiden Schritte der Testfalldefinition und Testdatenauswahl aus [Gri95] zusammen, da es bei der Modellierung von Testfällen mit TPT keine sinnvolle Trennung zwischen Testfällen und Testdaten gibt: Testfälle sind zunächst auf oberster Ebene abstrakt, werden aber immer weiter verfeinert, so dass letzten Endes hinter jedem abstrakten Element eines Testfalls eine präzise Beschreibung der zugehörigen Testdaten existiert. Details zur Modellierung von Testfällen werden im Kapitel 4 beschrieben.

Testdurchführung. Die Testdurchführung erfolgt im Anschluss an die Testmodellierung. Für die Testdurchführung wird eine Testengine benötigt, die für die Ausführung der Testfälle auf der jeweiligen Plattform zuständig ist. Die Testdurchführung erfolgt automatisch und zeichnet dabei alle für die spätere Testauswertung relevanten Daten auf. Ein parallel zur Testdurchführung ablaufendes Monitoring des Überdeckungsgrads (vgl. [Gri95]) wird von TPT nicht unterstützt. Details zur Testdurchführung werden im Kapitel 6 beschrieben.

Testauswertung. Für jeden Testfall werden die geforderten zu analysierenden Eigenschaften formal spezifiziert. Diese Eigenschaften bilden gemeinsam gewissermaßen das Testorakel.

Für die Testauswertung werden einerseits die zu analysierenden Eigenschaften und andererseits die relevanten aufgezeichneten Daten aus der Testdurchführung benötigt. Die Testauswertung erfolgt vollautomatisch und ermittelt für jede spezifizierte Eigenschaft, ob sie für den durchgeführten Testfall erfüllt ist oder nicht. Die Testauswertung ist Schwerpunkt des Kapitels 7.

Testdokumentation. Während der Testdokumenation werden alle Ergebnisse des Tests für den Tester lesbar aufbereitet, um einen möglichst schnellen Überblick über den Erfolg oder Misserfolg auch bei einer großen Anzahl durchgeführter Tests zu ermöglichen. Die Erstellung der Testdokumentation erfolgt automatisch und dient auch als Testnachweis. Die Testdokumentation wird in der vorliegenden Arbeit nicht näher erläutert.

Rahmenaktivitäten. In [Gri95] beschriebenen administrativen Rahmenaktivitäten der Testdatenverwaltung, Testplanung und Testorganisation wer-

den von TPT nicht adressiert, wenngleich beispielweise die Testdatenverwaltung durch definierte Beschreibungsformate für Testfälle und Testergebnisse unterstützt wird. Für die Testplanung und Testorganisation legt TPT keine spezifische Vorgehensweise fest.

Kapitel 3

Kontinuierliches Verhalten

In diesem Kapitel werden die semantischen Grundlagen für die Modellierung von Testfällen für das kontinuierliche Verhalten geschaffen. Es wird zunächst diskutiert, welche verschiedenen Ansätze es zur Modellierung von kontinuierlichem Verhalten gibt und welche Vor- bzw. Nachteile diese haben (Abschn. 3.2). Hierfür wird zuvor ein einfaches universelles Zeitmodell entwickelt, das den Vergleich der teilweise sehr unterschiedlichen Ansätze ermöglicht (Abschn. 3.1). Anschließend wird die Theorie der stromverarbeitenden Funktionen zusammen mit der Theorie hybrider Systeme (hybride Automaten) in einer für TPT spezifischen Form eingeführt (Abschn. 3.3 und 3.4). Das entstehende Modell entspricht der semantischen Basis der Modellierungstechniken für TPT, die im darauf folgenden Kapitel 4 vorgestellt werden.

3.1 Modellierung der Zeit

Vielen Arbeiten, die sich mit der Modellierung und Analyse von reaktiven Systemen beschäftigen, liegt ein explizit definiertes oder integriertes Zeitmodell zu Grunde. Je nachdem, welches Ziel mit der Zeitmodellierung verfolgt wird, kann die formale Definition der Zeit sehr unterschiedlich ausfallen.

Bei der Modellierung des kontinuierlichen Verhalten wird die Wechselwirkung eines reaktiven Systems mit seiner Umwelt aus globaler Sicht an der Schnittstelle zwischen System und Umwelt betrachtet. Die Zeit spielt hierbei eine zentrale Rolle: Zum einen müssen Abläufe und Vorgänge *qualitativ* bezüglich des Zeitpunktes ihres Auftretens zu ordnen sein. Dadurch können kausale Zusammenhänge formal erfasst werden, für die nur die Reihenfolge und keine Zeitabstände eine Rolle spielen. Zum anderen ist es insbesondere im Zusammenhang mit Realzeitsystemen notwendig, ein *quantitatives* Maß für die Dauer, Verzögerung und Distanz (Reaktionszeit) von Vorgängen und Abläufen zu unterstützen. Mit Hilfe dieser quantitativen Eigenschaften

können demnach beispielsweise echte Realzeitaspekte modelliert werden.

Für die Modellierung des Verhaltens an den Systemschnittstellen ist ein lineares, globales Zeitmodell am besten geeignet. Die lineare Zeit unterstützt die globale Sicht auf das System und seine Wechselwirkung mit der Umgebung durch eine „globale Uhr“ und ist dadurch vergleichsweise einfach modellierbar.

Das einfachste lineare Zeitmodell ist die *reale Zeit* $\mathbb{R}$, die der Menge der reellen Zahlen entspricht. Dieses Modell ist sehr natürlich und entspricht dem Zeitmodell, wie es aus anderen naturwissenschaftlichen Disziplinen, z.B. der Physik, bekannt ist. Die reale Zeit ist nach dieser Definition nicht beschränkt, d.h., es können prinzipiell alle Zeitpunkte in der Zukunft und in der Vergangenheit betrachtet werden. Diese Unbeschränktheit vereinfacht die Modellierung; in der praktischen Anwendung ist die Menge der *relevanten* Zeitpunkte jedoch meist beschränkt.

Einige der im folgenden Abschnitt 3.2 diskutierten Techniken dienen der Modellierung eines Systems und nicht nur seiner Schnittstelle. Interoperiert ein solches System mit seiner Umgebung in realer Zeit, so ist die Zuordnung von an der Schnittstelle zwischen System und Umgebung beobachtbaren Aktionen und Vorgängen zu realen Zeitpunkten sehr natürlich. Führt das System innerhalb einer solchen Aktion jedoch mehrere *interne* Schritte aus, verliert sich der reale Zeitbezug, da der exakte reale Zeitpunkt, zu dem ein Teilschritt abgearbeitet wurde, für die Interaktion mit der Umgebung nicht beobachtbar ist. Die Zeitpunkte mehrerer zusammenhängender Rechenschritte werden in diesem Fall idealisiert ein und demselben realen Zeitpunkt zugeordnet, so dass die für interne Rechenschritte notwendige reale Rechenzeit idealisiert als Null definiert ist. Für die Schnittstelle zur Umwelt des Systems und damit auch für den Black-Box-Test sind diese Zwischenschritte nicht beobachtbar und damit auch nicht verhaltensrelevant. In der Literatur wird auch von Macro- und Microsteps gesprochen [HN96, AH92].

Die Verallgemeinerung der realen Zeit um die Modellierung solcher Teilschritte wird im Folgenden als *universelle Zeit* bezeichnet und basiert auf den Konzepten der Timed State Sequences [HMP92], der Interval Sequences [AH92] und der Hybrid Traces [MMP92]. Formal wird die universelle Zeit als eine Menge T definiert, für die eine totale Ordnung $\leq$ und eine Funktion $\|\_\| : \mathsf{T} \to \mathbb{R}$ existieren, die für jeden Zeitpunkt $t \in \mathsf{T}$ den zugeordneten *realen Zeitanteil* $\|t\| \in \mathbb{R}$ definiert. Für einen realen Zeitpunkt $x \in \mathbb{R}$ wird die Anzahl der universellen Zeitpunkte $t \in \mathsf{T}$ mit dem realen Anteil $\|t\| = x$ als *Multiplizität* bezeichnet [AH92]. Ein realer Zeitpunkt x heißt *Interleaving-Punkt*, wenn seine Multiplizität größer als 1 ist. Für die universelle Zeit gelten folgende axiomatische Eigenschaften:

T1. Die Funktion $\|\_\|$ ist monoton steigend (d.h. $t \leq t' \iff \|t\| \leq \|t'\|$).

T2. Die Multiplizität ist für alle $x \in \mathbb{R}$ endlich.

T3. In jedem reellen Intervall $[x_1, x_2]$ gibt es nur endlich viele Interleaving-Punkte.

T4. Die Menge der realen Zeitanteile $\{\|t\| \mid t \in \mathsf{T}\}$ ist unbeschränkt.

Durch die Eigenschaft T1 ist garantiert, dass sich die Ordnung der universellen Zeitpunkte und die des realen Anteils nicht widersprechen. Mit T2 und T3 ist sichergestellt, dass in jedem reellen Zeitintervall höchstens endlich viele universelle Zeitpunkte existieren, für die die reale Zeit nicht fortschreitet: Existieren zwei universelle Zeitpunkte $t_1 \neq t_2$ mit $\|t_1\| = \|t_2\|$, so repräsentieren diese einen „internen Rechenschritt“ des Systems, bei dem die verbrauchte reale Zeit unerheblich ist und idealisiert als 0 abstrahiert wird. Von solchen Schritten kann es in jedem reellen Zeitintervall sinnvollerweise nur endlich viele geben. Die Eigenschaft T4 garantiert die *Non-Zeno*-Eigenschaft (vgl. [AL92]), d.h., es wird zugesichert, dass die reale Zeit nicht konvergiert, sondern mit fortschreitender Zeit unbeschränkt wächst. Gleichzeitig folgt aus T4, dass auch T unbeschränkt und (abzählbar oder überabzählbar) unendlich ist.

Eine universelle Zeitmenge T heißt *streng monoton*, wenn $\|\_\|$ streng monoton ist; sonst heißt sie *Interleaving-Zeit* oder *schwach monoton*. Streng monotone Zeitmodelle haben offensichtlich keine Interleaving-Punkte.

Weiterhin heißt eine universelle Zeitmenge T *dicht*, wenn $\{\|t\| \mid t \in \mathsf{T}\} = \mathbb{R}$ ist. Die reale Zeit $\mathbb{R}$ ist demnach ein Spezialfall der universellen Zeit. $\mathbb{R}$ ist dicht und streng monoton.

T heißt *diskret*, wenn $\{\|t\| \mid t \in \mathsf{T}\}$ abzählbar unendlich ist, d.h., diskrete Zeitmengen lassen sich immer als unendliche Folge der Form $\cdots < t_{-2} < t_{-1} < t_0 < t_1 < \cdots < t_i < \ldots$ darstellen. Zeitmodelle, die weder dicht noch diskret sind, sind zwar nach obiger Definition theoretisch möglich; sie sind jedoch auf Grund des Paradigmenwechsels nicht sinnvoll einsetzbar und werden im Folgenden nicht weiter betrachtet.

3.2 Kontinuierliche Modellierung

Mit Hilfe des universellen Zeitbegriffs werden nun verschiedene Techniken zur Modellierung des kontinuierlichen Verhaltens gegenübergestellt. Es wird sich zeigen, dass sich Modellierungstechniken besonders gut eignen, die auf einem dichten, streng monotonen Zeitmodell (isomorph zu $\mathbb{R}$) operieren.

Jeder mit TPT modellierte Testfall muss einerseits einen Verlauf für die Eingabegrößen vorgeben und andererseits den Verlauf der Ausgangsgrößen des Systems auswerten können. Wie bereits im Kapitel 1 erwähnt wurde, soll dies zwecks Minimierung des Testaufwandes automatisiert möglich sein. Jeder einzelne Testfall muss hierfür ein semantisch eindeutig definiertes, ausführbares Szenario beschreiben. Zur Modellierung eines Testfalls wird also eine Beschreibungstechnik benötigt, mit der sich solche Szenarien beschreiben lassen.

Temporale Logik. Die klassische temporale Logik [Pnu77] dient der Modellierung von zeitdiskretem Verhalten, d.h., ihr liegt ein diskretes Zeitmodell zu Grunde. Mit dieser Logik können demnach keine kontinuierlichen Verläufe modelliert werden. In mehreren Erweiterungen der temporalen Logik wurde die Behandlung von dichter (realer) Zeit ermöglicht [Ost89, AH89]. Dennoch ist die temporale Logik vorrangig zur Modellierung loser Eigenschaften eines Systems entwickelt worden. Eine deterministische Modellierung des Systemverhaltens mit temoraler Logik ist demnach generell problematisch. Außerdem ist die temporale Logik nur stark eingeschränkt ausführbar. Aus diesen Gründen ist der Einsatz der temporalen Logik als Modellierungstechnik für das kontinuierliche Verhalten nicht geeignet.

Timed Automata. Diese Technik ist sehr gut für die ausführbare Modellierung des kontinuierlichen Verhaltens geeignet [AH92, Alu99]. Insbesondere ist eine anschauliche grafische Notation des Verhaltens in Form spezieller Zustandsautomaten möglich. Entscheidender Nachteil ist aber, dass die Technik zwar auf der realen Zeit (dichte, streng monotone Zeit) basiert, die einzigen echt kontinuierlichen Verläufe aber die Uhren (*clocks*) sind, die einen konstanten Anstieg von 1 haben. Dadurch sind Timed Automata vergleichsweise einfach definiert und unterstützen die formale Analyse des Verhaltens. Ihre Ausdrucksmächtigkeit für kontinuierliches Verhalten ist jedoch so stark eingeschränkt, dass sie nicht verwendet werden können.

Timed Transition Systems. Timed Transition Systems unterstützen – ähnlich wie die Timed Automata – eine einfache grafische Notation für die Verhaltensmodellierung (vgl. [HMP92, AH92]). Der Nachteil dieses Ansatzes liegt in dem zu Grunde liegenden diskreten Interleaving-Zeitmodell, das für die Timed Transition Systems in Form von Timed State Sequences definiert ist. Die fehlende Mächtigkeit hinsichtlich der Modellierung dichter Verläufe schränkt die Einsatzfähigkeit der Timed Transition Systems zu stark ein, so dass auch diese Technik für das Time Partition Testing nicht verwendet werden kann.

Stromverarbeitende Funktionen. Ein vielversprechender Ansatz ist die Idee, kontinuierliche Systeme als Funktionen zu modellieren, die Datenströme verarbeiten und Datenströme produzieren. Insbesondere sind die Arbeiten von [Bro97] und [MS97] interessant, da sie auch die Modellierung von Strömen bzw. Verläufen über der dichten, streng monotonen Zeit unterstützen. Kontinuierliche Verläufe lassen sich wegen der dichten Zeit sehr einfach und natürlich modellieren. Die strenge Monotonie der Zeit ist für die Modellierung des Schnittstellenverhaltens ohne Betrachtung interner Berechnungsschritte ebenfalls nahe liegend. Die stromverarbeitenden Funktionen sind kompositional, d.h., sie lassen sich hierarchisch von einfachen zu komplexen Funktionen

zusammensetzen, was der im Kapitel 4 vorgestellten Idee der hierarchischen Modellierung von Testfällen entgegenkommt.

Für die Modellierung des kontinuierlichen Verhaltens wird im folgenden Abschnitt 3.3 die Idee der stromverarbeitenden Funktionen aufgegriffen. Für die weiter unten erläuterte Kopplung mit der Technik hybrider Systeme, die eine sehr mächtige und intuitive Basis für die Modellierung des kontinuierlichen Verhaltens darstellt, sind jedoch spezifische Erweiterungen der stromverarbeitenden Funktionen notwendig, die die Entwicklung eines eigenen Modells notwendig machen.

Hybride Systeme. Eine spezielle Form von endlichen Automaten ist in der Literatur als hybride Systeme (vgl. [ACH+95]) bekannt und basiert auf der Idee der Phase Transition Systems [MMP92, NSY93]. Die hybriden Systeme beruhen auf der Zerlegung der realen Zeit in eine Sequenz von Phasen. In jeder Phase beschreiben kontinuierliche Gesetze das Verhalten der Variablen. Das Verhalten an einem Phasenübergang wird hingegen durch zeitlose Bedingungen und Anweisungen bestimmt, die spontan operieren.

Den hybriden Systemen liegt eine dichte Interleaving-Zeit zu Grunde. Die Interleaving-Punkte entsprechen den Transitionspunkten, d.h., der Wechsel zwischen zwei Phasen des hybriden Systems erfolgt zeitlos (in Bezug auf die reale Zeit). Die Theorie der hybriden Systeme eignet sich neben den stromverarbeitenden Funktionen hervorragend für die Modellierung von kontinuierlichem Verhalten. Insbesondere die Möglichkeit hybrider Systeme, das Verhalten als eine Folge von Phasen modellieren zu können, ist – wie später demonstriert wird – für die Modellierung von Testfällen für das kontinuierliche Verhalten sehr natürlich.

Fazit. Die semantische Fundierung der Testmodellierung von TPT soll sowohl die Vorteile der stromverarbeitenden Funktionen als auch der hybriden Systeme vereinen. Als Zeitmodell für diese Modellierung eignet sich die dichte, streng monotone Zeit am besten und wird im Folgenden durch die Menge der reellen Zahlen $\mathbb{R}$ repräsentiert.

Beide Techniken werden in den folgenden beiden Abschnitten 3.3 und 3.4 in einer für TPT spezifischen Form eingeführt und integriert. Sie bilden damit einen mächtigen Kalkül für die Testmodellierung von kontinuierlichem Verhalten.

3.3 Stromverarbeitende Funktionen

Die im Folgenden gegebene Definition stromverarbeitender Funktionen basiert in ihren Grundkonzepten auf der Definition aus [Bro97]. Der entscheidende Unterschied ist die Einführung des Sequenzialisierungsoperators, mit dem es

möglich ist, zu einem definierten Zeitpunkt τ spontan von einer Verhaltensbeschreibung A zu einer Verhaltensbeschreibung B zu wechseln. Mit Hilfe dieses Operators wird im Abschnitt 3.4 die Integration stromverarbeitender Funktionen mit hybriden Systemen formal fundiert.

Ein *Typ* M ist eine beliebige Menge von so genannten *Messages*, die die dummy-Message $\varepsilon \in M$ enthalten muss. Ein *Strom* des Typs M ist eine totale Funktion $s : \mathbb{R} \to M$, die jedem realen Zeitpunkt $t \in \mathbb{R}$ eine Message $s(t) \in M$ zuordnet. Die Message $s(t) = \varepsilon$ repräsentiert in einem Strom eine „zum Zeitpunkt t nicht vorhandene“ Botschaft. Diese Repräsentation partieller Ströme ist im Sinne eines einfachen Modells besser geeignet als der Umgang mit partiellen Abbildungen.

Die Tatsache, dass Ströme nach dieser Definition im Gegensatz zu [Bro97] auch für negative Zeitpunkte definiert sind, wird im Zusammenhang mit der Definition von Komponenten im nächsten Abschnitt erklärt.

Der Strom $s \oplus \tau$ bezeichnet die Verschiebung des Stroms s um die Zeitspanne $\tau \in \mathbb{R}$ „nach links“, d.h., $(s \oplus \tau)(t) = s(t + \tau)$. Offensichtlich gilt $(s \oplus \tau_1) \oplus \tau_2 = s \oplus (\tau_1 + \tau_2)$.

Gegeben sei eine endliche Menge C von *Kanälen*. Ein Kanal repräsentiert eine benannte Variable, der Ströme zugeordnet werden können. Jeder Kanal $\alpha \in C$ hat einen definierten Typ M_α. Eine *Belegung* c der Kanäle C ordnet jedem Kanal $\alpha \in C$ einen Strom $c_\alpha : \mathbb{R} \to M_\alpha$ zu. Als $\vec{C}$ wird die Menge aller möglichen Belegungen der Kanäle aus C bezeichnet. Folgende Begriffe werden für Belegungen definiert:

- Für eine Belegung $c \in \vec{C}$ sowie eine weitere Menge von Kanälen $D \subseteq C$ bezeichnet $c|_D$ die übliche Domain-Restriktion.

- Die Domain-Restriktion wird auf Belegungen verallgemeinert: Für eine Belegung $c \in \vec{C}$ und ein reelles Zeitintervall I bezeichnet $c|_I$ die Zuordnung von $c_\alpha|_I$ für alle $\alpha \in C$.

- Für Belegungen $c \in \vec{C}$ und $d \in \vec{D}$ mit disjunkten Mengen C und D ist $c \cup d$ die vereinigte Belegung der Kanäle aus $C \cup D$, d.h., $c \cup d \in \overrightarrow{C \cup D}$. Alternativ kann aber auch $(c, d) \in \vec{C} \times \vec{D}$ geschrieben werden. Die Mengen $\overrightarrow{C \cup D}$ und $\vec{C} \times \vec{D}$ beschreiben beide die Menge aller möglichen Belegungen für Kanäle aus C und D. Die Notationen sind also austauschbar und werden je nach Kontext entsprechend der jeweils besseren Lesbarkeit eingesetzt. Auf Grund der Disjunktheit gilt: $\overrightarrow{C \cup D} \hat{=} \vec{C} \times \vec{D}$ und $c \cup d \hat{=} (c, d)$.

- Für eine Belegung $c \in \vec{C}$ ist $c \oplus \tau$ die Belegung, die durch Anwendung der Verschiebung $s \oplus \tau$ auf alle in c enthaltenen Ströme s entsteht.

- Zwei Belegungen $c, c' \in \vec{C}$ heißen *kompatibel in* I, wenn c und c' im Zeitintervall I gleich sind, d.h., wenn $c|_I = c'|_I$. Zwei in I kompati-

ble Belegungen werden als $c =_I c'$ notiert. Weiterhin werden folgende abkürzenden Schreibweisen verwendet:

$$\begin{array}{lll} c =_{\leq t} c' & \text{gdw.} & c =_{(-\infty,t]} c' \\ c =_{<t} c' & \text{gdw.} & c =_{(-\infty,t)} c' \\ c =_{\geq t} c' & \text{gdw.} & c =_{[t,\infty)} c' \end{array}$$

3.3.1 Komponenten

Der zentrale Begriff für stromverarbeitende Funktionen ist der einer Komponente. Er wird zunächst formal definiert und anschließend erläutert.

Gegeben seien zwei disjunkte, endliche Mengen von Kanälen I und O, die Eingänge bzw. Ausgänge genannt werden. Eine *Komponente* von I nach O ist eine Relation $F : \vec{I} \leftrightarrow \vec{O}$, die folgende Eigenschaft erfüllt, die als *Zeitkausalität* bezeichnet wird: Es existiert ein reelles $\delta > 0$, so dass für alle Belegungen $i, i' \in \vec{I}$ und $o, o' \in \vec{O}$ mit $o =_{<0} o'$, $(i, o) \in F$ und $(i', o') \in F$ sowie für alle $t \geq 0$ folgende Eigenschaft gilt:

$$i =_{\leq t} i' \wedge o =_{\leq t-\delta} o' \implies o =_{\leq t} o'$$

Die Eigenschaft $(i, o) \in F$ wird im Folgenden kurz als $F(i, o)$ notiert.

Erläuterung: Grob gesprochen definiert eine Komponente eine Belegung für die Ausgänge O in Abhängigkeit der Belegung der Eingänge I (vgl. Abb. 3.1). Eine für die Integration mit hybriden Systemen wichtige Eigenschaft von Komponenten ist die Möglichkeit, dass sie sich zeitlich sequenziell verketten lassen[1]. Das bedeutet, dass eine Komponente F_1 die Belegung der Ausgänge O bis zu einem Zeitpunkt τ bestimmt und ab diesem Zeitpunkt eine Komponente F_2 die weitere Belegung von O definiert. Sowohl F_1 als auch F_2 haben hierbei eine in sich geschlossene, klar definierte Semantik. Bei der sequenziellen Verkettung ist es aber häufig notwendig, dass die Komponente F_2 Bezug auf die Belegung von O *vor* dem Zeitpunkt τ nehmen muss. Soll F_2 beispielsweise stetig und konstant an den bisherigen, von F_1 definierten Verlauf von O anknüpfen, ohne F_1 selbst zu kennen, muss F_2 in der Lage sein, auf die „Historie der Belegung von O“ zuzugreifen.

Um den Komponentenbegriff möglichst einfach definieren zu können, wird deshalb festgelegt, dass *jede* Komponente F die Belegung der Ausgänge O nur für Zeitpunkte $t \geq 0$ definiert, während die Belegung für Zeitpunkte $t < 0$ *nicht* durch F charakterisiert wird, sondern die „Historie“ der Ausgänge beschreibt. Der definierte Startpunkt einer Komponente ist also immer $t = 0$. Gilt das Verhalten der obigen Komponente F_2 ab einem Zeitpunkt $\tau \neq 0$, so verschiebt sich der Nullzeitpunkt aus Sicht von F_2 so, dass τ dem „lokalen Nullpunkt“ von F_2 entspricht (vgl. Abschn. 3.3.6). Da eine Komponente die

[1] Diese Eigenschaft wird im Abschnitt 3.3.6 zur Definition des Sequenzialisierungsoperators ausgenutzt.

Belegung von O für negative Zeitpunkte generell nicht definiert, ist sie keine Funktion $\vec{I} \to \vec{O}$, sonderen eine Relation $\vec{I} \leftrightarrow \vec{O}$.

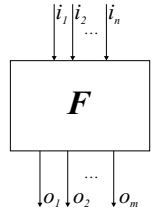

Abbildung 3.1: Schemadarstellung einer Komponente

Die charakteristische Eigenschaft von Komponenten ist die Zeitkausalität. Sie sichert Folgendes zu: Sind die Belegung der Eingänge I bis zu einem Zeitpunkt t und die Belegung der Ausgänge O bis $t-\delta$ bekannt, dann bestimmt eine Komponente eindeutig die Belegung der Ausgänge bis t. Eine Komponente kann also für die Definition des Verhaltens an den Ausgängen sowohl auf die Eingänge als auch auf die Historie der Ausgänge selbst Bezug nehmen. Dieser Zusammenhang wird auf nicht negative Zeitpunkte ($t \geq 0$) beschränkt. Für negative Zeitpunkte wird generell gefordert, dass die Ausgangsbelegungen bekannt sein müssen. (Dies entspricht der formalen Vorauss. $o =_{<0} o'$ der Zeitkausalität.)

Durch die Zeitkausalität werden zwei Eigenschaften sichergestellt: (1) Die Belegung von O ist zu jedem Zeitpunkt $t \geq 0$ ohne Kenntnis von Belegungen aus der „Zukunft“ eindeutig und kann somit “online” berechnet werden. (2) Die Belegung von O hängt für positive Zeitpunkte nicht von sich selbst ab. Jede Komponente beschreibt für eine gegebene Belegung $i \in \vec{I}$ (für alle $t \in \mathbb{R}$) und eine gegebene Belegung $o \in \vec{O}$ (für alle $t < 0$) eindeutig den Verlauf von o für alle $t \geq 0$, wie das nachfolgende Theorem 3.1 zeigt, das ein Korollar zu dem weiter unten angegebenen Theorem 3.2 ist. Die Definition der Zeitkausalität ist auch in [Bro97] unter dem Begriff *time guarded by a finite delay* und in [MS97] unter dem Begriff *delayed behavior* zu finden.

Theorem 3.1. Für jede Komponente $F : \vec{I} \leftrightarrow \vec{O}$ mit Belegungen $i \in \vec{I}$ sowie $o, o' \in \vec{O}$ mit $F(i, o)$ und $F(i, o')$ gilt

$$o =_{<0} o' \implies o = o'$$

(Die Beweise dieses sowie aller folgenden Theoreme und Korollare sind im Anhang A enthalten.)

Initialbelegung und Totalität. Die Eigenschaft des Theorems 3.1 beschreibt, dass jede Komponente für eine gegebene *Initialbelegung* (d.h. für

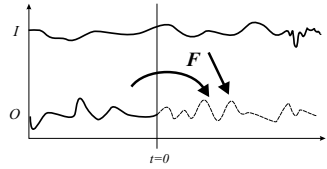

Abbildung 3.2: Definition der Ausgänge durch eine Komponente

Belegungen der Eingänge für alle Zeitpunkte und der Ausgänge für negative Zeitpunkte) eindeutig die Fortsetzung der Ausgangsbelegungen (für alle nicht negativen Zeitpunkte) definiert (vgl. Abb. 3.2). Der Begriff der Initialbelegung wird folgendermaßen formal definiert: Sei $\cong$ die Äquivalenzrelation mit $(i,o) \cong (i',o')$ gdw. $i = i'$ und $o =_{<0} o'$. Jede Restklasse $[i,o]_\cong$ zu dieser Relation heißt *Initialbelegung*. In jeder Initialbelegung $[i,o]_\cong$ ist nach Theorem 3.1 höchstens ein Paar (i,o') mit $F(i,o')$ enthalten. Existiert diese Belegung o', wird sie als $F[i,o] := o'$ bezeichnet.

Nicht für alle Initialbelegungen $[i,o]_\cong$ muss für eine Komponente die Belegung $F[i,o]$ definiert sein. Beispielsweise ist $F = \emptyset$ eine Komponente, die zu keiner Initialbelegung eine entsprechende „Fortsetzung“ $F[i,o]$ definiert. In diesem Sinne heißt eine Komponente F *total*, wenn für alle (i,o) die Belegung $F[i,o]$ definiert ist (sonst heißt sie *partiell*). Partielle Komponenten sind zwar theoretisch möglich, jedoch für die praktische Betrachtung nicht relevant. Es wird sich zeigen, dass die weiter unten definierten Operatoren die Totalität von Komponenten erhalten.

Verzögerte Wirksamkeit, Unabhängigkeit. Sei $F : \vec{I_1} \times \vec{I_2} \leftrightarrow \vec{O}$ eine Komponente (I_1 und I_2 disjunkt). Dann heißt die Menge I_2 bzgl. F *verzögert wirksam*, wenn ein $\delta > 0$ existiert, so dass für alle Belegungen $i_1, i_1' \in \vec{I_1}$, $i_2, i_2' \in \vec{I_2}$ und $o, o' \in \vec{O}$ mit $o =_{<0} o'$, $F(i_1, i_2, o)$ und $F(i_1', i_2', o')$ sowie für alle $t \geq 0$ folgende Eigenschaft gilt:

$$i_1 =_{\leq t} i_1' \wedge i_2 =_{\leq t-\delta} i_2' \wedge o =_{\leq t-\delta} o' \implies o =_{\leq t} o'$$

Ist eine Menge I_2 nicht verzögert wirksam bzgl. F, heißt sie *unmittelbar wirksam* bzgl. F.

Die verzögerte Wirksamkeit ist eine Verschärfung der Zeitkausalität, da sie die um δ verzögerte Wirkung neben den Ausgängen O auch für die Eingänge I_2 fordert. In diesem Fall darf das Verhalten der Komponente F an den Ausgängen O zum Zeitpunkt t also nicht direkt von den Eingangswerten für I_2 zu demselben Zeitpunkt t abhängen. Die Menge $I_2 = \emptyset$ ist für jede Komponente trivialerweise verzögert wirksam. Gilt die obige Eigenschaft für *alle* $\delta > 0$, heißt F *unabhängig* von $I_2 \cup O$.

Ist die Menge I_2 verzögert wirksam und ist $\delta > 0$ ein Wert, der die Bedingung der verzögerten Wirksamkeit erfüllt, dann gilt folgende Eigenschaft:

Theorem 3.2. Für alle Belegungen $i_1, i_1' \in \vec{I_1}$, $i_2, i_2' \in \vec{I_2}$ sowie $o, o' \in \vec{O}$ mit $o =_{<0} o'$, $F(i_1, i_2, o)$ und $F(i_1', i_2', o')$ sowie für alle $t \geq 0$ gilt

$$i_1 =_{\leq t} i_1' \wedge i_2 =_{\leq t-\delta} i_2' \implies o =_{\leq t} o'$$

Dieses Theorem drückt formal aus, dass eine Komponente die Ausgänge O bis einschließlich t bereits eindeutig charakterisiert, wenn die Eingänge I_1 bis zum Zeitpunkt t, die Eingänge I_2 bis $t - \delta$ und die Ausgänge O für negative Zeitpunkte bekannt sind.

3.3.2 Parallelisierung

Mit Hilfe der im Folgenden vorgestellten Parallelisierung ist es möglich, mehrere Komponenten parallel zu einer komplexeren Komponente zu kombinieren und dabei insbesondere zu ermöglichen, dass die so verschalteten Komponenten wechselseitig auf ihre Ausgänge referenzieren können (vgl. Abb. 3.3).

Definiert eine Komponente $F_1 : \vec{I_1} \times \vec{O_2} \leftrightarrow \vec{O_1}$ das Verhalten für Ausgänge O_1 und eine Komponente $F_2 : \vec{I_2} \times \vec{O_1} \leftrightarrow \vec{O_2}$ das Verhalten für O_2 (O_1 und O_2 disjunkt), so bilden diese beiden Komponenten zeitlich parallel verschaltet wiederum eine Komponente, die das Verhalten von $O_1 \cup O_2$ definiert. Bei dieser Parallelisierung soll es möglich sein, dass die Komponenten gegenseitig auf ihre Ausgänge Bezug nehmen können. Deshalb sind die Ausgänge O_1 von F_1 gleichzeitig Eingänge von F_2; analog für O_2 (vgl. Abb. 3.3). Die Parallelisierung zweier Komponenten F_1 und F_2 ist wie folgt definiert:

Seien zwei Komponenten F_1 und F_2 wie oben gegeben, wobei o.B.d.A. die Eingänge O_2 bzgl. F_1 verzögert wirksam sind. Dann heißt die Relation $F_1 \parallel F_2 : \overrightarrow{I_1 \cup I_2} \leftrightarrow \overrightarrow{O_1 \cup O_2}$ *Parallelisierung* von F_1 und F_2, wenn für alle $i \in \overrightarrow{I_1 \cup I_2}$, $o \in \overrightarrow{O_1 \cup O_2}$ gilt:

$$(F_1 \parallel F_2)(i, o) \text{ gdw. } F_1(i|_{I_1}, o|_{O_2}, o|_{O_1}) \text{ und } F_2(i|_{I_2}, o|_{O_1}, o|_{O_2})$$

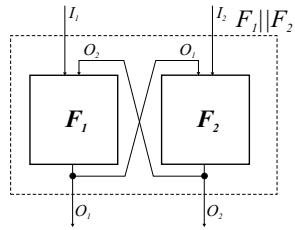

Abbildung 3.3: Schema einer Parallel-Komponente

Die Voraussetzung der verzögerten Wirksamkeit von O_2 bzgl. F_1 ist notwendig, damit eine der Komponenten – in diesem Fall F_1 – zu jedem Zeitpunkt t unabhängig von den Ausgängen der anderen Komponente F_2 ist. Nur unter dieser Voraussetzung ist die Zeitkausalität gewährleistet. Man beachte, dass die verzögerte Wirksamkeit aber nur in eine Richtung gelten muss, wie folgendes Theorem zeigt:

Theorem 3.3. Für alle Komponenten $F_1 : \vec{I_1} \times \vec{O_2} \leftrightarrow \vec{O_1}$ und $F_2 : \vec{I_2} \times \vec{O_1} \leftrightarrow \vec{O_2}$, für die o.B.d.A. die Eingänge O_2 bzgl. F_1 verzögert wirksam sind, ist $F_1||F_2$ eine Komponente.

Nach diesem Theorem ist sichergestellt, dass die Parallelisierung zweier Komponenten wieder eine Komponente ist. Es lassen sich weiterhin folgende interessante Eigenschaften zeigen.

Theorem 3.4.

1. Die Parallelisierung ist kommutativ und assoziativ.

2. Die Parallelisierung zweier totaler Komponenten ist total.

Die Parallelisierung erhält also die Totalität der parallelisierten Komponenten, d.h., eine solche parallelisierte Komponente definiert für alle Initialbelegungen eine entsprechende Fortsetzung.

3.3.3 Der Feedback-Operator

Als Spezialfall der Parallelisierung kann ein Feedback einer Komponente definiert werden. Hierbei sollen die Ausgänge einer Komponente an Teile der Eingänge zurückgeführt werden. Ein Beispiel: Eine Komponente mit der Definitionsgleichung $out(t) = in(t-0.7)$ ist so definiert, dass sie als Ausgangsstrom *out* die Verzögerung des Eingangsstroms *in* um 0.7 modelliert. Wird für diese Komponente der Kanal *out* nach *in* so zurückgeführt, dass die Belegungen der Kanäle identifiziert werden, dann entsteht eine neue Komponente, für die es nur noch einen relevanten Kanal *out* gibt, dessen Belegung die Gleichung $out(t) = out(t - 0.7)$ erfüllt, so dass ein periodisches Signal entsteht.

Der Feedback-Operator kann als Spezialfall der Parallelisierung dadurch definiert werden, dass die rückzukoppelnde Komponente F mit einer speziellen Umbenennungskomponente *Ren* parallelisiert wird, die die Belegungen der rückzukoppelnden Kanäle unverändert auf die entsprechenden Eingangskanäle durchführt (vgl. Abb. 3.4).

Den Eigenschaften der Parallisierung entsprechend lässt sich leicht zeigen, dass das Feedback einer Komponente wiederum eine Komponente bildet, wenn der rückzukoppelnde Eingang verzögert wirksam ist. Außerdem ist eine solche Feedback-Komponente total, wenn F total ist.

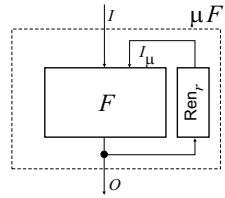

Abbildung 3.4: Darstellung einer Feedback-Komponente

Sowohl in [Bro97] als auch in [MS97] wurde der Beweis der Eindeutigkeit der Feedback-Konstruktion ebenfalls erbracht. Der entsprechende Beweis im Kontext der vorliegenden Arbeit (Spezialfall des Beweises von Theorem 3.3) unterscheidet sich von den Beweisen dieser Arbeiten wegen der Verallgemeinerung auf negative Zeitpunkte.

3.3.4 Ausdrücke und Gleichungen

Eine sehr natürliche Form der Definition von Komponenten sind Gleichungen der Form $c(t) = expr$ für einen Kanal c, wobei $expr$ ein Definitionsterm ist, in dem auf beliebige Kanäle – insbesondere auch auf c selbst – sowie auf den aktuellen Zeitpunkt t Bezug genommen werden kann. Die konkreten Operatoren, mit denen ein solcher Definitionsterm gebildet werden kann, werden erst im Abschnitt 4.4.3 erläutert. Die Konstruktion von Ausdrücken soll im Folgenden formal definiert werden.

Sei I eine endliche Menge von Kanälen. Eine Funktion $e : \vec{I} \to \mathbb{R} \to M$ heißt *stromverarbeitender Ausdruck vom Typ* M, falls für alle $i, i' \in \vec{I}$ und alle $t \in \mathbb{R}$ gilt $i =_{\underline{\leq} t} i' \implies e(i)(t) = e(i')(t)$. Diese Eigenschaft ist mit der Zeitkausalität von Komponenten vergleichbar: Der durch den Ausdruck e definierte Strom darf zu jedem Zeitpunkt nur von der Belegung i bis einschließlich t abhängen. Entsprechend kann auch die Eigenschaft der verzögerten Wirksamkeit auf Ausdrücke übertragen werden: Für zwei disjunkte Mengen I und I_v heißt I_v *verzögert wirksam* bzgl. des Ausdrucks $e : \vec{I} \times \vec{I_v} \to \mathbb{R} \to M$, wenn ein $\delta > 0$ existiert, so dass für alle $i, i' \in \vec{I}$, $i_v, i'_v \in \vec{I_v}$ und alle $t \in \mathbb{R}$ gilt $i =_{\underline{\leq} t} i' \wedge i_v =_{\underline{\leq} t-\delta} i'_v \implies e(i, i_v)(t) = e(i', i'_v)(t)$.

Für einen Kanal c sowie einen Ausdruck $e_c : \vec{I} \times \{\vec{c}\} \to \mathbb{R} \to M_c$ vom Typ M_c, für den $\{c\}$ verzögert wirksam ist, heißt die Komponente $G_c : \vec{I} \leftrightarrow \{\vec{c}\}$ mit

$$G_c(i, o) \iff \forall t \geq 0 \bullet o_c(t) = e_c(i, o)(t)$$

Gleichung für c mit e_c. Sie ist so definiert, dass sie die Eingangsbelegung o von c mit dem Funktionswert von e_c identifiziert und dadurch letztendlich ein Feedback bildet. Durch diese Konstruktion ist es möglich, dass im

Definitionsterm der Gleichung der Kanal c selbst referenziert werden kann. Dass diese Konstruktion tatsächlich eine Komponente definiert, wird durch folgendes Theorem nachgewiesen:

Theorem 3.5. Jede Gleichung G_c für einen Ausdruck $e_c : \vec{I} \times \{\vec{c}\} \to \mathbb{R} \to M_c$, für den $\{c\}$ verzögert wirksam ist, ist erstens eine Komponente und zweitens total.

3.3.5 Temporale Prädikate

Temporale Prädikate werden in dieser Arbeit eingeführt, um die Übergangsbedingungen für den Sequenzialisierungsoperator formal fundieren zu können (vgl. Abschn. 3.3.6). Ein temporales Prädikat legt in Abhängigkeit einer gegebenen Eingangsbelegung i für jeden Zeitpunkt t fest, ob das Prädikat erfüllt ist oder nicht. Der kleinste erfüllende Zeitpunkt $\tau \geq 0$ wird dabei als Transitionspunkt interpretiert. Da die Existenz eines solchen kleinsten Zeitpunktes wegen des dichten Zeitmodells $\mathbb{R}$ i. Allg. nicht garantiert ist, werden temporale Prädikate folgendermaßen definiert:

Die Menge $\mathbb{B} = \{0, 1\}$ heißt *boolscher Typ*. Ein stromverarbeitender Ausdruck $P : \vec{I} \to \mathbb{R} \to \mathbb{B}$ heißt *temporales Prädikat*, wenn für alle $i \in \vec{I}$ ein eindeutig definierter Schaltpunkt $\tau \geq 0$ ($\tau \in \mathbb{R}^\infty$) existiert mit

$$\begin{aligned} P(i)(t) &= 0 \quad \text{für alle } t \text{ mit } 0 \leq t < \tau \\ P(i)(\tau) &= 1 \quad \text{falls } \tau \neq \infty \end{aligned}$$

τ ist demnach der erste nicht negative Zeitpunkt, zu dem das Prädikat erfüllt ist. Gibt es keinen nicht negativen Zeitpunkt, der P erfüllt, gilt per Definition $\tau = \infty$. Der so definierte Zeitpunkt τ wird für temporale Prädikate als $P[i] := \tau$ notiert.

Anmerkung: Ein temporales Prädikat ist ein spezifischer stromverarbeitender Ausdruck und deshalb punktweise definiert. Der Wahrheitswert zu einem Zeitpunkt t kann grundsätzlich vom gesamten Verlauf $i \in \vec{I}$ für alle $t' \leq t$ abhängen, so dass auch temporallogische Ausdrücke modelliert werden können. Die weiter unten im Abschnitt 4.4.3 beschriebenen Operatoren, mit denen stromverarbeitende Ausdrücke und temporale Prädikate gebildet werden können, schließen die Verwendung temporaler Operatoren (*always*, *until*, *chop* etc.) jedoch aus, um ein effektives Berechnungsmodell für modellierte Testfälle garantieren zu können.

3.3.6 Sequenzialisierung

Definiert eine Komponente $F_1 : \vec{I} \leftrightarrow \vec{O}$ das Verhalten bis zu einem Zeitpunkt $\tau \geq 0$ und definiert ab diesem Zeitpunkt eine andere Komponente $F_2 : \vec{I} \leftrightarrow \vec{O}$ das weitere Verhalten, so bildet diese *Sequenzialisierung* der Komponenten wiederum eine Komponente.

Diese Idee der zeitlichen Verkettung kann verallgemeinert werden, indem anstelle eines festen Zeitpunktes τ ein temporales Prädikat zur Festlegung des Transitionspunktes verwendet wird. Der Transitionspunkt τ ist somit nicht fest vorgegeben, sondern in Abhängigkeit der Eingangsbelegungen definiert: Gegeben seien die Komponenten F_1 und F_2 wie oben und ein Prädikat $P : \overrightarrow{I \cup O} \to \mathbb{R} \to \mathbb{B}$, für das O verzögert wirksam ist. Dann heißt die Funktion $F_1 \overset{P}{\rightsquigarrow} F_2 : \vec{I} \leftrightarrow \vec{O}$ *Sequenzialisierung* von F_1 und F_2 mit der *Transitionsbedingung* P und ist folgendermaßen definiert: Für alle Belegungen $i \in \vec{I}$ und $o \in \vec{O}$ mit $(F_1 \overset{P}{\rightsquigarrow} F_2)(i, o)$ heißt $\tau := P[i \cup o]$ *Transitionspunkt* bzgl. (i, o), und es gibt eine Belegung $\bar{o} \in \vec{O}$ mit $o =_{<\tau} \bar{o}$, so dass gilt:

$$\begin{array}{l} F_1(i, \bar{o}) \\ F_2(i \oplus \tau, o \oplus \tau) \quad \text{, falls } \tau \neq \infty \end{array}$$

Die Sequenzialisierung ist demnach so definiert, dass ihre Ausgangsbelegung o bis ausschließlich τ mit der Ausgangsbelegung $\bar{o}$ der Komponente F_1 übereinstimmt. Ab dem Zeitpunkt τ bestimmt F_2 den weiteren Verlauf von o. Für F_2 wird – entsprechend der Definition der Verschiebung von Belegungen – bei τ die lokale Uhr zurückgesetzt. Im Falle $\tau = \infty$ gilt $\bar{o} = o$, und das Verhalten von F entspricht dem von F_1, unabhängig von F_2. Dieser Grenzfall modelliert eine Transition, die für die gegebenen Belegungen niemals schaltet.

Interessant an dieser Definition ist vor allem das Verhalten von F zum Transitionszeitpunkt τ selbst. Die Ausgangsbelegung für τ wird nach obiger Definition durch die Komponente F_2 bestimmt. Die Transition schaltet also *bevor* die Belegung der Ausgänge bestimmt wird. Aus diesem Grund müssen für die Transitionsbedingung die Kanäle O auch verzögert wirksam sein. Nur mit dieser Bedingung ist der kausale Zusammenhang zwischen F_1, P und F_2 für die Sequenzialisierung garantiert, wie folgendes Theorem belegt:

Theorem 3.6. Für beliebige Komponenten $F_1 : \vec{I} \leftrightarrow \vec{O}$ und $F_2 : \vec{I} \leftrightarrow \vec{O}$ sowie ein Prädikat $P : \overrightarrow{I \cup O} \to \mathbb{R} \to \mathbb{B}$, für das O verzögert wirksam ist, ist $F_1 \overset{P}{\rightsquigarrow} F_2$ eine Komponente.

Interessant und von praktischer Bedeutung ist wiederum die Tatsache, dass die Sequenzialisierung die Totalität erhält:

Theorem 3.7. Seien F_1 und F_2 zwei totale Komponenten und P ein temporales Prädikat, die die Voraussetzung der Sequenzialisierung erfüllen. Dann ist auch $F_1 \overset{P}{\rightsquigarrow} F_2$ total.

Korollar 3.8. Ist T ein Prädikat, das immer erfüllt ist, und F ein Prädikat, das niemals erfüllt ist, dann gelten folgende Eigenschaften für beliebige Komponenten F_1 und F_2:

1. $(F_1 \overset{T}{\rightsquigarrow} F_2) = F_2$

2. $(F_1 \overset{F}{\rightsquigarrow} F_2) = F_1$

Man beachte, dass $\rightsquigarrow$ nicht assoziativ ist, da beispielsweise nach obigem Korollar $(F_1 \overset{F}{\rightsquigarrow} F_2) \overset{T}{\rightsquigarrow} F_3 = F_3$ gilt, aber $F_1 \overset{F}{\rightsquigarrow} (F_2 \overset{T}{\rightsquigarrow} F_3) = F_1$ ist. Wird die Klammerung im Folgenden weggelassen, wird immer Rechts-Assoziativität angenommen, d.h., es gilt $F_1 \overset{P}{\rightsquigarrow} F_2 \overset{Q}{\rightsquigarrow} F_3 = F_1 \overset{P}{\rightsquigarrow} (F_2 \overset{Q}{\rightsquigarrow} F_3)$.

Stetige Anknüpfungen. Ein Beispiel, das zeigt, wofür die obige Definition der Sequenzialisierung effektiv eingesetzt werden kann, sind stetige Anknüpfungen von Signalverläufen. Betrachtet man Ströme vom Typ der reellen Zahlen, so sind solche Größen häufig stetig. Bei der Modellierung der Sequenzialisierung muss diese Stetigkeit demnach modellierbar sein. Entscheidend ist hierbei die Definition von F_2: Da F_2 auf die Historie der Ausgänge Bezug nehmen kann, kann F_2 beispielsweise ohne weiteres eine konstante Fortsetzung der Historie an der Stelle $t = 0$ definieren, was die Stetigkeit impliziert. Durch die Sequenzialisierung entspricht der lokale Zeitpunkt $t = 0$ von F_2 dem globalen Transitionspunkt $t = \tau$.

Die Stetigkeit von Signalen wird aus mehreren Gründen nicht zwingend per Definition für alle Signale gefordert. Zum einen haben nicht alle Signale einen reellen Typ. Zum anderen ist die Stetigkeit teilweise nicht weitreichend genug: Unter Umständen soll neben der Stetigkeit auch die n-fache Differenzierbarkeit erfüllt sein. Aus diesem Grund wurde die Sequenzialisierung so definiert, dass derartige Aspekte von den Ausdrucksmöglichkeiten her modelliert werden *können*, jedoch nicht modelliert werden *müssen*.

3.3.7 Verkettung und Schaltfolgen

Als Verallgemeinerung der Sequenzialisierung sollen nun mehrere, verkettete Sequenzialisierungen genauer untersucht werden. Eine *Verkettung* S wird hierbei definiert als eine endliche oder unendliche Folge von Sequenzialisierungen der Form $S = (F_1 \overset{P_1}{\rightsquigarrow} F_2 \overset{P_2}{\rightsquigarrow} F_3 \overset{P_3}{\rightsquigarrow} \dots)$. Für beliebige Belegungen $i \in \vec{I}$ und $o \in \vec{O}$ mit $S(i, o)$ wird das Verhalten durch die Folge der Komponenten F_i bestimmt, wobei von F_i zu F_{i+1} „umgeschaltet“ wird, sobald P_{i+1} erfüllt ist. Interessant ist demnach die Frage, *wann* genau zwischen den Komponenten für Belegungen (i, o) umschaltet wird: Die unendliche Folge $\langle \tau_0, \tau_1, \tau_2, \dots \rangle$ heißt *Schaltfolge von* S bzgl. der Belegungen (i, o) und ist folgendermaßen für alle $n \geq 0$ definiert:

$$\begin{aligned} \tau_0 &:= 0 \\ \tau_{n+1} &:= \begin{cases} \tau_n + P_{n+1}[(i \cup o) \oplus \tau_n] & \text{falls } \tau_n \neq \infty \text{ und } P_{n+1} \text{ in } S \text{ existiert} \\ \infty & \text{sonst} \end{cases} \end{aligned}$$

Erläuterung: Für jedes Prädikat P_{n+1} ist das zugehörige τ_{n+1} so definiert, dass es dem Schaltpunkt von P_{n+1} unter der Belegung (i, o) entspricht. Wegen der Rechts-Assoziativität von $\rightsquigarrow$ wird für jedes Prädikat P_{n+1} der Koordinatenursprung auf den vorherigen Schaltpunkt τ_n verschoben. τ_n muss zu dem resultierenden „lokalen" Schaltpunkt addiert werden, da τ_{n+1} auf der globalen Zeitachse betrachtet werden soll.

Ist für ein P_n der zugehörige Schaltpunkt τ_n unendlich, d.h., kann die Transition niemals schalten, können entsprechend auch alle folgenden Transitionen niemals schalten, so dass die oben verwendete Fallunterscheidung notwendig ist. Für endliche Verkettungen wird ebenfalls $\tau_n = \infty$ definiert, wenn P_n nicht mehr in S existiert. Zusätzlich wird $\tau_0 = 0$ definiert und entspricht dem initialen Koordinatenursprung.

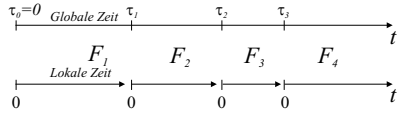

Abbildung 3.5: Verkettung mit Schaltfolge

Die so definierte Schaltfolge gewährleistet, dass jedes F_n das Verhalten von S im entsprechenden Intervall $[\tau_{n-1}, \tau_n)$ festlegt, falls $\tau_{n-1} \neq \infty$ (d.h., wenn der entsprechende Zustand überhaupt betreten wird). Diese Eigenschaft wird durch folgendes Theorem nachgewiesen:

Theorem 3.9. Für alle F_n aus S mit $n \geq 1$ und $\tau_{n-1} \neq \infty$ gibt es ein $\bar{o} =_{<\tau_n} o$ mit $F_n(i \oplus \tau_{n-1}, \bar{o} \oplus \tau_{n-1})$.

Dieses Theorem ist vor allem wichtig, da es garantiert, dass unendliche Sequenzialisierungsfolgen eine definierte Semantik haben, die in jedem Intervall $[\tau_{n-1}, \tau_n)$ nur von der entsprechenden Komponente F_n abhängt. Für die spezielle Verkettung $F_1 \overset{P_1}{\rightsquigarrow} F_2$ gilt nach Definition $\tau_0 = 0$, $\tau_1 = P[i \cup o]$ und $\tau_2 = \tau_3 = \cdots = \infty$. Wie man leicht sieht, spiegelt Theorem 3.9 in diesem Fall genau die Definition der Sequenzialisierung aus Abschnitt 3.3.6 wider, d.h., das Theorem ist die aus der Definition abgeleitete Verallgemeinerung für beliebige Verkettungen.

3.4 Hybride Systeme

Die Theorie der hybriden Systeme wurde in mehreren Arbeiten der letzten Jahre ausführlich beschrieben (vgl. [ACH+95, AH97, Hen96, NSY93]). Die zu Grunde liegende Idee besteht darin, den kontinuierlichen Verlauf eines Systems durch einen endlichen Zustandsautomaten zu beschreiben. Jeder konkrete kontinuierliche Verlauf entspricht einem Lauf durch den Automaten, wobei

der gesamte kontinuierliche Verlauf – entsprechend der Folge der durchlaufenen Zustände – in eine Folge von (Zeit-)Phasen zerlegt wird. Die Transitionen des Automaten bestimmen die Zeitpunkte des Zustands- und damit auch des Phasenwechsels. Innerhalb einer Phase legt der entsprechende Zustand fest, wie der kontinuierliche Verlauf der Phase definiert ist.

Die im Folgenden beschriebene Modellierung unterscheidet sich im Detail aus mehreren Gründen von der Modellierung anderer Autoren. Wichtigster Grund ist die gewünschte Integration mit den stromverarbeitenden Funktionen, die bereits ein semantisches Modell für zeitlich veränderliche Größen und die Verhaltensbeschreibung auf Basis solcher Größen bieten, während die in der Literatur gegebenen Definitionen hybrider Systeme diese Basis implizit modellieren. Weitere Gründe sind die in dieser Arbeit gewählte reale Zeitbasis (vgl. Abschn. 3.1) im Gegensatz zur dichten Interleaving-Zeit anderer Arbeiten, die gewünschte deterministische Semantik der Modellierung des kontinuierlichen Verhaltens und der Focus anderer Ansätze auf der Verwendung hybrider Systeme für die formale Verifikation und lose Systemspezifikation.

Ein hybrides System wird im Kontext der vorliegenden Arbeit wie folgt definiert: Ein Tupel $H = (V, E, src, dest, init, I, O, behavior, cond)$ heißt *hybrides System* und besteht aus folgenden Teilen:

- einem endlichen Automaten $(V, E, src, dest, init)$, der sich in üblicher Weise aus einer endlichen Menge V von Zuständen, einer endlichen Menge E von Transitionen, einer Funktion $src : E \to V$, die jeder Transition den Quellzustand zuordnet, einer Funktion $dest : E \to V$, die jeder Transition den Zielzustand zuordnet, und einem ausgezeichneten initialen Zustand $init \in V$ zusammensetzt.

- einer *Signatur* (I, O), die die Eingangskanäle I und die Ausgangskanäle O festlegt, wobei beide Mengen endlich und disjunkt sind,

- einer Funktion *behavior*, die jedem Zustand $v \in V$ eine Komponente $behavior(v) : \vec{I} \leftrightarrow \vec{O}$ zuordnet, die *Verhalten* von v genannt wird,

- einer Funktion *cond*, die jeder Transition $e \in E$ ein temporales Prädikat $cond(e) : \overrightarrow{I \cup O} \to \mathbb{R} \to \mathbb{B}$ zuordnet, für das die Eingänge O verzögert wirksam sind und das *Condition* von e genannt wird.

Das Verhalten, das während eines einzelnen Zustandes eines hybriden Systems gelten soll, wird nach dieser Definition mit Hilfe von Komponenten beschrieben, die den Zuständen mit Hilfe der Funktion *behavior* zugeordnet werden. Nahtlos in dieses Konzept bettet sich die Darstellung der Transitionsbedingungen mit Hilfe von Prädikaten (vgl. Abschn. 3.3.5) ein, die den Transitionen mit Hilfe der Funktion *cond* zugeordnet werden.

Die obige Definition eines hybriden Systems beschreibt nur den syntaktischen Aufbau. Semantisch betrachtet definiert ein so konstruiertes hybrides System für eine gegebene Initialbelegung einen Lauf durch den Automaten in Form einer Folge von Transitionen und Zuständen, deren Zustände das Verhalten des hybriden Systems in dem jeweiligen Zeitabschnitt bis zum Schalten der nächsten Transition festlegen. Diese Idee entspricht genau der Definition einer Verkettung, wie sie im vergangenen Abschnitt definiert wurde. In diesem Sinne wird die Semantik eines hybriden Systems folgermaßen definiert:

Ein hybrides System H, das wie oben definiert ist, beschreibt semantisch eine Komponente $(\mathsf{hybrid}\, H) : \vec{I} \leftrightarrow \vec{O}$ für die gilt: Seien $i \in \vec{I}$ und $o \in \vec{O}$ beliebig. Dann gilt genau dann $(\mathsf{hybrid}\, H)(i, o)$, wenn es eine endliche oder unendliche Folge φ von Zuständen v_n und Transitionen e_n der Form $\varphi = v_1 \stackrel{e_1}{\to} v_2 \stackrel{e_2}{\to} v_3 \stackrel{e_3}{\to} v_4 \ldots$ mit folgenden Eigenschaften gibt:

1. Die Folge φ bildet einen syntaktisch korrekten Pfad durch den Automaten, der bei *init* beginnt, d.h., es gilt: $v_1 = init$, $src(e_n) = v_n$ und $dest(e_n) = v_{n+1}$.

2. Für die entsprechende endliche oder unendliche Verkettung $\boldsymbol{F}^{\varphi} = F_1 \stackrel{P_1}{\rightsquigarrow} F_2 \stackrel{P_2}{\rightsquigarrow} F_3 \ldots$ mit $F_n = behavior(v_n)$ und $P_n = cond(e_n)$ gilt $\boldsymbol{F}^{\varphi}(i, o)$. Sei $\langle \tau_0, \tau_1, \ldots \rangle$ die zugehörige Schaltfolge.

3. Sei $\varphi' = v'_1 \stackrel{e'_1}{\to} v'_2 \stackrel{e'_2}{\to} v'_3 \ldots$ eine weitere, von φ verschiedene Folge, die obigen Eigenschaften 1 und 2 erfüllt. Sei $\langle \tau'_0, \tau'_1, \ldots \rangle$ die Schaltfolge von $\boldsymbol{F}^{\varphi'}$ bzgl. (i, o). Sei weiterhin $m \geq 1$ der größte Index, für den $\forall\ 1 \leq j < m \bullet e_j = e'_j$ gilt (d.h., φ und φ' stimmen genau bis einschließlich $v_m = v'_m$ überein und unterscheiden sich an der folgenden Transition $e_m \neq e'_m$ oder mind. eine der beiden Transitionen existiert nicht mehr). Dann gilt:

   (a) $\tau_m = \infty \implies \tau'_m = \infty$, d.h., kann φ nicht mehr schalten, so kann auch φ' nicht schalten.

   (b) $\tau_m \neq \infty \implies \tau_m < \tau'_m$, d.h., φ schaltet früher in den Zustand v_{m+1} als φ' in v'_{m+1}.

Die Folge φ repräsentiert nach dieser Definition den Lauf durch den Automaten und ist abhängig von den Eingangsbelegungen i und o. Von allen mögliche Läufen syntaktisch korrekten Läufen ist φ derjenige, der „am schnellsten" schaltet. Gibt es hinter einem Zustand v_m mehrere alternative Transitionen, so muss die in φ gewählte Transition für die Belegungen (i, o) früher schalten als φ'.

Nach dieser Definition ist offensichtlich, dass die Komponente $\mathsf{hybrid}\, H$ nicht für alle Systeme H total. Es ist für bestimmte Systeme mit gegebener Initialbelegung $[i, o]_{\simeq}$ potenziell möglich, dass es keine Folge φ gibt, die

die Eigenschaften 1-3 erfüllt. Folgende Randbedingungen sind jedoch für die Totalität hinreichend.

1. Für alle Transitionen $e_1, e_2 \in E$, die denselben Quellzustand haben ($src(e_1) = src(e_2)$), gilt für alle Belegungen $i \in \vec{I}$ und $o \in \vec{O}$ mit $cond(e_1)[i \cup o] \neq \infty$ auch $cond(e_1)[i \cup o] \neq cond(e_2)[i \cup o]$. Mit anderen Worten: Es gibt keine nicht deterministischen Entscheidungen in Bezug auf schaltende Transitionen, wenn mindestens eine Transition überhaupt nach endlicher Zeit schaltet.

2. Für jeden Zustand v ist das Verhalten $behavior(v)$ total.

Diese beiden Eigenschaften sind hinreichend, um die Totalität eines hybriden Systems zu gewährleisten (ohne Beweis). Sie sind konstruktiv – d.h. während der Modellierung eines hybriden Systems – leicht zu gewährleisten, wenngleich sie nicht automatisiert analysiert werden können (nicht entscheidbar).

3.4.1 Hybride Systeme mit Aktionen

Die Konstruktion der hybriden Systeme kann um die übliche Technik von Aktionen ergänzt werden, die an Transitionen annotiert werden. Aktionen sind nützlich, wenn neben den *zeitkontinuierlichen* auch *zeitdiskrete* Kanäle modelliert werden sollen, d.h. solche Kanäle, deren Werte sich nur an den diskreten Transitionspunkten ändern. Dadurch können beispielsweise Schleifenzähler modelliert werden, die die Anzahl des Durchlaufs eines bestimmten Zustandes zählen. Die Erhöhung des Schleifenzählers findet exakt zum Zustandswechsel statt; innerhalb eines Zustandes bleibt der Wert konstant.

Solche zeitdiskreten Kanäle werden üblicherweise nicht an jeder Transition verändert. Existiert an einer Transition keine Definition für einen zeitdiskreten Kanal, behält er seinen alten Wert bei. Solche Kanäle werden später im Zusammenhang mit der konkreten Testfallmodellierung mit TPT als nichtflüchtige Kanäle bezeichnet (vgl. Abschn. 4.5.1).

Um den Zugriff auf zeitdiskrete Kanäle auch innerhalb des kontinuierlichen Verhaltens zu ermöglichen und um einen Paradigmenwechsel des zu Grunde liegenden Modells für zeitdiskrete Größen zu vermeiden, können Aktionen als Spezialfall von Komponenten definiert werden. Die Idee basiert auf der Tatsache, dass bei einer Sequenzialisierung $F_1 \overset{P}{\rightsquigarrow} F_2$, die zum Zeitpunkt τ von F_1 nach F_2 schaltet, die an die Transition „P“ gebundene Aktion nicht nur zum Zeitpunkt τ gültig ist, sondern auch für alle folgenden Zeitpunkte gültig bleibt. D.h., betrachtet man eine Aktion als Komponente, so gilt sie immer genau parallel zu F_2. Die Besonderheit solcher „Aktions-Komponenten“ ist, dass sie die Verläufe für positive Zeitpunkte konstant halten.

Auf die formale Definition von Aktionen soll in dieser Arbeit verzichtet werden, da sie eine relativ umfangreiche, technische Konstruktion erfordert, bei der parallel zum eigentlichen hybriden System ein zweites hybrides System konstruiert wird, das das Transitionsverhalten implementiert. In jedem Fall bietet die Definition kaum neue Erkenntnisse, sondern formalisiert lediglich einen weiteren, wenngleich mächtigen Operator. Die praktische Bedeutung von Aktionen wird bei der Beschreibung der entsprechenden konkreten Technik in TPT im Abschnitt 4.5 ausführlich erläutert.

3.4.2 Termination

Hybride Systeme schalten zwischen einzelnen Zuständen, können dabei aber bislang nicht terminieren. Für die Modellierung von Testfällen ist es notwendig festzulegen, wann ein Testfall terminiert. Insofern ist es nahe liegend, die hybriden Systeme um *finale Zustände* anzureichern. Erreicht ein Testfall während der Durchführung einen finalen Zustand, so wird dies als Termination des Systems interpretiert.

Finale Zustände werden als echte Teilmenge $V_t \subset V$ der Zustandsmenge V definiert. Für finale Zustände $f \in V_t$ darf es keine ausgehenden Transitionen t mit $src(t) = f$ geben. Ist die Folge φ aus der obigen Definition von $\mathsf{hybrid}\, H$ endlich und ist der letzte Zustand v_{m+1} ein finaler Zustand, dann terminiert das hybride System entsprechend der Transition e_m zum zugehörigen Transitionspunkt τ_m.

Zur formalen Modellierung des Terminationspunktes wird für jede Komponente F und Belegungen (i, o) mit $F(i, o)$ der Terminationspunkt $\sharp(F, i, o)$ definiert. Für hybride Komponenten gilt also $\sharp(\mathsf{hybrid}\, H, i, o) := \tau_m$ (wobei τ_m der im letzten Absatz beschriebene Terminationspunkt ist). Offensichtlich ist diese Definition des Terminationspunktes zeitkausal, da der Zeitpunkt nur von der Belegung der Kanäle bis einschließlich $\sharp(\mathsf{hybrid}\, H, i, o)$ selbst abhängt (ohne Beweis).

Werden hybride Systeme mit finalen Zuständen auf oberster Modellierungsebene verwendet, ist die Terminationssemantik mit Hilfe von $\sharp(F, i, o)$ hinreichend beschrieben. Auf Grund der Kompositionalität der stromverarbeitenden Komponenten kann ein terminierendes System aber auch Teil einer komplexeren Komponente sein, wobei die Terminationssemantik einer Teilkomponente natürlich auch Auswirkung auf die Terminationssemantik der Gesamtkomponente hat. Mit anderen Worten: Durch die Einführung finaler Zustände muss nun auch für die anderen Operatoren definiert werden, wie die Terminationssemantik der gebildeten Komponenten ist.

Aus diesem Grund wird zur Vereinheitlichung der Sichtweise für alle Komponenten die Terminationseigenschaft $\sharp(F, i, o)$ eingeführt. Für Gleichungen ist die Definition der Termination einfach: Eine Gleichung G terminiert niemals, d.h., es gilt $\sharp(G, i, o) := \infty$.

Interessanter ist die Terminationssemantik für die Parallelisierung: Die Termination einer parallelen Komponente $F_1 \parallel F_2$ wird definiert als $\sharp(F_1 \parallel F_2, i, o) := \mathsf{min}(\sharp(F_1, i|_{I_1}, o|_{O_2}, o|_{O_1}), \sharp(F_2, i|_{I_2}, o|_{O_1}, o|_{O_2}))$. D.h., eine parallele Komponente terminiert, sobald mindestens eine ihrer Teilkomponenten terminiert.

Der interessanteste Punkt in Bezug auf die Termination ist die Frage, wie sich bei einer Sequenzialisierung $F = F_1 \overset{P}{\rightsquigarrow} F_2$ (oder allgemeiner: bei einem hybriden System) die Termination der Komponente F_1 auf die Semantik von F auswirkt: Der kritische Fall ist der, bei dem F_1 zu einem Zeitpunkt $\sharp(F_1, i, o)$ terminiert, die Transitionsbedingung P aber erst zu einem späteren Zeitpunkt τ erfüllt ist und somit das Verhalten von F zwischen $\sharp(F_1, i, o)$ und τ von der „terminierten“ Komponente F_1 bestimmt wird. Um diesen Fall zu verhindern, wird für die Semantik der Sequenzialisierung einschränkend gefordert, dass generell für alle Transitionen $\tau \leq \sharp(F_1, i, o)$ gilt. Um zu verhindern, dass diese Einschränkung bei der praktischen Modellierung von hybriden Automaten zu Transitionsbedingungen führt, die die Terminationseigenschaft von F_1 „nachbilden“ müssen, um die obige Ungleichung zu erfüllen, wird in der konkreten Modellierungssprache von TPT ein spezielles Schlüsselwort „`exited`“ eingeführt, das innerhalb von Transitionsbedingungen verwendet werden kann und genau dann wahr ist, wenn der vorherige Zustand terminiert (vgl. Abschn. 4.5.1). Dadurch kann die obige Einschränkung sehr einfach gewährleistet werden.

3.5 Diskretisiertes Verhalten

Bereits bei der Diskussion der Zeitkausalität im Abschnitt 3.3.1 wurde darauf hingewiesen, dass das Verhalten „online“ berechnet werden soll, d.h., das Verhalten einer Komponente soll an den Ausgängen zu einem Zeitpunkt t allein auf Basis der Eingänge bis einschließlich t eindeutig bestimmt werden können. Für die maschinelle Berechenbarkeit ist diese Eigenschaft notwendig, jedoch nicht hinreichend: Die oben beschriebene Modellierung des kontinuierlichen Verhaltens in dichter Zeit hat den entscheidenden Nachteil, dass eine maschinelle Berechnung in diesem Zeitmodell nicht möglich ist.

Bei der Modellierung von Komponenten lag der Schwerpunkt zunächst darauf, dass die intuitive Semantik von kontinuierlichem Verhalten widergespiegelt wird und dass bezüglich der Berechenbarkeit lediglich der (zeit-)kausale Zusammenhang zwischen Ein- und Ausgangsverhalten berücksichtigt wird. Ein wichtiges und zentrales Ziel von TPT ist jedoch die *maschinelle Berechnung* von Testfällen für kontinuierliches Verhalten. Während die intuitive Semantik solcher Testfälle mit Hilfe der bisherigen Definition von stromverarbeitenden Komponenten beschrieben werden kann, ist diese Komponentendefinition wegen des dichten Zeitmodells als Grundlage der ausführbaren

Semantik denkbar ungeeignet.

Aus diesem Grund soll im Folgenden eine diskretisierte Variante der stromverarbeitenden Komponenten definiert werden, bei der die reale Zeitachse in diskrete Zeitabschnitte gleicher Länge der Form $[i\cdot\delta, (i{+}1)\delta)$ (mit $i \in \mathbb{Z}$) zerlegt wird. Jedes diskrete Zeitintervall hat demnach die Länge δ. Die Konstante δ wird auch als *Schrittweite* der Berechnung bezeichnet; die Zeitpunkte $i \cdot \delta$ heißen *Abtastzeitpunkte* der Diskretisierung. Die Idee dieser Zerlegung ist, dass bei der maschinellen Berechnung innerhalb eines Intervalls die Belegungen aller Kanäle konstant sind. Ein *diskreter Strom* $s : \mathbb{R} \to M$ der Schrittweite δ ist demnach ein Strom, für den in jedem Intervall $I = [i \cdot \delta, (i + 1) \cdot \delta)$ gilt: $\forall t, t' \in I \bullet s(t) = s(t')$. Diskrete Ströme abstrahieren also gewissermaßen von dem Verhalten zwischen zwei Abtastzeitpunkten. Diese Definition diskreter Ströme lässt sich auf Belegungen verallgemeinern: Eine *diskrete Belegung* $c \in \vec{C}$ der Schrittweite δ ist eine Belegung, die jedem Kanal $\alpha \in C$ einen diskreten Strom der Schrittweite δ zuordnet. Mit diesen Begriffen lässt

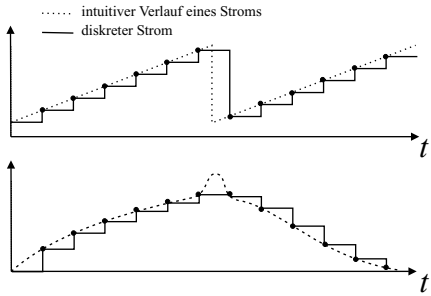

Abbildung 3.6: Beispiele diskreter Ströme

sich nun eine diskrete Komponente folgendermaßen definieren: Eine *diskrete Komponente* $F : \vec{I} \leftrightarrow \vec{O}$ der Schrittweite δ ist eine Komponente mit der zusätzlichen Eigenschaft, dass alle Belegungen i und o, für die $F(i, o)$ gilt, diskrete Belegungen der Schrittweite δ sind.

Nach dieser Definition genügt es offensichtlich, das diskretisierte Verhalten der Eingangsbelegungen zu den Abtastzeitpunkten $n\cdot\delta$ zu beobachten und das Verhalten der Ausgangsbelegungen zu ebendiesen Zeitpunkten festzulegen. Die Belegungen haben für die Zwischenzeitpunkte $t \in (i \cdot \delta, (i+1)\cdot\delta)$, die unmittelbar auf einen Abtastzeitpunkt $i\cdot\delta$ folgen, entsprechend der Definition diskreter Ströme immer denselben Wert wie zum Zeitpunkt $i \cdot \delta$ selbst.

Die Definition der ausführbaren Semantik der Modellierungssprache von TPT wird im Kapitel 4 basierend auf dem Begriff der diskreten Komponenten erläutert. Diskrete Komponenten sind nach der obigen Definition gewöhnliche stromverarbeitende Komponenten mit der speziellen einschränkenden

Eigenschaft, dass sie ausschließlich mit diskreten Belegungen operieren. Die ausführbare, diskretisierte Semantik weicht zwangsläufig von der intuitiven Semantik ab, wenn die intuitive Semantik nicht bereits diskret ist. Diskretisierungsfehler sind dann die Folge. Aus anderen Anwendungsbereichen ist bekannt, dass derartige Diskretisierungs- bzw. Auflösungsfehler zwischen der intuitiven Semantik und dem ausführbaren Modell in der Praxis unvermeidlich sind. In der Fließkomma-Arithmetik wird beispielsweise für den Vergleich zweier Werte f_1 und f_2 üblicherweise der Ausdruck $(\mathtt{abs}(f_1 - f_2) < \varepsilon)$ verwendet, obwohl die zu Grunde liegende Intention die Bedingung $(f_1 = f_2)$ ist. Es soll nun beschrieben werden, welche praktischen Folgen die Zeitdiskretisierung haben kann.

Wertefehler. Entsprechend der Definition diskreter Ströme sind die Werte für alle Zeitpunkte zwischen zwei Abtastzeitpunkten für Eingangs- als auch Ausgangsströme konstant. Eingangsseitig entstehen dadurch Abweichungen ausschließlich in diesen Zwischenräumen, während die Werte der Abtastzeitpunkte immer exakt mit dem intuitiven Modell übereinstimmen (vgl. Abb. 3.7). Ausgangsseitig können sich Wertefehler unter Umständen akkumulieren: Für eine Komponente mit $\mathtt{o(t)} = \int_0^{\mathtt{t}} \mathtt{i(x)dx}$, die eine Eingangsgröße `i` integriert, akkumuliert sich der Fehler von `i`, so dass sich der Fehler mit fortschreitendem t immer weiter verstärkt. Dieses numerische Problem ist jedoch nicht TPT-spezifisch. Es ist aus anderen Bereichen bekannt, wo ebenfalls mit reellwertigen Zahlen operiert wird, und hat in der Praxis dazu geführt, dass bei der Verwendung von Integratoren oder ähnlichen Komponenten die Diskretisierungsfehler einkalkuliert, abgeschätzt oder anderweitig kompensiert werden.

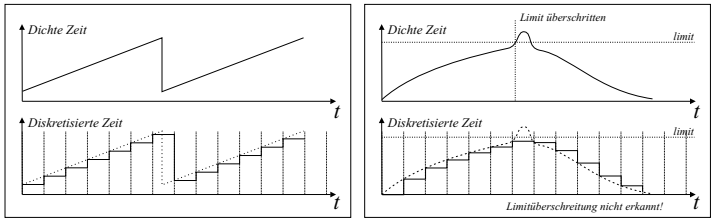

Abbildung 3.7: Fehler bei der diskreten Berechnung

Einfache Zeitfehler. Verändert sich das Verhalten einer Komponente sprunghaft (z.B. durch schaltende Transitionen), so kann es auf Grund der oben beschriebenen Wertefehler zusätzlich zu Zeitfehlern kommen, wenn die Bedingung erst verzögert erkannt wird. Ein Prädikat P mit $P(t) \iff (t \geq 2.1)$ ist für $\delta = 1.0$ beispielsweise erst zum Zeitpunkt $t = 3.0$ erfüllt. Ist P Tran-

sitionsbedingung einer Komponente ($F_1 \overset{P}{\rightsquigarrow} F_2$), bestimmt in diesem Fall F_1 das Verhalten von F an den Ausgängen im Ausführungsmodell entsprechend länger als im intuitiven Modell; der Zeitfehler ist jedoch immer kleiner als δ, kann sich aber durch mehrere in Folge schaltende Transitionen akkumulieren.

Nicht erkennbare Ereignisse. Kritisch sind vor allem solche Fehler, bei denen eine bestimmte Bedingung nur *zwischen* zwei Abtastzeitpunkten erfüllt ist und deshalb nicht erkannt werden kann. Hierdurch kann sich das intuitive vom ausführbaren Verhalten drastisch unterscheiden. Enthält ein hybrides System beispielsweise eine Transition mit der Bedingung ($i(t) \geq limit$), so schaltet die Transition nach dem Ausführungsmodell nicht, wenn die Überschreitung nur zwischen zwei Abtastzeitpunkten gilt (vgl. Abb. 3.7). Um dennoch die Erkennung zu ermöglichen, muss die Bedingung entsprechend abgeschwächt werden, um den Fehler zu kompensieren. Für reellwertige Kanäle f_1 und f_2 kann hierzu anstelle einer Transitionsbedingung ($f_1(t) = f_2(t)$) die bereits erwähnte Bedingung ($\mathsf{abs}(f_1(t) - f_2(t)) < \varepsilon$) verwendet werden.

All diese Diskretisierungsfehler sind unvermeidlich und müssen bei der praktischen Erstellung von Testfällen mit den im folgenden Kapitel beschriebenen Modellierungstechniken einkalkuliert werden. Eine formale Betrachtung der Diskretisierungsfehler ist zwar theoretisch möglich, vereinfacht den praktischen Umgang mit diesen Fehlern jedoch nicht. Aus diesem Grund wird auf eine weitergehende Analyse der Diskretisierungsproblematik im Rahmen dieser Arbeit verzichtet.

3.6 Zusammenfassung

In diesem Kapitel wurde zunächst informell und anschließend formal beschrieben, was im Zusammenhang mit TPT unter kontinuierlichem Verhalten zu verstehen ist. Als Zeitbezug wurde das reale Zeitmodell $\mathbb{R}$ gewählt. Die darauf aufsetzende Modellierung des kontinuierlichen Verhaltens mit Hilfe stromverarbeitender Komponenten bildet deshalb die intuitive Sicht auf das kontinuierliche Verhalten ab, die jedoch nicht ausführbar ist. Als Einschränkung wurden deshalb auch diskrete Komponenten definiert, die im nächsten Kapitel die Grundlage für die Definition der ausführbaren Semantik sind.

Mit den oben beschriebenen Modellierungstechniken für stromverarbeitende Komponenten – insbesondere mit den Konstruktionen für Gleichungen, für die Parallelisierung von Komponenten sowie für hybride Systeme – lässt sich das kontinuierliche Verhalten von Systemen ausdrucksstark und intuitiv beschreiben. Wesentlicher Aspekt ist hierbei die Modularität und Kompositionalität dieser Techniken, die durch die verschiedenen Operatoren möglich wird. Ein komplexes Gesamtsystem lässt sich hierarchisch aus Teilsystemen

zusammensetzen, wobei jedes Teilsystem eine abgeschlossene, klar definierte Semantik hat. Durch die Fusion von hybriden Systemen und stromverarbeitenden Funktionen erhält man unter anderem implizit die Mächtigkeit paralleler Automaten durch Parallelisierung $(\mathsf{hybrid}\ H_1) \parallel (\mathsf{hybrid}\ H_2)$ zweier hybrider Systeme. Weiterhin lassen sich auf Grund der konsistenten semantischen Basis auch hierarchische Automaten modellieren, indem das Verhalten $behavior(v)$ eines Zustands v wiederum durch ein hybrides (Sub-)System modelliert wird.

Die Theorie der stromverarbeitenden Funktionen ist die formale Basis der intuitiven und ausführbaren Semantik von Testfällen, die mit dem Time Partition Testing erstellt werden. Daneben liefert diese Theorie auch ein abstraktes Denkmodell, das den praktischen Umgang mit der Modellierungssprache erleichtern kann.

Kapitel 4

Testfallmodellierung

Wie bereits im Kapitel 1 erläutert wurde, ist die gezielte systematische Auswahl von Testfällen entscheidend für die Anzahl der aufgedeckten Fehler eines gegebenen Testobjekts und damit für die Qualität und Effektivität des Tests. Neben einer systematischen und wohlüberlegten Auswahl von relevanten Testfällen ist hierbei eine zentrale Frage, wie solche Testfälle möglichst genau beschrieben werden können. Unpräzise, widersprüchliche oder missverständliche Testfallspezifikationen bergen die Gefahr von Fehlinterpretationen. Die daraus resultierenden Probleme sind mit denen von fehlerhaften Systemspezifikationen vergleichbar: Ein fehlinterpretierter Test wird anders durchgeführt als ursprünglich intendiert, so dass Fehler im System, die mit diesem Test hätten aufgedeckt werden können, gegebenenfalls unerkannt bleiben.

Trotz ihrer offensichtlichen Bedeutung spielt die Testspezifikation in der Praxis meist eine untergeordnete Rolle. Viel dominanter sind die nachfolgenden Schritte der Testdurchführung, Testauswertung und Testdokumentation, da sich diese Schritte unmittelbar auf die Qualität des Prozesses auswirken und starken Einfluss auf die Praktikabilität und Effizienz des Testvorgehens haben. Ein zentrales und in der Praxis häufig adressiertes Problem ist hierbei die Automatisierung der Testdurchführung, durch die der erforderliche Testaufwand erheblich reduziert wird. Insbesondere für Wiederholungstests und Regressionstests bietet die Automatisierung das Potenzial, Tests ohne nennenswerten Aufwand wiederholt ablaufen zu lassen. Unter diesem Gesichtspunkt ist der Stellenwert der effizienten Testdurchführung in der Praxis durchaus nachvollziehbar.

Um die theoretisch begründeten Qualitätsanforderungen hinsichtlich der systematischen *Auswahl* von Testfällen und die praktischen Anforderungen hinsichtlich der *Automatisierung* gleichermaßen erfüllen zu können, wäre demnach ein Testverfahren wünschenswert, das die systematische Auswahl von

Testfällen unterstützt, wobei jeder definierte Testfall mit Hilfe einer präzisen Beschreibungssprache mit ausführbarer Semantik modelliert wird. Durch Anwendung eines solchen Verfahrens wäre eine hohe Fehleraufdeckungsrate einerseits und ein effizienter Testprozess andererseits garantiert.

TPT hat den Anspruch, genau diese Brücke zwischen Theorie und Praxis des Tests zu schlagen. Die Erläuterung der Vorgehensweise erfolgt hierbei in zwei Schritten. Im folgenden Kapitel wird zunächst die Sprache von TPT beschrieben, mit der *einzelne* Testfälle modelliert werden können. Diese Sprache hat eine ausführbare Semantik, die direkt mit der formalen Semantik der stromverarbeitenden Funktionen des letzten Kapitels korreliert. Anschließend wird im Kapitel 5 auf die systematische Auswahl einer Menge von Testfällen genauer eingegangen.

4.1 Ausführbare Testfälle

Ausführbare Testfälle müssen so modelliert werden, dass sie eine klar definierte ausführbare Semantik haben, um die automatisierte Testdurchführung zu unterstützen. Insofern sind ausführbare Testfälle mit Programmen vergleichbar, da auch diese von einer Maschine ausgeführt bzw. berechnet werden. Gleichwohl besteht, wie bereits im Abschnitt 1.2 erläutert wurde, zwischen Testfällen und Programmen ein wesentlicher Unterschied: Während ein Programm bzw. ein implementiertes System die Aufgabe hat, ein bestimmtes Problem unter *allen* denkbaren Randbedingungen und in verschiedenen Konstellationen zu lösen, beschränkt sich ein Testfall immer nur auf *eine* konkrete Probleminstanz. Hierdurch ist die Komplexität eines Testfalls verglichen mit der Komplexität des zu testenden Systems deutlich geringer. Diese geringe Komplexität ist inhärent für Testfälle, da andernfalls die Wahrscheinlichkeit von Fehlern in den Testfällen genauso groß wäre wie im zu testenden System selbst.

Im Abschnitt 1.4 wurde darauf hingewiesen, dass ein Testfall niemals einzeln definiert wird. Beim Test eines Systems wird immer eine Stichprobe – also eine Menge von Testfällen – aus der Menge aller möglichen Eingabesituationen gewählt, um die Funktionalität des Systems zu prüfen. Je komplexer das System ist, um so mehr Testfälle werden benötigt, um das gesamte Spektrum der Funktionalität abzuprüfen und somit die Qualität des Systems sicherzustellen. Die Komplexität eines einzelnen Testfalls wird durch die wachsende Systemkomplexität hingegen nur wenig oder gar nicht beeinflusst. Dies hat zur Folge, dass vor allem für komplexe Systeme der Schwerpunkt des Tests auf der geeigneten Auswahl und der überschaubaren Modellierung der Menge von Testfällen liegt. Die Modellierungstechnik, mit der die ausführbaren Testfälle beschrieben werden sollen, muss dies berücksichtigen.

Zusammenfassend kristallisieren sich also beim Vergleich zwischen Programmen und ausführbaren Testfällen folgende Kernpunkte heraus:

§1. Ausführbare Testfälle haben genau wie Programme eine ausführbare Semantik, beschreiben also formal berechenbare Funktionen.

§2. Jeder ausführbare Testfall hat eine deutlich geringere Komplexität als das System, das er testet.

§3. Die Anzahl der benötigten Testfälle steigt mit der Komplexität des Systems.

§4. Die Komplexität eines einzelnen Testfalls nimmt mit steigender Komplexität des Systems nur geringfügig oder gar nicht zu.

§5. Bei der Modellierung von Testfällen muss insbesondere der Aspekt der Modellierung der *Stichprobe*, d.h. einer Menge unterschiedlicher Testfälle, mit berücksichtigt werden.

Ausführbare Testorakel. Ausführbare Testfälle ermöglichen die automatisierte Ermittlung des Ausgabeverhaltens $o = s(i)$ eines Systems s bei Eingabe von i im Sinne des Testmodells aus Abschnitt 1.4. Zum Test gehört jedoch auch die Bewertung des Ergebnisses mit Hilfe eines Testorakels r. In der Regel ist diese Bewertung nichttrivial und erfordert bei einer großen Anzahl von Testfällen einen erheblichen Aufwand. Insofern ist neben der automatisierten Testdurchführung auch die Automatisierung des Testorakels im Zuge der Testauswertung erforderlich. Auf die Testauswertung und ihre Automatisierung wird im Kapitel 7 ausführlich eingegangen.

4.2 Modellierungstechniken

Es stellt sich damit die Frage, ob die semantische Gemeinsamkeit von ausführbaren Testfällen und Programmen den Einsatz klassischer Programmiersprachen zur Beschreibung von Testfällen rechtfertigt oder ob ihr Einsatz in diesem Umfeld nicht sinnvoll ist bzw. welche alternativen Techniken eingesetzt werden können. Im folgenden Abschnitt soll zunächst erörtert werden, welche Techniken sich am besten eignen.

Es lässt sich beobachten, dass existierende Testwerkzeuge, die eine automatisierte Testdurchführung unterstützen, meist auf klassische Programmiersprachen zurückgreifen. Solche Sprachen sind zum Beispiel Script-Sprachen wie Visual Basic oder Python, aber auch einfache Batch-Sprachen zur sequenziellen Abarbeitung von Teiltestschritten. Auch die Testsprache TTCN3 [GH99] ist letzten Endes lediglich eine Programmiersprache zur Modellierung ausführbarer Testfälle.

Der Schwerpunkt dieser Techniken liegt also deutlich auf der *Programmierung* und der Automatisierung von Testfällen. Anschaulichkeit und Lesbarkeit der Testfälle sowie der Aspekt der Modellierung einer *Menge* von Testfällen ist häufig unbefriedigend berücksichtigt. Hinzu kommt, dass im Umfeld eingebetteter Systeme die Systementwickler und Domain-Experten meist keine Programmierer sind. Dennoch müssen sie in der Lage sein, Testfälle zu spezifizieren und zu lesen, da sie das gewünschte Verhalten des Systems und die tatsächlich kritischen Fälle am besten kennen. Die Beschreibung von Testfällen muss demzufolge *einfach* möglich sein. Der Einsatz von komplexen Programmiersprachen als einziges zur Verfügung stehendes Ausdrucksmittel ist somit problematisch und führt zu mangelhafter Akzeptanz von derartigen Testverfahren und Testwerkzeugen in der Praxis.

Wesentlich näher liegend ist die Modellierung mit Hilfe grafischer Beschreibungstechniken, da diese meist deutlich intuitiver und damit auch für Nichtprogrammierer einfach anwendbar und lesbar sind. Ein Problem ist in diesem Fall jedoch die gewünschte präzise ausführbare Semantik. Sie erzwingt einen Detaillierungsgrad, der bei der grafischen Modellierung von komplexen Systemen zwangsläufig auch zu sehr komplexen und damit häufig schwer lesbaren grafischen Modellen führt. Bei der konstruktiven Systementwicklung ist der Einsatz grafischer Modellierungstechniken zur Programmierung komplexer Systeme aus diesem Grund durchaus umstritten.

Bei der Testfallmodellierung ist der Einsatz grafischer Techniken hingegen unkritisch, da die Übersichtlichkeit und Lesbarkeit wegen der geringeren Komplexität von Testfällen gewährleistet bleibt (vgl. §2 im letzten Abschnitt). Ermöglicht die grafische Modellierungstechnik auch eine anschauliche und intuitive Darstellung von Testfällen, ist damit auch der Zugang zum Inhalt der Testfälle für Nichtprogrammierer gewährleistet.

Aus diesem Grund verwendet TPT für die Modellierung von Testfällen die grafische Repräsentation von abstrakten Automaten. Jeder Zustand des Automaten beschreibt einen bestimmten zeitlichen Abschnitt des Tests, der *Phase* genannt wird. Die Transitionen des Automaten definieren die möglichen Übergänge von einer Phase zur nächsten. Ein solcher Automat ist einfach lesbar und bietet eine gängige Notation für den Testablauf: Jeder Pfad durch den Automaten definiert einen potenziell möglichen Testablauf des modellierten Tests. Dieser grafischen Technik liegen die im letzten Kapitel eingeführten hybriden Automaten zu Grunde, wie weiter unten noch ausführlicher dargestellt wird.

Die rein grafische Darstellung von Zuständen und Transitionen legt zunächst nur die *Existenz* von Zuständen und die mögliche zeitliche Abfolge der Zustände, nicht aber das *Verhalten* innerhalb der Zustände und die genauen Übergangsbedingungen der Transitionen fest. Für die Definition dieses Verhaltens wird bei Automatennotationen üblicherweise kein grafischer Formalismus eingesetzt, da dies für die Ausdrucksmächtigkeit und die kompakte

Darstellung der Sachverhalte auf dieser Ebene eher hinderlich wäre. Stattdessen werden textbasierte formale Sprachen verwendet. Der Nachteil dieser Sprachen im Vergleich mit grafischen Techniken ist jedoch wiederum, dass sie für Nichtexperten meist schwer verständlich sind.

Um durch die formalen Sprachanteile nicht die intuitive Lesbarkeit der Testmodelle zu zerstören, werden die grafischen Anteile (Zustände und Transitionen) in TPT mit natürlichsprachlichen Texten annotiert, die den zu Grunde liegenden Sachverhalt kompakt und intuitiv leicht verständlich im Klartext umschreiben (vgl. Abb. 4.1). Die grafische Darstellung eines Testablaufs bleibt hierdurch auch für Nichtprogrammierer lesbar und – auf dem Level der Klartextbeschreibungen – sogar modellierbar. Hinter allen Zuständen und Transitionen verbergen sich die formalen Details der Modellierung, die das Verhalten im Detail präzisieren. Diese Details werden in der Regel von Experten modelliert, die mit der formalen Sprache vertraut sind.

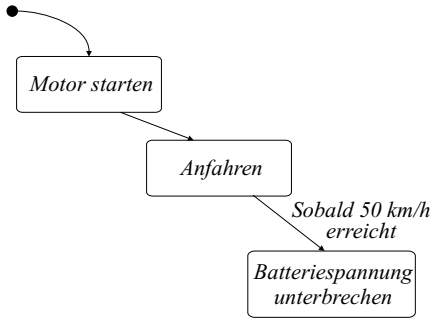

Abbildung 4.1: Grafische Modellierung mit Ablaufautomaten

So gesehen greift TPT sowohl auf grafische Techniken als teilweise auch auf klassische Konzepte von Programmiersprachen zurück. Die klare Trennung zwischen grafischer Darstellung und den dahinter verborgenen formalen Details hat den Vorteil, dass ein intuitives Verständnis des Testablaufs bei Betrachtung der grafischen Darstellung gewährleistet ist. Auf der anderen Seite hat diese Trennung jedoch den Nachteil, dass eine gewisse Disziplin bei der Modellierung notwendig ist, da die Gefahr existiert, dass formale Modellierung und natürlichsprachliche Umschreibung dieser Modellierung in der Grafik inkonsistent werden: Die natürlichsprachliche Intention eines Testfalls und seine formale Durchführungssemantik stimmen dann nicht überein. Dieses Problem ist jedoch inhärent für alle formalen Beschreibungen mit informellen Spezifikationen, da hier die Konsistenzen nicht automatisch geprüft und Widersprüche nicht automatisiert aufgedeckt werden können. Trotz dieses Problems überwiegen die Vorteile der textuellen Annotation aus den oben

genannten Gründen.

Die Modellierungssprache von TPT für Testfälle basiert auf den im Kapitel 3 beschriebenen Techniken der stromverarbeitenden Funktionen und hybriden Systeme. In den folgenden Abschnitten wird diese Modellierungssprache im Einzelnen vorgestellt und semantisch jeweils mit Hilfe der Formalismen des letzten Kapitels definiert.

4.3 Szenarien

Die für TPT verwendete Modellierungstechnik soll kompositional sein. Ein Testfall besteht in der Regel aus einer Folge von einzelnen Phasen, die jeweils unterschiedliche Abschnitte des Gesamttestfalls beschreiben. Eine solche Phasenfolge wird mit Hilfe eines hybriden Automaten beschrieben. Für die Präzisierung dieser Phasen soll es möglich sein, dass sie sich wiederum aus einer Folge von Unterphasen zusammensetzen, so dass ein hierarchischer Automat entsteht. Die einzelnen Bausteine, aus denen sich die so entstehende Hierarchie zusammensetzt, werden bei TPT als *Szenarien* bezeichnet.

So gesehen ist sowohl der Gesamttestfall als auch jede einzelne Phase und Unterphase, aus denen sich der Gesamttestfall zusammensetzt, ein Szenario. Der Gesamttestfall zeichnet sich als spezielles Szenario nur dadurch aus, dass er die Wurzel der Hierarchie ist und unmittelbar einen vollständigen Testfall beschreibt.

4.3.1 Schnittstelle zum System

Um mit TPT möglichst viele unterschiedliche Systeme testen zu können, muss das zu Grunde liegende Beschreibungsmodell universell und abstrakt genug sein, um an die verschiedenen realen Systemumgebungen angepasst werden zu können. Essentielle Voraussetzung eines Systems, dessen kontinuierliches Verhalten getestet werden soll, ist seine klar definierte Schnittstelle. Für TPT wird angenommen, dass diese Schnittstelle flach ist, d.h., es gibt eine (flache) Menge von Eingangsgrößen des Systems und analog eine (flache) Menge von Ausgangsgrößen. Ein Testfall hat die Aufgabe, die Eingangsgrößen mit Daten zu versorgen (*Stimulation*) und die Ausgangsgrößen des Systems zu beobachten (*Observation*). Versteht man einen Testfall als ein auführbares System, so kehren sich die Begrifflichkeiten aus der Sicht des Testfalls um: Jede Eingangsgröße des zu testenden Systems ist Ausgangsgröße des Testfalls; analog ist jede Ausgangsgröße des Systems Eingangsgröße des Testfalls (vgl. Abb. 4.2). Wird im Folgenden von Eingängen bzw. Ausgängen ohne weiteren Kontext gesprochen, ist damit generell die Sichtweise des Testfalls gemeint, d.h., Eingänge sind Observationsgrößen; analog sind unter Ausgängen die Stimulationsgrößen zu verstehen.

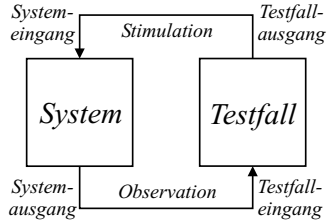

Abbildung 4.2: Wechselwirkung zwischen System und Testfall

Der Kreislauf zwischen dem zu testenden System und dem Testfall muss geschlossen sein, damit ein vollautomatischer Test ohne äußere, zusätzliche Einflussfaktoren möglich ist. Ist ein solcher vollautomatischer Test aus technischen Gründen nicht möglich, kann aber auch ein teilautomatischer Test modelliert werden, bei dem bestimmte Größen automatisiert als Stimulation definiert werden, während andere Größen „manuell“, d.h. außerhalb des automatisierten TPT-Frameworks, beeinflusst werden müssen.

Für die Modellierung der Kommunikation zwischen System und Testfall werden die *Kanäle* aus Kapitel 3 verwendet, die bei der Testfallmodellierung mit Hilfe von eindeutigen Namen (Identifiern) benannt werden.

Die konkrete technische Kopplung zwischen System und Testfall kann je nach Testplattform verschieden ausfallen. Beispiele sind Intraprozesskommunikation (z.B. gemeinsamer Variablenzugriff), Interprozesskommunikation (z.B. Shared Memory), verteilte Kommunikation (z.B. mittels TCP/IP) oder andere Mechanismen (z.B. Debugger-Interfaces für direkten Speicherzugriff eingebetteter Steuergeräte). Hierzu werden für die jeweiligen Testplattformen spezifische, so genannte *Testengines* implementiert, die für die Ausführung mit TPT modellierter Testfälle und den Datenaustausch mit dem Testobjekt (mittels Kanälen) verantwortlich sind. Im Abschnitt 6.3 wird später noch ausführlicher auf die Rolle der Testengines für die Testdurchführung eingegangen.

4.3.2 Deklarationen von Kanälen

TPT fordert die explizite Deklaration aller während der Modellierung eines Testfalls referenzierten und definierten Größen, um Modellierungsfehler (z.B. Tippfehler in Bezeichnern) bei der Kontextanalyse aufdecken zu können. Zu den zu deklarierenden Größen gehören neben den Kanälen auch Konstanten und Funktionen (vgl. Abschn. 4.4). Im Folgenden wird zunächst die Deklaration von Kanälen kurz erläutert.

Der Scope, in dem die deklarierten Kanäle gültig sind, ist immer global

für die gesamte Modellierung eines Testfalls, so dass alle Kanäle zwangsläufig auch global eindeutig benannt sein müssen. Damit genügt für die Deklaration von Kanälen eine zentrale Stelle: Das Testwerkzeug TPT bietet hierzu einen speziellen Editor, in dem die Kanäle deklariert werden können (vgl. Kapitel 8).

Bei der Deklaration wird jedem Kanal ein definierter Typ zugeordnet. Im Abschnitt 3.3 wurde bereits beschrieben, dass jeder Kanal c einen definierten Typ M_c haben muss. Für diesen Typ M_c eines Kanals c gibt es bei TPT die Einschränkung, dass nur folgende primitive Datentypen verwendet werden können:

- `boolean`: Boolscher Datentyp zur Modellierung von binären Signalen (Bits, Flags u.ä.).
- `int`: 64-bit Integer im Wertebereich $[-2^{63}+1, 2^{63}-1]$.
- `float`: Fließkomma-Werte nach der IEEE 754 Norm im Double-Precision-Format (64-Bit)

Alle drei Typen enthalten neben den „üblichen“ Werten jeweils einen zusätzlichen Wert ε, der zur Repräsentation eines nicht definierten Wertes dient, um partielle Funktionen modellieren zu können, wie bereits im Abschnitt 3.3 beschrieben wurde. Für `float`-Daten kann der Wert ε durch `NaN` nach IEEE 754 repräsentiert werden. Bei `boolean`-Daten ist für die Repräsentation mindestens ein zweites Bit notwendig. Da Booleans bei den meisten Rechnerarchitekturen ohnehin durch Bytes repräsentiert werden, ist dies problemlos möglich. Für die Darstellung von ε für `int`-Kanäle wird der Wert -2^{63} (bzw. vorzeichenlos: 2^{63}) verwendet, weswegen der kleinste darstellbare Integerwert um Eins größer ist als in der 64-Bit-Prozessorarithmetik üblich. Diese spezielle Semantik der Werte erfordert deshalb jeweils eine Sonderbehandlung bei arithmetischen Operationen auf diesen Datentypen.

Neben den drei oben beschriebenen primitiven Datentypen werden keine weiteren Typen für Kanäle unterstützt. Die drei Typen zeichnen sich vor allem durch eine natürliche Einbettung ihrer Werte in $\mathbb{R}$ aus: Für `float` und `int` ist diese Einbettung trivial; boolsche Daten werden üblicherweise als Konstanten 0 und 1 abgebildet. Dadurch kann jeder Strom, also jede konkrete Belegung eines Kanals, durch den Graph einer Funktion $\mathbb{R} \to \mathbb{R} \cup \{\varepsilon\}$ veranschaulicht werden. Dies ist sowohl für das visuelle Vorstellungsvermögen während der Modellierung eines Testfalls als auch für die schnelle Erfassbarkeit der aufgezeichneten Ergebnisse während der manuellen Analyse der Testergebnisse wichtig. Diese Behauptung lässt sich auch anhand von Beobachtungen aus der Praxis belegen: Messprotokolle, bei denen Größen über einen bestimmten Zeitraum hinweg erfasst wurden, werden in den meisten Fällen grafisch mit Hilfe von Oszillographen oder ähnlichen Analysewerkzeugen veranschaulicht

und analysiert, die die Daten in Form von Graphen $\mathbb{R} \to \mathbb{R}$ darstellen. Nur für die Analyse im Detail werden gegebenenfalls einzelne Zeit-Werte-Paare betrachtet.

Komplexere Datentypen wie Mengen, Listen oder Zeichenketten erlauben diese anschauliche Repräsentation als Graphen $\mathbb{R} \to \mathbb{R}$ nicht, weswegen die Verwendung dieser Datentypen für Schnittstellengrößen bei der Modellierung von reaktiven Systemen in der Praxis nicht üblich ist. Aus diesem Grund soll auf die Unterstützung derartiger Datentypen für TPT im Sinne einer möglichst einfachen und schlanken Modellierungstechnik verzichtet werden. Die Beschränkung auf die obigen drei Basistypen schränkt zwar theoretisch das Einsatzfeld für TPT ein, in der Praxis werden jedoch für Systeme mit kontinuierlichem Verhalten aus denselben Gründen wie oben beschrieben meistens auch auf der Konstruktionsseite nur einfache arithmetische Datentypen verwendet, so dass die Beschränkung aus Sicht der Praxis unkritisch ist.

4.3.3 Signatur von Szenarien

Im Abschnitt 4.3.1 wurde bereits beschrieben, dass die Kommunikation zwischen dem zu testenden System und dem Gesamttestfall bei TPT mit Hilfe von Kanälen modelliert wird, die die Ein-/Ausgabeschnittstelle bilden.

Analog zum Gesamttestfall haben *alle* Szenarien s definierte Ein- und Ausgabeschnittstellen. Szenarien beschreiben – semantisch betrachtet – stromverarbeitende Komponenten $[\![s]\!] : \vec{I} \leftrightarrow \vec{O}$, wobei I die Menge der Eingangskanäle und O die Menge der Ausgangskanäle ist (vgl. Abschn. 4.3.4). Um die Übersicht über die Abhängigkeiten zwischen den einzelnen Szenarien des Gesamttestfalls zu gewährleisten, verlangt TPT die explizite Deklaration dieser Schnittstelle, die als *Signatur* des Szenarios bezeichnet wird. Die Signatur für Szenarien ist mit der Signatur von Methoden und Funktionen klassischer Programmiersprachen vergleichbar. Auch bei diesen Sprachen ist die explizite Angabe der Schnittstelle mit allen Ein- und Ausgabeparametern zur Dokumentation und zur Verbesserung der statischen Analysemöglichkeiten üblich und sinnvoll.

Neben den Ein- und Ausgängen eines Szenarios unterstützt TPT auch *lokale Kanäle*, die ebenfalls explizit in der Signatur deklariert werden müssen. Lokale Kanäle dienen – wie auch bei Programmiersprachen – zur Definition von Hilfsgrößen, die die Berechnung der Ausgangsgrößen vereinfachen, die jedoch nicht selbst zur Ausgangsschnittstelle gehören. Die lokalen Kanäle sind – semantisch betrachtet – ebenfalls Ausgänge der entsprechenden Komponente, d.h., die Menge der Ausgangskanäle der zugehörigen Komponente entspricht der Vereinigung der Ausgänge und lokalen Kanäle des Szenarios.

Für jedes Szenario müssen demnach drei paarweise disjunkte Mengen von Kanälen festgelegt werden: Eingänge, Ausgänge und lokale Kanäle. Alle in der Signatur referenzierten Kanäle müssen selbstverständlich deklariert sein.

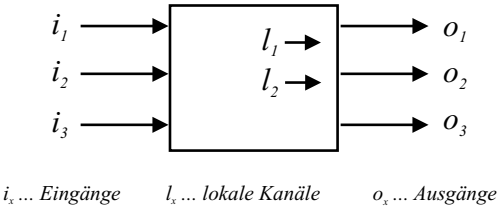

Abbildung 4.3: Signatur eines Szenarios

Die Signatur eines Szenarios mit den drei Mengen von Kanälen kann, wie in Abbildung 4.3 dargestellt, grafisch veranschaulicht werden. Das Werkzeug TPT unterstützt die Definition der Signatur eines Szenarios mit Hilfe eines speziellen grafischen Editors diesem Schema entsprechend (vgl. Kapitel 8).

4.3.4 Intuitive Semantik

Für die Modellierung von Szenarien stehen bei TPT auf beliebigen Hierarchieebenen zwei unterschiedliche Techniken zur Verfügung: Einerseits können Szenarien mit Hilfe der *direkten Definition* in Form einfacher Gleichungssysteme modelliert werden. Diese Technik wird im Abschnitt 4.4 detailliert beschrieben. Andererseits kann das so genannte *Time Partitioning* verwendet werden, um einen Testablauf mit parallelen hybriden Automaten zu modellieren. Diese Technik wird im Abschnitt 4.5 erläutert.

Die intuitive Bedeutung eines Szenarios s mit den Eingängen I, Ausgängen O und lokalen Kanälen L ist – unabhängig von der verwendeten Modellierungstechnik – eine stromverarbeitende Komponente $[\![s]\!] : \vec{I} \leftrightarrow \overrightarrow{O \cup L}$. Die genaue Definition dieser Komponente hängt von den verwendeten Modellierungselementen ab und wird in den Abschnitten 4.4 sowie 4.5 erläutert.

Die intuitive Semantik ist nicht ausführbar. Deshalb wird für Szenarien zusätzlich eine ausführbare Semantik definiert. Intuitive und ausführbare Semantik unterscheiden sich vor allem wegen der Diskretisierungsfehler, die durch die notwendige getaktete Berechnung zu diskreten Abtastzeitpunkten entstehen (vgl. Abschn. 3.5).

4.3.5 Ausführbare Semantik

Die ausführbare Semantik eines Szenarios s lässt sich durch eine diskrete Komponente $[s] : \vec{I} \leftrightarrow \overrightarrow{O \cup L}$ der Schrittweite δ beschreiben. Per Definition von Komponenten ist die Belegung der Ausgänge $O \cup L$ zu einem Zeitpunkt $t = n \cdot \delta$ (mit $n \in \mathbb{N}$) eindeutig bestimmt, wenn die Eingänge I für alle $t' \leq t$ und die Ausgänge $O \cup L$ für alle $t' < t$ gegeben sind.

Aus diesem Grund lässt sich die ausführbare Semantik eines Szenarios dadurch beschreiben, dass für jeden Abtastzeitpunkt $t = n \cdot \delta$ die Belegung der Ausgänge in Abhängigkeit der gegebenen Belegungen der Eingänge (für $t' \leq t$) und der Ausgänge (für $t' < t$) definiert wird. Mit der Festlegung der Ausgangsbelegung zum Abtastzeitpunkt t sind nach Definition der diskreten Komponenten (vgl. Abschn. 3.5) auch die Belegungen für alle nachfolgenden Zeitpunkte im Intervall $(t, t + \delta)$ bestimmt. Dadurch genügt es für die ausführbare Semantik, iterativ die Belegung zu allen Zeitpunkten $t = n \cdot \delta$ zu berechnen. Diese iterative Berechnung wird für einen Gesamttestfall nach folgendem Ausführungsmodell durchgeführt:

Ausführungsmodell eines Gesamttestfalls. Beschreibt ein Szenario s einen Gesamttestfall, so legt das Szenario s mit seiner ausführbaren Semantik $[s]$ das Verhalten der Ausgänge zu einem einzelnen Abtastzeitpunkt $n \cdot \delta$ fest. Für die Berechnung des Verhaltens über der gesamten Zeit dient folgendes Ausführungsmodell:

1. Die Berechnung beginnt beim Zeitpunkt $t = 0$ und erfolgt in äquidistanten Schritten der Schrittweite δ. Initial werden die Belegungen der Kanäle I, O und L für alle $t < 0$ als ε angenommen, d.h., die Werte der Vergangenheit sind initial per definitionem undefiniert, da die Werte nicht bekannt sind. Dann wird der Zähler n auf 0 gesetzt ($n := 0$) und mit Schritt 2 fortgefahren.

2. Der aktuelle Zeitpunkt der Berechung ist $t := n \cdot \delta$. Für diesen Zeitpunkt werden die Belegungen der Eingänge I vom Kontext des Szenarios, d.h. vom zu testenden System, eingelesen.

3. Basierend auf den Belegungen der Kanäle I (für $t' \leq t$) sowie der Ausgänge $O \cup L$ (für $t' < t$) wird nun die Belegung der Ausgänge zum Zeitpunkt t errechnet. Als Ergebnis dieser Berechnung stehen entweder die Belegungen der Ausgänge $O \cup L$ fest, oder das Szenario signalisiert seine Termination, d.h., es gilt $\sharp(s, i, o) = t$. In letzterem Falle ist die Durchführung des Testfalls damit beendet.

4. Ist das Szenario noch nicht terminiert, wird der Zähler n um Eins inkrementiert ($n := n+1$). Die Berechnung wird anschließend mit Schritt 2 wiederholt.

Nach diesem Ausführungsmodell genügt es für die Repräsentation der Belegung eines Kanals, die Folge der Werte der Kanäle zu den Abtastzeitpunkten $n \cdot \delta$ zu verwalten. Da die Berechnung des Verhaltens immer bei $t = 0$ beginnt und die Ströme unterhalb dieses Nullpunktes per Definition undefiniert sind, ist diese Repräsentation als Folge zu jedem Berechnungszeitpunkt t endlich.

Entsprechend der obigen Charakterisierung ermöglicht die ausführbare Semantik trotz der notwendigen Zeitdiskretisierung den Umgang mit der dichten Zeit und basiert damit auf demselben zu Grunde liegenden Modell wie die intuitive Semantik: den stromverarbeitenden Komponenten. Wie im Abschnitt 3.5 diskutiert, ist der Unterschied der beiden Semantiken offenbar nur in den Diskretisierungsfehlern begründet. Ein formaler Zusammenhang zwischen beiden Semantiken wird in dieser Arbeit nicht hergestellt.

4.4 Direkte Definition

Ist die Signatur eines Szenarios s mit den Eingängen I, Ausgängen O und lokalen Kanälen L festgelegt, muss das Verhalten des Szenarios modelliert werden. Die beiden Techniken, die TPT hierfür bietet, sind die so genannte *direkte Definition* sowie das *Time Partitioning*. In diesem Abschnitt wird zunächst die direkte Definition vorgestellt.

Bei der direkten Definition werden die Werte der Kanäle $O \cup L$ (Ausgangs- und lokale Kanäle) in Abhängigkeit von den Belegungen der Eingangskanäle mit Hilfe eines einfachen Gleichungssystems modelliert. Dieses Gleichungssystem besteht für ein Szenario s syntaktisch aus einer Folge von Gleichungen $g_{c_1}, g_{c_2}, \ldots, g_{c_n}$, wobei für jeden Kanal $c_i \in O \cup L$ genau eine entsprechende Gleichung g_{c_i} der Form $c_i(t) = E_i$ in der Folge existieren muss.

$$\begin{array}{lcl} c_1(\mathtt{t}) & = & E_1 \\ c_2(\mathtt{t}) & = & E_2 \\ \ldots & & \\ c_n(\mathtt{t}) & = & E_n \end{array}$$

Beispiel: Seien `a` und `b` Eingangskanäle eines Szenarios, `x` und `y` Ausgangskanäle sowie `l` ein lokaler Kanal. Dann könnte eine direkte Definition wie folgt lauten:[1]

```
x(t) = (t < 2) ? a(t) : x(t-2) /* zyklisch */
l(t) = b(t) + 2
y(t) = l(t/2) * sin(2 * t)
```

In diesem Beispielszenario gibt es jeweils genau eine Gleichung für die Ausgangskanäle `x` und `y` sowie den lokalen Kanal `l`. Die rechte Seite einer Gleichung $c(\mathtt{t}) = E$ ist ein Ausdruck, der zu jedem Zeitpunkt t den entsprechenden Wert für die Belegung von c festlegt. Die Ströme der Kanäle werden also explizit punktweise definiert.

[1] Die Bedeutung der im Beispiel verwendeten Operatoren wird im Abschnitt 4.4.3 beschrieben.

In einem solchen Ausdruck E kann unter anderem auch Bezug auf alle in s bekannten Kanäle $A := I \cup O \cup L$ genommen werden. Man beachte hierbei, dass dies auch für den durch die Gleichung g_c definierten Kanal c selbst zutrifft, d.h., es darf im Ausdruck auch auf den durch die Gleichung definierten Kanal c Bezug genommen werden, wobei die verzögerte Wirksamkeit gewährleistet werden muss (vgl. Abschn. 3.3.4). So wird in der Gleichung `x(t) = (t < 2) ? a(t) : x(t-2)` beispielsweise auf den Wert von `x` zum Zeitpunkt $\mathtt{t} - 2$ referenziert, um ein zyklisches Signal mit der Periodendauer 2 modellieren zu können.

4.4.1 Intuitive Semantik

Der Ausdruck E einer Gleichung g_c der Form $c(\mathtt{t}) = E$ beschreibt semantisch einen stromverarbeitenden Ausdruck $[\![E]\!] : \vec{A} \to \mathbb{R} \to M_c$ vom Typ M_c, für den der Kanal c verzögert wirksam sein muss (vgl. Abschn. 3.3.4). Nach Definition eines stromverarbeitenden Ausdrucks darf der Wert des Ausdrucks zu einem Zeitpunkt t also nur von den Belegungen der Kanäle A bis einschließlich t abhängen. Die konkrete Semantik hängt natürlich von dem jeweiligen konkreten Ausdruck ab. Die möglichen Formen von Ausdrücken werden zusammen mit ihrer Semantik im Abschnitt 4.4.3 erläutert.

Die gesamte Gleichung g_c weist den vom Ausdruck E bestimmten Wert $y = [\![E]\!](a)(t)$ dem Kanal c zum Zeitpunkt t zu und beschreibt damit semantisch die stromverarbeitende Gleichung $[\![g_c]\!] : \vec{A_{\setminus c}} \leftrightarrow \{\vec{c}\}$ für den Ausdruck $[\![E]\!]$ mit $A_{\setminus c} := A \setminus \{c\}$ (vgl. Abschn. 3.3.4). Die intuitive Semantik einer Gleichung g_c der Form $c(t) = E$ ist demnach eine Komponente mit den Eingängen $A_{\setminus c}$ und dem Ausgang c. Das Verhalten dieser Komponente wird allein durch den Ausdruck E bestimmt. Im Abschnitt 4.4.3 wird ausführlich erläutert, wie ein solcher Ausdruck syntaktisch definiert werden kann und welche Semantik er hat.

Intuitive Semantik des Gleichungssystems. Betrachtet man nun alle Gleichungen des Gleichungssystems gemeinsam, so gelten diese zeitlich parallel, d.h. im selben Zeitbereich. Folglich kann das gesamte Gleichungssystem des Szenarios s semantisch als Parallelisierung der Komponenten der enthaltenen Gleichungen definiert werden. Formal kann die intuitive Semantik der direkten Definition deshalb sehr einfach folgendermaßen definiert werden:

$$[\![s]\!] := [\![g_{c_1}]\!] \parallel [\![g_{c_2}]\!] \cdots \parallel [\![g_{c_n}]\!]$$

4.4.2 Ausführbare Semantik

Die ausführbare Semantik eines Ausdrucks E ist ebenfalls ein stromverarbeitender Ausdruck $[E] : \vec{A} \to \mathbb{R} \to M_c$, für den c verzögert wirksam sein

muss.

Die ausführbare Semantik einer Gleichung g_c der Form $c(t) = E$ ist eine diskrete Komponente $[g_c] : \vec{A_{\setminus c}} \leftrightarrow \{\vec{c}\}$ der Schrittweite δ, für die für alle diskreten Belegungen i und o genau dann $[g_c](i, o)$ gilt, wenn

$$\forall n \in \mathbb{N} \bullet [E](i \cup o)(n \cdot \delta) = o_c(n \cdot \delta)$$

gilt. Das heißt, die Belegung des Ausgangs c stimmt zu den Abtastzeitpunkten mit dem Wert des Ausdrucks E überein. Die dazwischen liegenden Werte sind damit nach Definition diskreter Belegungen eindeutig.

Um dieses ausführbare Verhalten berechnen zu können, genügt es demnach, iterativ zu den Abtastzeitpunkten $t = n \cdot \delta$ $(n \in \mathbb{N})$ den Wert $y = [E](a)(t)$ zu berechnen; dieser Wert entspricht der Belegung des Ausgangs c zum Zeitpunkt t.

Ausführbare Semantik des Gleichungssystems. Die ausführbare Semantik des gesamten Szenarios ist analog zur Definition der intuitiven Semantik als Parallelisierung

$$[s] := [g_{c_1}] \parallel [g_{c_2}] \cdots \parallel [g_{c_n}]$$

aller Gleichungen mit folgender Einschränkung definiert: Wie bereits erwähnt, können die Gleichungen wechselseitig auf Ausgangskanäle anderer Gleichungen Bezug nehmen. Insgesamt muss hierbei gewährleistet sein, dass die unmittelbaren Wirksamkeiten der Ausgangskanäle im Gleichungssystem azyklisch sind. D.h., ist ein Ausgangskanal c_j einer Gleichung g_{c_j} unmittelbar wirksam bzgl. einer Gleichung g_{c_i}, so muss der Ausgangskanal c_i transitiv über alle Gleichungen verzögert wirksam bzgl. g_{c_j} sein. Mit anderen Worten: Es muss in diesem Fall möglich sein, c_j eindeutig zu berechnen, bevor c_i berechnet wurde. Ohne diese Forderung der azyklischen Abhängigkeiten wäre eine Berechnung des Gleichungssystems durch einfache Termauswertung unmöglich. Stattdessen wäre ein algebraischer Gleichungslöser erforderlich, der zwar einerseits die Ausdrucksmächtigkeit erhöhen, aber im Allgemeinen sowohl die Eindeutigkeit einer Lösung als auch die Echtzeitfähigkeit verhindern würde.

Im folgenden Beispiel kann zu einem Zeitpunkt t die erste Gleichung vor der zweiten Gleichung eindeutig berechnet werden, da `b` für die erste Gleichung verzögert wirksam ist:

```
a(t) = b(t-1.0) // 'b' ist verzögert wirksam
b(t) = a(t)     // 'a' ist unmittelbar wirksam
```

Im zweiten Beispiel sind die beiden Ausgangskanäle der Gleichungen wechselseitig unmittelbar wirksam (d.h. nicht verzögert wirksam). Eine sequentielle Berechnung der Ausgangskanäle ist nicht möglich. (Im Beispiel gibt es eine unendliche Lösungsmenge.) Solche Zyklen können i. Allg. erst zur Laufzeit und nicht während der Kontextanalyse erkannt werden und führen zu einem Laufzeitfehler bei der Berechnung.

```
a(t) = b(t)-1.0 // 'b' ist unmittelbar wirksam
b(t) = a(t)     // 'a' ist unmittelbar wirksam
```

Damit bleibt sowohl für die intuitive als auch die ausführbare Semantik nur noch zu klären, welche Ausdrücke E es gibt und wie ihre intuitive Semantik $[\![E]\!]$ bzw. ihre ausführbare Semantik $[E]$ definiert ist. Dies wird im folgenden Abschnitt erläutert.

4.4.3 Ausdrücke für Gleichungen

Ein Ausdruck E beschreibt semantisch eine Funktion $[\![E]\!] : \vec{A} \to \mathbb{R} \to M_c$ bzw. $[E] : \vec{A} \to \mathbb{R} \to M_c$, die für eine gegebene Belegung der Kanäle A und einen gegebenen Zeitpunkt t den Wert $[\![E]\!](a)(t)$ bzw. $[E](a)(t)$ des Ausdrucks vom Typ M_c festlegt. Im Folgenden sollen die unterschiedlichen Operatoren erläutert werden, mit denen Ausdrücke gebildet werden können.

Da TPT nur die elementaren Datentypen `int`, `float` und `boolean` unterstützt (vgl. Abschn. 4.3.2), beschränken sich die Operatoren folglich auf arithmetische und logische Operatoren. Die Ausdruckssprache von TPT orientiert sich stark an der Syntax von C, um einem möglichst großen Anwenderkreis den Einsatz von TPT zu erleichtern: Die Sprache C wird vor allem im Umfeld von eingebetteten Steuerungssystemen in wachsendem Umfang auch von Ingenieuren anderer Fachrichtungen beherrscht. Auch kommerzielle Modellierungswerkzeuge wie Matlab/Simulink/Stateflow orientieren sich bei den textuellen Modellanteilen an der Syntax und Semantik von C.

Eine detaillierte Beschreibung der Syntax der Ausdruckssprache von TPT ist im Anhang B.1 enthalten. Im Folgenden werden vorrangig die von C abweichenden Konstrukte anhand von Beispielen erläutert. Hierfür werden die folgenden Kanäle verwendet: i und j sind Kanäle vom Typ `int`; x, y und z sind `float`-Kanäle; p ist ein boolscher Kanal. Es wird vorausgesetzt, dass diese Kanäle bereits deklariert sind.

Ausdrücke aus C

Die Formen von Ausdrücken, die mehr oder weniger direkt aus C übernommen wurden, werden im Folgenden nur kurz aufgelistet.

- **Konstante Werte:** Es werden numerische Konstanten (z.B. `2`, `−2.7`, `+3`) als auch boolsche Konstanten (`true` und `false`) unterstützt. Die beiden boolschen Konstanten sind in TPT – anders als in C – Schlüsselworte der Sprache.
- **Referenz auf benannte Konstanten:** Konstanten können in TPT – ähnlich wie Kanäle – global definiert und anschließend beliebig in Ausdrücken referenziert werden. Ihre Namen müssen global eindeutig

sein, d.h., es gibt einen gemeinsamen globalen Namensraum für Kanäle und Konstanten. Jede benannte Konstante hat einen Typ (`int`, `float`, `boolean`) und einen definierten Wert.

- **Typkonvertierung (type cast):** Bei TPT wird – anders als in C – immer eine explizite Typkonvertierung gefordert, um die für C typischen Fehler durch die implizite Typumwandlung zu vermeiden. Die Syntax ist identisch. Die detaillierte Beschreibung zur Werteabbildung bei Casts ist im Anhang B.1 auf Seite 158 zu finden.

- **Bedingte Ausdrücke:** Der aus C bekannte `(_)?_:_` -Operator steht auch in TPT zur Verfügung, um einfache Fallunterscheidungen innerhalb von Ausdrücken beschreiben zu können.

- **Arithmetische Operatoren:** Es stehen die üblichen binären Operatoren `+`, `-`, `*`, `/` sowie die unären Vorzeichen `+`, `-` zur Verfügung.

- **Logische und Vergleichsoperatoren:** Es stehen die in C üblichen binären Operatoren `&&`, `||`, `<`, `>`, `<=`, `>=`, `==` und `!=` sowie die unäre boolsche Negation `!` zur Verfügung.

Die Semantik der obigen Ausdrucksformen ist trivial. Beispielsweise gilt $[\![X\texttt{+}Y]\!](a)(t) = [\![X]\!](a)(t) + [\![Y]\!](a)(t)$ sowie $[\![\texttt{2}]\!](a)(t) = 2$ für beliebige Eingangsbelegungen $a \in \vec{A}$ und Zeitpunkte t. Analog gilt diese Definition für die ausführbare Semantik $[X\texttt{+}Y]$ und $[\texttt{2}]$.

Im Folgenden werden die zusätzlichen Formen von Ausdrücken ausführlich vorgestellt, die für TPT spezifisch sind und in klassischen Programmiersprachen nicht existieren.

Aktueller Zeitpunkt

Ein besonderer Ausdruck ist das Schlüsselwort „`t`“, das den aktuellen Zeitpunkt beschreibt, zu dem der Ausdruck ausgewertet wird. Der Ausdruck „`t`“ hat den Typ `float`. Zeitwerte werden in TPT generell durch Werte vom Typ `float` repräsentiert. Semantisch beschreibt der Ausdruck `t` intuitiv den stromverarbeitenden Ausdruck $[\![\texttt{t}]\!] : \vec{A} \to \mathbb{R} \to M_c$ mit $[\![\texttt{t}]\!](a)(t) = t$ für alle Belegungen $a \in \vec{A}$ und Zeitpunkte t. Die ausführbare Semantik ist entsprechend definiert, d.h., es gilt analog $[\texttt{t}](a)(t) = t$.

Schrittweite

Die ausführbare Semantik von TPT basiert auf einer konstanten äquidistanten Schrittweite δ, mit der die Simulationszeit von einem Berechnungsschritt zum nächsten weiterzählt. Die Tatsache, dass die Berechnung nur quasi-kontinuierlich abläuft, ist für die Modellierung von Testfällen weitestgehend irrelevant,

wenngleich die resultierenden Diskretisierungsfehler natürlich berücksichtigt werden müssen. Darüber hinaus gibt es jedoch einige wenige Ausnahmen, bei denen die Berechnungsschrittweite explizit in der Modellierung verwendet wird. Hierfür gibt es den speziellen Ausdruck „@“, der die konstante Schrittweite δ der Berechnung repräsentiert und den Typ `float` hat. Die ausführbare Semantik dieses Ausdrucks ist $[@](a)(t) := \delta$.

Diese Schrittweite „@“ wird bei der Testfallmodellierung zum einen benötigt, um Diskretisierungsfehler der Berechnung genau abschätzen zu können (vgl. Abschn. 3.5). In der Definitionsgleichung von $p(t)$ im unten stehenden Beispiel wird die Grenze `LIMIT` in Abhängigkeit von der Schrittweite nach unten korrigiert. Hat der Verlauf von x den maximalen Anstieg 0.1, ist dadurch sichergestellt, dass p garantiert `true` ist, wenn x den Grenzwert `LIMIT` überschreitet.

Ein anderes Beispiel, bei dem die Schrittweite zwischen den Berechnungszeitpunkten benötigt wird, ist die Modellierung der so genannten *stetigen Anknüpfungen*, wie sie in dem unten stehenden Beispiel bei der Definition von $y(t)$ und $z(t)$ verwendet wurde (vgl. auch Abschn. 3.3.6):

```
p(t) = x(t) > (LIMIT - 0.1 * @)
y(t) = y(-@)
z(t) = z(-@) + 0.1 * sin(t)
```

In der Gleichung für $\mathtt{y}(t)$ wird im Ausdruck auf der rechten Seite auf den Wert des Kanals `y` zum Zeitpunkt -@ zugegriffen. Dieser Zeitpunkt entspricht in der ausführbaren Semantik dem letzten Berechnungszeitpunkt vor $t = 0$ (also $t =$ -δ). Bei hybriden Automaten, die für jeden Zustand eine lokale Uhr verwenden (vgl. Abschn. 3.4), bedeutet dies, dass gezielt auf den letzten berechneten Wert *vor* dem Betreten der aktuellen Phase zugegriffen werden kann. Dieser Zugriff ist notwendig, um innerhalb einer Phase eine *stetige Anknüpfung* an den vorherigen Verlauf modellieren zu können. Die obige Modellierung von `y` beschreibt also die konstante Fortsetzung des bisherigen Verlaufs von `y`, wobei bei $t = 0$ ein stetiger Übergang garantiert ist – unabhängig davon, welchen Wert der Kanal y vor Betreten der aktuellen Phase hatte (vgl. Abb. 4.4).

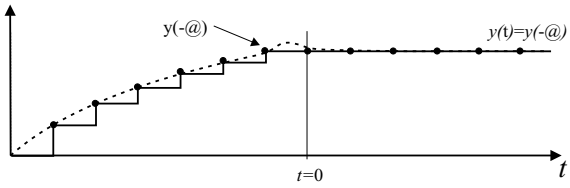

Abbildung 4.4: Stetige Anknüpfung

Analog ist `z` so definiert, dass der Verlauf bei $t = 0$ stetig an den bisherigen Verlauf anknüpft und dann mit einer Sinusfunktion um diesen Wert schwingt.

Aus intuitiver Sicht gibt es keine Berechnungsschrittweite δ. Insofern wird für die intuitive Semantik des Ausdrucks „@“ eine „künstliche Schrittweite“ δ eingeführt, deren Wert gegen Null konvergiert. Mit anderen Worten: Die ausführbare Semantik ist $[\![@]\!](a)(t) := \delta$ und für einen Ausdruck E, in dem „@“ vorkommt, wird für die intuitive Semantik der Grenzwert $\mathsf{lim}_{\delta \to +0}\ [\![E]\!](a)(t)$ gebildet. Man beachte, dass die Existenz eines solchen Grenzwertes und damit auch die Existenz der intuitiven Semantik nicht in jedem Fall garantiert ist. Ein Beispiel – wenngleich ohne praktische Relevanz – ist der Ausdruck $sin(1/@)$. Dieser Ausdruck hat eine ausführbare, aber keine intuitive Semantik. Diese Tatsache resultiert aus der „Näherungsrechnung“ der ausführbaren Semantik, durch die ein definierter Wert ohne Bedeutung im dichten Zeitmodell entsteht.

Die intuitive Bedeutung der oben aufgeführten Gleichung für $p(t)$ ist entsprechend der Grenzwertbildung $p(t) = x(t) > \texttt{LIMIT}$. Die Definition für $y(t)$ entspricht intuitiv dem Grenzwert $y(t) = \mathsf{lim}_{\delta \to +0}\ y(-\delta)$, dessen Existenz i. Allg. nicht garantiert ist. Entsprechendes gilt auch für $z(t)$.

Zugriff auf Kanäle zu beliebigen Zeitpunkten

Wie bereits erwähnt, kann innerhalb von Ausdrücken auf Werte sämtlicher Kanäle A lesend zugegriffen werden. Da Kanäle bei TPT – semantisch betrachtet – Ströme repräsentieren, ein Ausdruck aber immer einen konkreten, einzelnen Wert vom Typ `int`, `float` oder `boolean` beschreibt, ist der Zugriff auf einen Kanal innerhalb eines Ausdrucks nur unter Angabe eines Zeitpunktes möglich, zu dem der Wert des Kanals ermittelt werden soll. Syntaktisch wird der Zugriff auf einen Kanal in der Form $d(timeExpr)$ notiert, wobei d der Name des Kanals und $timeExpr$ ein Ausdruck vom Typ `float` ist, dessen Wert den Zeitpunkt beschreibt, zu dem auf den Kanal zugegriffen werden soll.

Beispiele:

```
y(t) = x(t)
z(t) = x(t-1.0)
```

In der obigen Gleichungen wird der Kanal y zum Zeitpunkt $t \geq 0$ identisch zum Wert von x zum Zeitpunkt t definiert. Die Ströme von x und y stimmen also für nicht negative Zeitpunkte überein. z hingegen entspricht dem um 1.0 verzögerten Verlauf von x (für nicht negative Zeitpunkte).

Die intuitive Semantik eines Ausdrucks der Form $d(timeExpr)$ ist ein stromverarbeitenden Ausdruck $[\![d(timeExpr)]\!] : \vec{A} \to \mathbb{R} \to M_d$ mit der Eigenschaft

$$[\![d(timeExpr)]\!](a)(t) := a_d(\ [\![timeExpr]\!](a)(t)\ )$$

Die ausführbare Semantik ist analog als

$$[d(timeExpr)](a)(t) := a_d(\ [timeExpr](a)(t)\ )$$

definiert. Der Kanal d muss nach dieser Definition zwangsläufig in A enthalten sein, damit der Strom a_d existiert. D.h., es können sinnvollerweise nur Kanäle referenziert werden, die in der Signatur vom Szenario s enthalten sind. Zusätzlich muss der Zeitausdruck $timeExpr$ offensichtlich einen Wert kleiner oder gleich t liefern (d.h. $[\![timeExpr]\!](a)(t) \leq t$), damit auf keinen Wert eines Kanals „in der Zukunkt“ $t' > t$ zugegriffen wird und somit die Zeitkausalität des stromverarbeitenden Ausdrucks verletzt ist.

Die Eigenschaft $d \in A$ kann während der Kontextanalyse automatisch nachgewiesen werden, da A endlich ist. Problematisch ist jedoch der Nachweis $[\![timeExpr]\!](a)(t) \leq t$. Ein automatisierter statischer Nachweis der Eigenschaft ist ohne syntaktische Einschränkung des Ausdrucks $timeExpr$ praktisch nicht möglich. Durch syntaktische Einschränkungen ist aber bereits ein einfacher Ausdruck wie $d(t - delay(t))$, wobei $delay$ ein Eingangskanal ist, der zu jedem Zeitpunkt den Verzögerungsoffset für den Zugriff auf d festlegt, nicht mehr realisierbar; es sei denn, aus dem Kontext ist automatisch deduzierbar, dass $delay(t) \geq 0$ gilt, was in der Regel nicht realistisch ist.

Dies bedeutet, dass eine sinnvolle Einschränkung des Arguments $timeExpr$, die eine statische Analyse der Bedingung $[\![timeExpr]\!](a)(t) \leq t$ ermöglicht und die Ausdruckmächtigkeit nicht behindert, praktisch nicht möglich ist. Stattdessen wird die Eigenschaft während der Laufzeit – also zum Zeitpunkt der Testdurchführung – analysiert. Dies kann gänzlich ohne Einschränkungen der Ausdrucksmächtigkeit des Zeitarguments geschehen. Wird die Bedingung zur Laufzeit verletzt, führt dies in diesem Fall zu einem Laufzeitfehler und damit zum Testabbruch.

Aus diesen Gründen ist es unvermeidlich, auch bei der automatisierten Testdurchführung – ähnlich wie bei der Ausführung gewöhnlicher Programme – mit Laufzeitfehlern umzugehen bzw. mit Laufzeitfehlern zu rechnen. Diese Notwendigkeit hängt jedoch nicht ausschließlich mit dem Zugriff auf Kanäle zu beliebigen Zeitpunkten zusammen. Auch der klassische Laufzeitfehler der Division durch Null kann bei der automatisierten Testdurchführung mit TPT auftreten.

Zyklische Signale. Ein nützliches Beispiel der Referenz auf die Historie eines Kanals `x` im Ausdruck E der Gleichung $\mathtt{x(t)} = E$ ist die Definition zyklischer Signale.

```
x(t) = (t < 1.0) ? (2 * t) : x(t-1.0)
```

In diesem Beispiel wird `x` bis ausschließlich zum Zeitpunkt 1.0 als lineare Funktion definiert. Anschließend wiederholt sich dieser lineare Anstieg mit der Periodendauer 1.0. Es entsteht also eine Sägezahnschwingung mit der Amplitude 2.0 und der Periodendauer 1.0.

Zugriff auf den letzten Wert eines Kanals

Eine spezielle Form des Zugriffs auf einen Kanal ist der Zugriff auf den *letzten Wert* des Kanals. Hierbei wird kein expliziter Zeitpunkt angegeben, zu dem der Wert gelesen werden soll. Der referenzierte Zeitpunkt ergibt sich stattdessen implizit aus dem aktuellen Zeitpunkt der Berechnung als $t - @$.

Wird innerhalb eines Ausdrucks auf einen Kanal d ohne Angabe eines Zeitpunktes referenziert, so entspricht dies dem Ausdruck $d(t - @)$. Insofern ist der Ausdruck d nur eine syntaktische Abkürzung für den Zugriff auf den letzten diskreten Zeitpunkt vor dem aktuellen Zeitpunkt. Die intuitive Semantik des Ausdrucks d kann demzufolge als Grenzwert folgendermaßen definiert werden:

$$[\![d]\!](a)(t) := \lim_{\delta \to +0} \; a_d(t-\delta)$$

Die ausführbare Semantik ist analog als

$$[d](a)(t) := a_d(\; t - \delta \;)$$

für die diskrete Schrittweite δ definiert.

Der Zugriff auf den aktuellen Wert eines Kanals ist vorrangig für zeitdiskrete Kanäle relevant, deren Werte in den Aktionen von Transitionen hybrider Automaten verändert werden (vgl. Abschnitte 3.4.1 und 4.5). Obwohl zeitdiskrete Kanäle – semantisch betrachtet – auch Ströme repräsentieren, sind sie von der Intention her „gewöhnliche Variablen“ im Sinne klassischer Programmiersprachen. Durch die Möglichkeit des Zugriffs auf den aktuellen Wert ist diese Programmiersprachen-Intention auch in TPT abgebildet: Auch ohne Angabe eines Zeitpunktes hat jeder Kanal trotzdem immer einen „aktuellen Wert“.

Funktionen

Die Funktionalität von Testfällen ist in der Regel vergleichsweise einfach, so dass die bisher erläuterten Operatoren zur Modellierung von Gleichungen in vielen Fällen bereits ausreichen. Bei genauerer Betrachtung praktischer Fälle wird aber schnell deutlich, dass zusätzlich auch weitere „klassische“ Funktionen wie $sin(x)$, $ln(x)$, $sqrt(x)$ etc. zur Modellierung wünschenswert sind. Es gibt grundsätzlich mehrere Möglichkeiten, wie derartige Funktionen in die Termsprache von TPT zu integrieren sind.

1. Vordefinierte, feste Bibliothek von Funktionen: Dieser Ansatz ist einfach implementierbar, da jede Funktion fest in der Termsprache verankert wird. Der offensichtliche Nachteil ist jedoch, dass es unmöglich ist, weitere Funktionen nachträglich hinzuzufügen.

2. Spezifische Sprachmittel in TPT zur Modellierung von neuen Funktionen in Analogie zur Definition von Funktionen bzw. Methoden in klassischen Programmiersprachen: Durch diesen Ansatz würde die Termsprache von TPT quasi zu einer Programmiersprache anwachsen und

hat damit den Nachteil, dass das einfache und leicht erlernbare Sprachkonzept von TPT durch die Funktionsmodellierung stark verkompliziert würde.

3. Offenes Konzept zur Integration von *Funktionsbausteinen*: TPT stellt eine Art „Plug-In-Schnittstelle“ für die Verwendung von Funktionen bereit. Spezifische Funktionen, die für die Modellierung von Testfällen verwendet werden sollen, können mit Hilfe von Standard-Programmiersprachen implementiert werden, die die TPT Plug-In-Schnittstelle unterstützen. Diese Funktionen können anschließend bei der Modellierung von Testfällen importiert werden.

TPT verwendet von diesen Möglichkeiten die dritte Variante der Unterstützung von Funktionsbausteinen. Jeder Funktionsbaustein hat in diesem Zusammenhang drei Aufgaben zu erfüllen:

1. Name der Funktion: Jede Funktion legt selbst den Namen fest, unter dem sie verwendet werden kann (z.B. „`noise`“ oder „`sin`“). Wird die Funktion für die Modellierung von Testfällen mit TPT eingebunden, wird sie unter diesem Namen *global*, d.h. für *alle* Ausdrücke, sichtbar. Der Name muss deshalb global eindeutig sein. Funktionen teilen sich mit allen Kanälen und Konstanten den Namensraum (vgl. Abschn. 4.3.2), so dass es beispielsweise keinen Kanal mit dem Namen `noise` geben darf, wenn eine Funktion `noise` verwendet werden soll.

2. Ausführungssemantik: Die Ausführungssemantik ist die „Berechnungsvorschrift“ der Funktion. Sie bestimmt den Funktionswert in Abhängigkeit von den gegebenen Argumenten. Hierbei kann die Funktion auch auf den Kontext der Ausführung von TPT zugreifen, um beispielsweise Werte von Kanälen abzufragen, den aktuellen Zeitpunkt zu bestimmen oder die Schrittweite der Berechnung zu ermitteln.

3. Kontextbedingungen für die Übersetzung von Testfällen: Die Analyse der notwendigen Kontextbedingungen für die Applikation einer Funktion wird von der Funktionskomponente selbst durchgeführt. Die TPT-Kontextanalyse ruft hierfür die spezifische Kontextanalyse einer Funktion für jede Applikationsstelle der Funktion separat auf. Funktionen können (und sollten) hierbei unter anderem die Anzahl und die Typen der übergebenen Argumente überprüfen; sie können aber je nach Bedarf auch weiterführende Analysen durchführen (z.B. prüfen, ob ein Parameter eine Konstante ist).

Ein Funktionsbaustein wird in zwei verschiedenen Rollen benötigt: Erstens für die Übersetzung von Testfällen während der Kontextanalyse und zweitens während der Testdurchführung zur Berechnung des Funktionswertes. Da diese zwei Aufgaben in der Regel auf unterschiedlichen Plattformen ablaufen,

besteht ein Funktionsbaustein in der Regel auch aus mehreren Teilen. Die Details der technischen Integration von Funktionsbausteinen in TPT hängt eng mit der Gesamtarchitektur von TPT zusammen und wird deshalb erst im Kapitel 6 genauer erläutert.

Die Syntax einer Funktionsanwendung innerhalb eines Ausdrucks entspricht der von C, d.h., dem Funktionsnamen folgt in Klammern die kommaseparierte Liste der Parameter $fun(arg_1, arg_2, \dots)$. Als Parameter sind grundsätzlich sowohl beliebige Ausdrücke vom Typ `int`, `float` oder `boolean` als auch konstante Zeichenketten (Strings) zulässig. Die Strings sind unter anderem sinnvoll, um einer Funktion gegebenenfalls auch Dateinamen, Kanalnamen, textuelle Optionen, Log-Texte o.ä. übergeben zu können. Jede Funktion muss in ihrer eigenen, spezifischen Kontextanalyse verifizieren, ob die übergebenen Argumente die richtige Anzahl und den richtigen Typ haben.

Durch den Komponentenansatz ist ein sehr offenes und flexibles Framework gegeben, in das prinzipiell auch sehr komplexe Funktionen eingebettet werden können. Im Folgenden sollen kurz einige mögliche Anwendungsfälle für Funktionen skizziert werden:

```
x(t) = 2.0 * sin(t/2.0 * pi())
y(t) = 10.0 + 0.1 * noise(t)
i(t) = min((int)(t*0.1), 100)
j(t) = loadfile("/usr/tpt/reference.dat", t)
```

Beispielsweise wird bei der Definition von `j` die Funktion `loadfile`() verwendet, die eine Referenzmessung aus einer existierenden Datei lädt und als Grundlage der Definition von `j` verwendet. Die Datei enthält also Referenzdaten, auf die jeweils unter Angabe eines Zeitpunktes zugegriffen werden kann.

4.4.4 Grafische Definition unabhängiger Verläufe

Die Definition von Kanälen mit Hilfe von Ausdrücken, wie sie im letzten Abschnitt beschrieben wurde, ermöglicht es, Kanäle in Abhängigkeit anderer Kanäle zu modellieren. In vielen Fällen sind Signalverläufe eines Kanals jedoch unabhängig von der Belegung anderer Kanäle. Solche Verläufe werden deshalb als *unabhängige Verläufe* bezeichnet. Beispiele sind konstante Verläufe wie $\texttt{x(t)} = \texttt{10}$ oder lineare Verläufe wie $\texttt{y(t)} = \texttt{2} * \texttt{t} - \texttt{3}$.

In solchen einfachen Fällen ist die Definition der Verläufe mit TPT alternativ auch auf grafische Weise möglich und sinnvoll. Vorteil der grafischen Definition von Verläufen ist die bessere Intuition und Anschaulichkeit im Gegensatz zur Repräsentation mit Hilfe von Gleichungen. Der Nachteil der grafischen Definition ist jedoch der höhere Modellierungsaufwand, der in vielen Fällen benötigt wird. Ziel ist es deshalb, die grafische Definition von unabhängigen Verläufen in TPT zu unterstützen, wegen des offensichtlichen

Für und Wider jedoch jederzeit zur gleichungsbasierten Definition wechseln zu können.

Zu diesem Zweck werden auch grafisch modellierte Signalverläufe intern in Form von Ausdrücken repräsentiert. Diese Ausdrücke können dann entweder direkt textuell editiert oder aber mit Hilfe des grafischen Signaleditors bearbeitet werden.

TPT unterstützt zwei Formen von Signalverläufen für die grafische Definition, die für die meisten praktisch relevanten Fälle ausreichend sind: *Streckenzüge* und *kubische Splines*. Beide Formen werden im Folgenden vorgestellt.

Streckenzüge (stückweise lineare, stetige Funktionen): Diese Funktionen werden grafisch wie im ersten Beispiel aus Abbildung 4.5 visualisiert und modelliert. Die Funktion wird durch eine Folge von linearen (Teil-)Funktionen gebildet, die jeweils stetig aneinander grenzen und somit einen Streckenzug bilden. Im grafischen Editor können die Stützstellen dieses Streckenzugs beliebig verändert, neu erzeugt oder gelöscht werden.

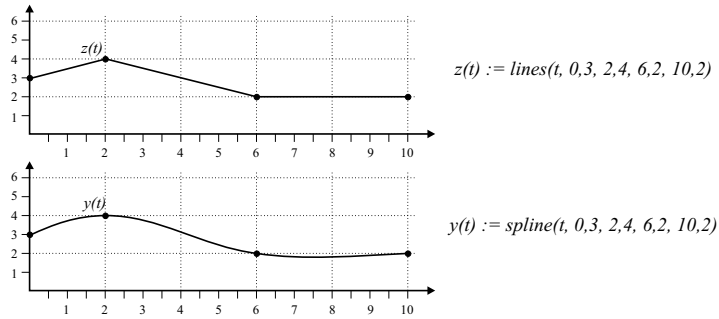

Abbildung 4.5: Beispiele der grafischen Signalmodellierung

Für die interne Repräsentation eines Streckenzugs als Ausdruck wird die Funktion `lines` verwendet. Diese Funktion gehört zur Standard-Funktionsbibliothek von TPT. Sie verlangt $2n+1$ Argumente ($n \geq 1$) vom Typ `float` und hat folgende Semantik:

Wird die Funktion mit den Argumenten $\texttt{lines}(\hat{x}, x_1, y_1, x_2, y_2, \ldots, x_n, y_n)$ aufgerufen, muss die Folge $\{x_i\}_{i=1,\ldots,n}$ streng monoton wachsend sein. Andernfalls ist der Wert von `lines` undefiniert. Ist die Monotonie-Eigenschaft erfüllt, gilt

$$\texttt{lines}(\hat{x}, x_1, y_1, \ldots, x_n, y_n) := \begin{cases} y_1 & \text{falls } \hat{x} < x_1 \\ (\hat{x} - x_i) \cdot \frac{y_{i+1} - y_i}{x_{i+1} - x_i} + y_i & \text{mit } x_i \leq \hat{x} < x_{i+1} \\ y_n & \text{falls } x_n \leq \hat{x} \end{cases}$$

Mit anderen Worten: Die Funktion modelliert den stückweise linearen Verlauf, der durch die definierten Stützstellen (x_i, y_i) geht. Vor der ersten Stützstelle (x_1, y_1) bzw. hinter der letzten Stützstelle (x_n, y_n) verläuft die Funktion konstant mit dem Wert y_0 bzw. y_n. Das erste Argument $\hat{x}$ der Funktion `lines`() ist der x-Wert, zu dem `lines`() den Funktionswert dieses stückweise linearen Verlaufs liefern soll.

In der grafischen Darstellung wird nur der Streckenzug mit den Stützstellen (x_i, y_i) betrachtet; der Ausdruck $\hat{x}$ kann also nur textuell editiert werden. Initial wird er als $\hat{x} =$ `t` festgelegt, so dass die X-Achse des Streckenverlaufs mit der Zeitachse übereinstimmt.

Kubische Splines: Stückweise lineare, stetige Funktion sind in den Stützpunkten nicht differenzierbar. Für viele reale Größen ist die Diffenzierbarkeit des Verlaufs jedoch erforderlich, wenn sie naturgemäß diffenzierbar sind. Für solche Funktionen bietet sich die kubische Splineinterpolation an, die per Definition zweifach diffenzierbar ist. Das Prinzip der Modellierung ist identisch zu Steckenzügen, es wird jedoch die folgende Funktion `spline`() eingesetzt (vgl. zweites Bsp. aus Abb. 4.5):

$$\texttt{spline}(\hat{x}, x_1, y_1, \ldots, x_n, y_n) := \begin{cases} y_1 & \text{falls } \hat{x} < x_1 \\ S(\hat{x}) & \text{mit } x_i \leq \hat{x} < x_{i+1} \\ y_n & \text{falls } x_n \leq \hat{x} \end{cases}$$

wobei $S(\hat{x})$ die Splinefunktion, für die gilt:

- $S(x_i) = y_i$ für $i = 1, \ldots, n$, d.h., (x_i, y_i) sind Stützstellen
- $S(x_i)$ ist in jedem Teilintervall $[x_i, x_{i+1}]$ ein Polynom vom Grad ≤ 3
- $S(x_i)$ ist zweifach stetig diffenzierbar in $\mathbb{R}$
- $S''(x_1) = S''(x_n) = 0$.

Für jede Menge von Stützstellen $\{(x_i, y_i)\}$ gibt es genau eine Funktion $S(\hat{x})$, die die obigen Eigenschaften erfüllt [BS+95].

Mit Hilfe der Streckenzüge und der kubischen Splines können die meisten praktisch relevanten unabhängigen Signalverläufe ohne weiteres approximiert werden. Die Sprachmächtigkeit ist jedoch beschränkt. Für spezifische Anforderungen sollte demnach auf die „normale“ Modellierung von Ausdrücken zurückgegriffen werden.

Verwendungsmöglichkeiten von $\hat{x}$: Durch die freie Wahl des Ausdrucks $\hat{x}$ können Streckenzüge und Splines für beliebige funktionale Zusammenhänge $\mathbb{R} \to \mathbb{R}$ verwendet werden. Gibt es beispielsweise eine charakteristische Kennlinie, die zu einem gegebenen Luftdruck den zugehörigen Ventilstrom bestimmt, der notwendig ist, um den Druck in einem Behälter aufzubauen, dann kann dies mit

$$\texttt{current(t)} = \texttt{spline(pressure(t)}\ , 0, 0,\ 1, 2.54,\ 2, 4.53, \dots)$$

dargestellt werden. Die Kennlinie mit ihren Stützstellen ist dabei grafisch editierbar. Lediglich der „Zugriffspunkt“ $\hat{x}$ muss nachträglich textuell verändert werden.

Analog kann ein periodisches Signal modelliert werden, indem $\hat{x}$ ein periodisches Signal beschreibt.

4.5 Time Partitioning

Die zweite Technik, die neben der direkten Definition zur Modellierung eines Szenarios verwendet werden kann, ist das so genannte *Time Partitioning*. Diese Technik basiert auf den im Abschnitt 3.4 vorgestellten hybriden Systemen, die den Gesamtablauf des Tests in einzelne zeitliche Phasen zerlegen.

4.5.1 TP-Diagramme

Grundlage der Modellierung von Szenarien mit dem Time Partitioning sind erweiterte Zustands-Übergangs-Diagramme, die im Folgenden kurz *TP-Diagramme* genannt werden. Mit Hilfe von TP-Diagrammen wird grafisch festgelegt, welche Zustände bzw. Phasen es gibt und unter welchen Bedingungen von einem Zustand zum nächsten gewechselt werden soll. Im Folgenden wird zunächst die Syntax der TP-Diagramme beschrieben. Ihre Semantik wird in den Abschnitten 4.5.2 und 4.5.3 erläutert.

Die für TP-Diagramme zur Verfügung stehenden Techniken wie Parallelität, Hierarchien, Aktionen an Transitionen basieren auf den Ideen der Statecharts von Harel [Har87]. Semantisch beschreiben TP-Diagramme jedoch stromverarbeitende Komponenten und unterscheiden sich deshalb in der Bedeutung von Statecharts.

Für TP-Diagramme stehen die folgenden syntaktischen Elemente zur Verfügung (vgl. Abb. 4.6), die zunächst informell eingeführt werden:

(a) **Zustände:** Zustände in TP-Diagrammen sind die syntaktische, grafische Repräsentation von Zuständen hybrider Systeme, wie sie im Abschnitt 3.4 eingeführt wurden. Jedem Zustand ist somit eine stromverarbeitende Komponente zugeordnet, die das Verhalten des Szenarios, das mit einem TP-Diagramm modelliert wird, für einen definierten Zeitabschnitt festlegt. Dieser Zeitabschnitt wird dadurch definiert, dass der Zustand zu einem bestimmten Zeitpunkt auf Grund einer schaltenden Transition betreten und zu einem späteren Zeitpunkt durch eine schaltende Transition wieder verlassen wird. Innerhalb des Zeitabschnitts bestimmt der Zustand mit seiner zugeordneten stromverarbeitenden Kom-

ponente das Verhalten des gesamten Szenarios (weitere Details hierzu folgen weiter unten in diesem Abschnitt).

Jeder Zustand wird als Rechteck dargestellt, das als Label eine informelle Kurzbeschreibung des Zustandes enthält (vgl. Abb. 4.6a).

(b) **Junctions:** Junctions sind Verknüpfungs- und Verzweigungspunkte für Transitionen. Wie weiter unten unter dem Punkt „Zustandsübergänge“ auf Seite 82 noch ausführlicher erläutert wird, dienen Junctions nur zur Verbindung von Transitionen, die semantisch immer gemeinsam betrachtet werden. Insofern sind Junctions nur syntaktische Abkürzungen, die es ermöglichen, einen Zustandsübergang Semantik erhaltend in mehrere Teile zu zerlegen.

Daraus resultiert insbesondere, dass es keinen nichtleeren Zeitabschnitt gibt, in dem der aktuelle Zustand des Automaten des TP-Diagramms eine Junction ist. Aus diesem Grund wird Junctions auch kein kontinuierliches Verhalten im Sinne einer stromverarbeitenden Komponente zugeordnet.

Junctions werden grafisch durch gefüllte Kreise symbolisiert (vgl. Abb. 4.6b).

(c) **Initiale Knoten:** Initiale Knoten modellieren den Eintrittspunkt in einen Automaten. Sie werden grafisch wie Junctions dargestellt. Im Unterschied zu Junctions haben initiale Knoten keine eingehenden Transitionen (vgl. Abb. 4.6c).

(d) **Finale Knoten:** Finale Knoten repräsentieren die Termination eines Szenarios. Von finalen Knoten gehen keine weiteren Transitionen aus (vgl. Abb. 4.6d).

(e) **Transitionen:** Transitionen beschreiben die Übergänge zwischen Zuständen, Junctions, initialen und finalen Knoten. Transitionen sind eine syntaktische Repräsentation der Transitionen hybrider Systeme, d.h., jeder Transition in einem TP-Diagramm sind analog zur Modellierung hybrider Systeme im Abschnitt 3.4 eine formale Transitionsbedingung und entsprechende Aktionen zugeordnet, die beschreiben, unter welchen Bedingungen die Transition schalten darf und welche Aktionen zu diesem Übergangszeitpunkt ausgeführt werden sollen.

Transitionen werden grafisch als Pfeile dargestellt und können bei Bedarf – zusätzlich zur formalen Transitionsbedingung – durch eine informelle Umschreibung der Transition in Form eines Labels annotiert werden, das an der Transition steht (vgl. Abb. 4.6e).

(f) **Parallelisierungen:** Analog zu Statecharts können auch in TP-Diagrammen parallele Automaten modelliert werden. Dies wird durch eine

horizontale Strichlinie dargestellt, mit der zwei parallelen Teilautomaten grafisch voneinander getrennt werden (vgl. Abb. 4.6f).

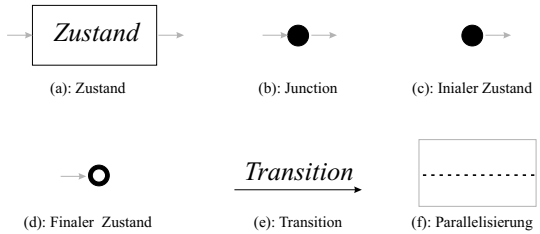

Abbildung 4.6: Diagrammelemente für TP-Diagramme

In Abbildung 4.7 ist ein typisches Beispiel für ein TP-Diagramm dargestellt, das alle verfügbaren grafischen Elemente von TP-Diagrammen enthält. Das Diagramm definiert zwei parallele Abläufe: Im ersten Ablauf werden drei Phasen unterschieden. Der Test beginnt mit dem initialen Knoten, von dem aus sofort in den Zustand „Einschalten“ gewechselt wird. Es schließen sich zwei alternative Transitionen an, an die jeweils unterschiedliche Bedingungen geknüpft sind. Die erste Transition schaltet, sobald die Initialisierung des Systems abgeschlossen ist; die zweite Transition schaltet 10 Sekunden nach Beginn des Einschaltvorgangs. Die Transition, deren Bedingung zuerst erfüllt ist, bestimmt den Folgezustand. Ist schließlich die Phase „Funktionstest“ abgeschlossen, terminiert der Testfall.

Der zweite, parallele Ablauf geht vom initialen Knoten sofort in den Zustand „Warten“ und wechselt in den Zustand „Bestätigen“, sobald die vorherige Transition „wenn Bestätigung erforderlich“ schaltet. In diesem Zustand bleibt der Automat, bis der Testfall terminiert.

Im Folgenden wird nun die Modellierung von TP-Diagrammen ausführlich erläutert.

TP-Diagramme als Graphen

Grundsätzlich stellt jedes TP-Diagramm einen gerichteten Graphen dar. Die Knoten dieses Graphen sind alle Zustände und Junctions sowie alle initialen und finalen Knoten. Die Kanten des Graphen sind die Transitionen; Parallelisierungen werden zunächst noch nicht betrachtet. Wird ein so gebildeter Graph bzw. der entsprechende Automat ausgeführt, durchläuft er den Graphen entlang eines Pfades $v_1 \overset{e_1}{\rightarrow} \cdots \overset{e_{n-1}}{\rightarrow} v_n \ldots$ mit den Knoten v_i und den Kanten e_i. Der Übergang von einem Knoten zum nächsten erfolgt hierbei jeweils zu einem definierten Zeitpunkt $t_i \in \mathbb{R}$, bei dem die Transition e_i schaltet,

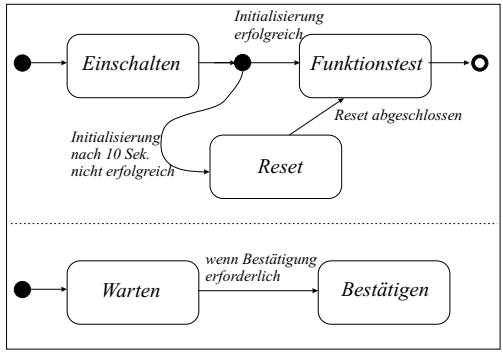

Abbildung 4.7: Beispiel eines TP-Diagramms

so dass sich der Automat während der Ausführung zwischen den Zeitpunkten t_{i-1} und t_i im Knoten v_i „aufhält“, was genau der Idee der hybriden Systeme entspricht.

Der entscheidende Unterschied der verschiedenen Arten von Knoten ist, dass für alle Zustände v_i die Ungleichung $t_{i-1} \neq t_i$ gilt, während für die drei anderen Arten von Knoten $t_{i-1} = t_i$ gilt. D.h., Zustände treten immer in nicht leeren Intervallen $[t_{i-1}, t_i)$ auf (die per Definition in Analogie zu den hybriden Systemen im Abschnitt 3.4 halboffen sind), während alle anderen Knoten nur zu einem Zeitpunkt $t_{i-1} = t_i$ auftreten.

Junctions, initiale und finale Knoten unterscheiden sich nur bzgl. ihres Auftretens im Pfad: Initiale Knoten sind immer die ersten durchlaufenen Knoten eines Pfades (v_1 ist also immer ein initialer Knoten); finale Knoten sind – wenn sie überhaupt erreicht werden – immer die letzten Knoten eines Pfades. Junctions sind alle übrigen Knoten.

In der Abbildung 4.8 ist ein Beispiel für einen Lauf durch ein TP-Diagramm abgebildet, bei dem die Transitionspunkte der einzelnen Transitionen visualisiert sind. Man erkennt in diesem Beispiel insbesondere, dass zum Zeitpunkt $t = t_2$ drei Transitionen in Folge schalten. Diese Betrachtung von Folgen von Transitionen wird weiter unten bei der Definition der Zustandsübergänge aufgegriffen.

Verhalten der Zustände

Da sich der Automat in jedem durchlaufenen Zustand für eine nicht leere Zeitspanne $[t_{i-1}, t_i)$ aufhält, muss der Zustand das kontinuierliche Verhalten in diesem Abschnitt definieren. Dieses Verhalten wird hierzu rekursiv mit Hilfe eines Sub-Szenarios modelliert (vgl. Abschn. 4.3), das dem Zustand zugeord-

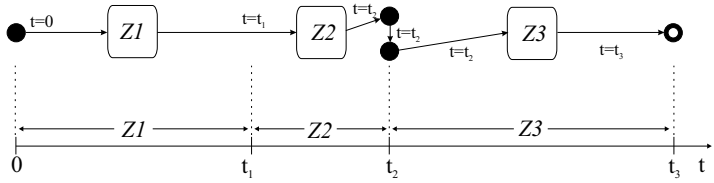

Abbildung 4.8: Exemplarischer Lauf durch ein TP-Diagramm

net ist. Es kann – wie alle Szenarien – mit Hilfe der direkten Definition oder durch ein Sub-TP-Diagramm modelliert werden und beschreibt semantisch eine stromverarbeitende Komponente.

Transitionen

Transitionen dienen dazu, die potenziellen Übergänge zwischen den Knoten eines Diagramms festzulegen. Jeder Transition ist eine formale *Transitionsbedingung* in Form eines Ausdrucks vom Typ `boolean` (vgl. Abschn. 4.4.3) zugeordnet, die in der grafischen Darstellung nicht eingeblendet wird. Die Transitionsbedingung kann zu jedem Zeitpunkt t berechnet werden und definiert, ob die Transitionsbedingung erfüllt ist oder nicht, d.h., ob die Transition potenziell schalten kann.

Für alle Transitionen eines TP-Diagramms gelten folgende syntaktische Randbedinungen:

T1. Ein initialer Knoten ist niemals Ziel einer Transition.

T2. Ein finaler Knoten ist niemals Quelle einer Transition.

T3. Es gibt keine Transitionen, die direkt von einem initialen zu einem finalen Knoten führen.

T4. Eine Junction hat mindestens eine eingehende und mindestens eine ausgehende Transition.

T5. Ein Zustand hat mindestens eine eingehende und höchstens eine ausgehende Transition.[2]

T6. Auf jedem zyklischen Pfad von Transitionen $v_1 \rightarrow \cdots \rightarrow v_n \rightarrow v_1$ muss mindestens ein Zustand liegen. Dies ist erforderlich, um zu verhindern, dass es unendliche Pfade von Junctions gibt, die keine Zeit verbrauchen.

[2]Die Tatsache, dass ein Zustand höchstens eine ausgehende Transition haben darf, bedeutet nicht, dass es keine Verzweigungen gibt (vgl. Abb. 4.7). Die Motivation der Einschränkung kann erst im Kapitel 5 im Zusammenhang mit der Definition der Pfadvariation gegeben werden.

Diese Randbedingungen sind statisch zum Zeitpunkt der Kontextanalyse prüfbar.

Anmerkung: Die Eigenschaft T3 verhindert, dass ein TP-Diagramm im Moment des Betretens des Automaten bereits wieder terminiert. D.h., es wird syntaktisch ausgeschlossen, dass eine solche spontane Termination möglich ist. Dieser Punkt wird im Zusammenhang mit der ausführbaren Semantik eines TP-Diagramms noch einmal aufgegriffen.

Zustandsübergänge

Ein Wechsel bzgl. des kontinuierlichen Verhaltens erfolgt durch das Schalten von einem Zustand A in einen Zustand B mit Hilfe von Transitionen. Der Weg von A nach B besteht in der Regel nicht nur aus *einer* Transition, sondern aus einer Folge $A \stackrel{p_1}{\rightarrow} \bullet \stackrel{p_2}{\rightarrow} \cdots \stackrel{p_{n-1}}{\rightarrow} \bullet \stackrel{p_n}{\rightarrow} B$ von Transitionen von A über eine Menge von Junctions bis hin zu B. Da Junctions jeweils nur zu einem Zeitpunkt auftreten, grenzt das Verhalten von A genau an das Verhalten von B (vgl. Abb. 4.8).

Würde man einen Zustandsübergang $A \stackrel{p_1}{\rightarrow} \cdots \stackrel{p_n}{\rightarrow} B$ tatsächlich als Folge von Einzeltransitionen interpretieren, so müsste als Voraussetzung immer $p_i \implies p_{i+1}$ sichergestellt sein, da beim Schalten von p_1 unmittelbar zum selben Zeitpunkt immer auch p_2 sowie alle weiteren Transitionen schalten müssen, da sich der Ablauf an den zwischen A und B liegenden Junctions nicht aufhalten darf. Als Konsequenz dieser Voraussetzung wären alle Bedingungen p_i mit $i > 1$ überflüssig, da sie trivialerweise erfüllt sind, wenn p_1 erfüllt ist.

Deshalb wird ein solcher Zustandsübergang in TPT immer als Ganzes interpretiert, d.h., ein Zustandsübergang ist genau dann erfüllt, wenn *alle* vorhandenen Bedingungen der Transitionen *gleichzeitig* erfüllt sind. Die gemeinsame Übergangsbedingung ist also $p_1 \wedge p_2 \wedge \cdots \wedge p_{n-1} \wedge p_n$.

TPT lässt keine Nichtdeterminismen zu, d.h., befindet sich das Diagramm zu einem Zeitpunkt t im Zustand A, dann darf es höchstens einen erfüllten Zustandsübergang geben. Diese semantische Restriktion kann leider nicht statisch, sondern nur zur Laufzeit überprüft werden. Wird also zur Laufzeit erkannt, dass es gleichzeitig zwei erfüllte Zustandsübergänge vom aktuellen Zustand aus gibt, dann terminiert die Ausführung mit einem Laufzeitfehler (vgl. Abschn. 4.5.3).

In Abb. 4.9 sind drei synaktische Beispiele für nicht triviale Zustandsübergänge aufgeführt. Im Beispiel (a) wird eine Verzweigung modelliert. Vom Zustand $Z1$ aus gibt es zwei mögliche Zustandsübergänge $Z1 \stackrel{t1}{\rightarrow} \bullet \stackrel{t2}{\rightarrow} Z2$ sowie $Z1 \stackrel{t1}{\rightarrow} \bullet \stackrel{t3}{\rightarrow} Z3$. Je nachdem, welche Transitionen erfüllt sind ($t1 \wedge t2$ oder $t1 \wedge t3$), ist der entsprechende Übergang erfüllt. Im Beispiel (b) wird die Transition $t3$ „wiederverwendet“. Diese Form ist nützlich, wenn für den Wechsel nach $Z3$ generell eine bestimmte Voraussetzung zu erfüllen ist, die in der Tran-

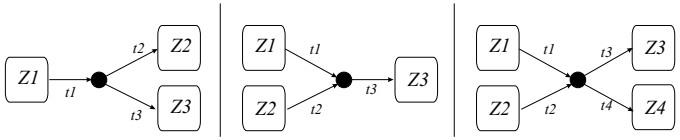

Abbildung 4.9: Zustandsübergänge

sitionsbedingung von $t3$ beschrieben wird. Beispiel (c) ist die Kombination von (a) und (b). Es ergeben sich vier unterschiedliche Zustandsübergänge. Sowohl für $Z1$ als auch für $Z2$ gibt es jeweils zwei mögliche Zustandsübergänge.

Aktionen an Transitionen

Transitionen können – zusätzlich zu den Transitionsbedingungen – bei Bedarf so genannte *Aktionen* zugeordnet werden. Schaltet eine Transition während eines Zustandsübergangs, wird die zugeordnete Aktion ausgeführt.

Jede Aktion besteht aus einer Folge von Zuweisungen der Form $c = E$, wobei c ein Kanal und E ein Ausdruck vom Typ M_c ist (vgl. Abschn. 4.4.3). Eine Aktion wird nur zu einem einzigen Zeitpunkt – dem Schaltzeitpunkt τ der Transition – ausgeführt. Dabei wird den Kanälen c zum Zeitpunkt τ der berechnete Wert des Ausdrucks E zugeordnet; anschließend verlaufen die Kanäle für die Dauer des nächsten Zustands konstant (vgl. auch Abschn. 3.4.1).

Von der natürlichen Intuition her ändert ein Kanal c, der in Aktionen definiert wird, nur dann seinen Wert, wenn an einer Transition eine Definition für c existiert. Ansonsten behält der Kanal seinen aktuellen Wert konstant bei. Solche Kanäle werden als *nichtflüchtig* (non-volatile) bezeichnet. Im Gegensatz dazu heißen alle bisher betrachteten Kanäle *flüchtig* (volatile), da es für diese zu jedem Zeitpunkt t immer eine explizite Definition geben muss (wie beispielweise bei der direkten Definition).

Um die unterschiedliche Bedeutung dieser zwei Arten von Kanälen explizit zu machen, wird für jeden Kanal bereits bei der Deklaration festgelegt, ob er flüchtig oder nichtflüchtig ist. Im Deklarations-Editor steht hierfür eine entsprechende Option zur Verfügung.

In Aktionen dürfen sinnvollerweise nur nichtflüchtige Kanäle definiert werden. Ein Zustand kann jedoch beide Arten von Kanälen definieren, da sein Verhalten wiederum mit einem TP-Diagramm modelliert sein könnte, das seinerseits Aktionen enthält.

Parallele Automaten

Mit Hilfe einer horizontalen Strichlinie ist es möglich, mehrere parallele Automaten in einem TP-Diagramm zu modellieren. Die dadurch abgegrenzten Bereiche des Diagramms werden *Regions* genannt. Jedes Element eines Diagramms (mit Ausnahme der Parallelisierungen selbst) muss eindeutig in einer Region liegen. Das bedeutet insbesondere für Transitionen, dass sie nicht Elemente verschiedener Regions verbinden dürfen.

Jede Region definiert damit einen eigenständigen Automaten, dessen intuitive Semantik eine stromverarbeitende Komponente ist. Mehrere parallele Automaten werden semantisch entsprechend durch die Parallelisierung dieser Komponenten abgebildet und definieren damit selbst eine Komponente (vgl. Abschn. 4.5.2).

Initialisierung

Initiale Knoten markieren den Einstiegspunkt in die Testdurchführung. In jeder Region muss es genau einen initalen Knoten geben. Jeder Zustandsübergang der Form $\bullet \rightarrow \bullet \rightarrow \cdots \rightarrow A$, der vom initialen Knoten über eine Folge von Junctions zu einem Zustand A führt, wird als *initialer Zustandsübergang* bezeichnet.

Da sich der Automat in einem initialen Knoten nicht aufhalten darf, muss zwangsläufig bereits zum Zeitpunkt $t = 0$ für beliebige Eingangsbelegungen der Kanäle ein initialer Zustandsübergang erfüllt sein, der den Folgezustand bestimmt. Um die oben erläuterte Determinismusforderung schaltender Transitionen zu erfüllen, bedeutet dies, dass bei $t = 0$ sogar immer *genau ein* initialer Zustandsübergang erfüllt sein muss. Diese Einschränkung wird zur Laufzeit überprüft (vgl. Abschn. 4.5.3).

Termination

Um die Termination eines Automaten zu modellieren, können finale Knoten verwendet werden. Sobald ein Zustandsübergang in einen finalen Knoten wechselt, terminiert der Automat. Insofern hat ein finaler Knoten kein kontinuierliches Verhalten.

Bezüglich der Termination ist insbesondere die Frage interessant, wie sich ein Automat verhält, wenn sein aktueller Zustand durch Termination des enthaltenen Subautomaten terminiert. Dieselbe Fragestellung wurde im Zusammenhang mit Komponenten ebenfalls bereits behandelt. In Analogie zu der im Abschnitt 3.4.2 gegebenen Definition soll ein terminierender Zustand keine spezifische Semantik erhalten. Es ist bei der Modellierung lediglich sicherzustellen, dass spätenstens zum Zeitpunkt der Termination eines Zustandes (genau) ein Zustandsübergang erfüllt ist, der den Folgezustand eindeutig definiert. Um dieses „Schalten bei Termination“ nicht von den semantischen,

internen Randbedingungen abhängig zu machen, die zu der Termination eines Zustandes geführt haben, wird stattdessen für Ausdrücke der Transitionsbedingungen ein spezielles Schlüsselwort „`exited`“ eingeführt, das einen boolschen Wert repräsentiert und folgendermaßen definiert ist: Für einen Zustandsübergang $A \xrightarrow{p_1} \bullet \xrightarrow{p_2} \cdots \xrightarrow{p_{n-1}} \bullet \xrightarrow{p_n} B$ ist der Ausdruck `exited` zum Zeitpunkt t für alle Transitionsbedingungen p_i genau dann erfüllt, wenn der Zustand A bei t terminiert.

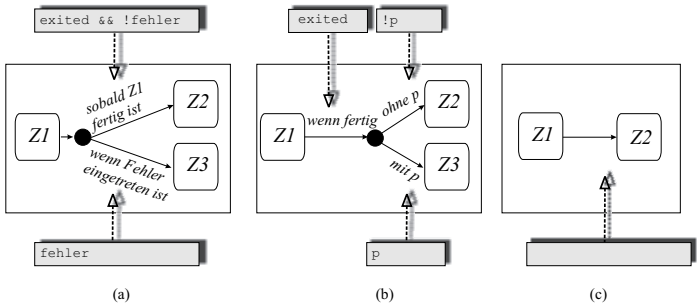

Abbildung 4.10: Zustandsübergänge

In Abbildung 4.10 sind Beispiele für die Verwendung der Termination dargestellt. Im Beispiel (a) gibt es zwei Zustandsübergänge: Der Übergang nach $Z1 \rightarrow \bullet \rightarrow Z3$ hat die formale Bedingung „`fehler`“, die an der zweiten Transition annotiert ist; die erste Transition hat keine Transitionsbedingung. „`fehler`“ ist hierbei ein boolscher Kanal. Der andere Übergang $Z1 \rightarrow \bullet \rightarrow Z2$ ist hingegen nur erfüllt, wenn der Zustand $Z1$ terminiert ist und zusätzlich `!fehler` gilt. Der zweite Teil der Bedingung ist notwendig, da ansonsten ein Nichtdeterminismus zum Laufzeitfehler führen würde, falls zu einem Zeitpunkt t $Z1$ terminiert und gleichzeitig `fehler` gilt.

Im Beispiel (b) gibt es ebenfalls zwei Zustandsübergänge, die beide nur schalten, wenn der Zustand $Z1$ terminiert. Die Übergänge unterscheiden sich in der zweiten Bedingung `p` bzw. `!p`, so dass sichergestellt ist, dass bei Termination von $Z1$ immer einer der beiden Übergange erfüllt ist.

Beispiel (c) beschreibt eine „Standardtransition“, die direkt von $Z1$ nach $Z2$ geht und deren Transitionsbedingung leer ist. Dieser Sonderfall ist eine syntaktische Abkürzung und ist gleichbedeutend mit der Transitionsbedingung `exited`. Mit anderen Worten: Wird an einer solchen Transition keine Bedingung annotiert, schaltet die Transition genau dann, wenn der Zustand $Z1$ terminiert. Diese Semantik ist sehr natürlich und vereinfacht die Modellierung, da diese Art von Übergängen häufig verwendet wird.

4.5.2 Intuitive Semantik

Die intuitive Semantik von TP-Diagrammen basiert auf den stromverarbeitenden Komponenten – insbesondere auf den hybriden Systemen aus Abschnitt 3.4. Eine vollständige formale Definition der intuitiven Semantik würde den Rahmen dieser Arbeit sprengen, so dass hier nur eine grobe Verknüpfung zur Semantik der stromverarbeitenden Komponenten hergestellt wird.

Für ein TP-Diagramm eines Szenarios s ist die intuitive Semantik eine Komponente $[\![s]\!] : \vec{I} \leftrightarrow \overrightarrow{O \cup L}$. Das TP-Diagramm wird durch die Parallelisierungen in eine Folge von Regions $R_1, R_2, \ldots, R_n$ zerlegt. Jede Region R_i bestimmt das Verhalten für eine Teilmenge der Ausgänge und lokalen Kanäle, die als $O_i \subseteq O \cup L$ bezeichnet werden soll. R_i definiert eine Komponente $[\![R_i]\!] : \vec{I_i} \leftrightarrow \vec{O_i}$, wobei $I_i := (I \cup O \cup L) \setminus O_i$. Dann gilt

$$[\![s]\!] := [\![R_1]\!] \parallel [\![R_2]\!] \parallel \cdots \parallel [\![R_n]\!]$$

Die Semantik einer Region lässt sich mit Hilfe eines hybriden Systems präzisieren: Jede Region R_i des TP-Diagramms beschreibt ein hybrides System $H_{R_i} = (V, E, src, dest, init, I_i, O_i, behv, cond)$ mit folgenden Bestandteilen:

V: Menge aller Zustände, initialen und finalen Knoten der Region (ohne Junctions)

E: Menge aller Zustandsübergänge (nicht Transitionen!) der Region

src: Abbildung, die jedem Zustandsübergang den Zustand bzw. den initialen Knoten zuordnet, bei dem der Zustandsübergang beginnt

dest: Analog, Abbildung, die jedem Zustandsübergang den Zustand bzw. einen finalen Knoten zuordnet, bei dem der Übergang endet

init: der eindeutige initiale Knoten der Region $init \in V$

I_i: Menge der Eingangskanäle wie oben definiert

O_i: Menge der Ausgangskanäle, die durch R_i bestimmt werden

behv: Jedem Zustand der Region wird mit *behv* die Semantik $[\![s]\!] : \vec{I_i} \leftrightarrow \vec{O_i}$ des Szenarios s zugeordnet, das zum Zustand gehört. Dem initialen und den finalen Knoten ordnet *behv* beliebige Komponenten zu (ihre Semantik hat keine Auswirkung).

cond: Für jeden Zustandsübergang $A \xrightarrow{p_1} \bullet \xrightarrow{p_2} \cdots \xrightarrow{p_{n-1}} \bullet \xrightarrow{p_n} B$ mit den Transitionsbedingungen p_i liefert *cond* das temporale Prädikat $[\![p_1 \wedge p_2 \wedge \cdots \wedge p_{n-1} \wedge p_n]\!]$. Sonderfall: Ist für einen Zustandsübergang $A \xrightarrow{p} B$ die Transitionsbedinung leer, dann ist die Semantik als $[\![\texttt{exited}]\!]$ definiert.

Die Semantik einer Region R_i ist dann definiert als die Semantik dieses hybriden Systems: $[\![R_i]\!] := \mathsf{hybrid}\, H_{R_i}$.

Nach dieser Definition ist ein TP-Diagramm die syntaktische Form einer Parallelisierung mehrerer hybrider Systeme. Auf die formale Betrachtung der intuitiven Semantik von Transitionsaktionen wird in dieser Arbeit verzichtet.

Die semantische Übersetzung des TP-Diagramms in hybride Systeme ist recht natürlich. Einzige Ausnahme: Junctions werden nur als syntaktische Abkürzungen für TP-Diagramme eingeführt. In der semantischen Beschreibung der hybriden Systeme treten keine Junctions auf. Sie werden nach obiger Definition aufgelöst, indem als Menge E des hybriden Systems anstelle von Transitionen des TP-Diagramms Zustandsübergänge betrachtet werden.

4.5.3 Ausführbare Semantik

Die ausführbare Semantik eines TP-Diagramms beschreibt eine diskrete Komponente $[s] : \vec{I} \leftrightarrow \overrightarrow{O \cup L}$ und ist – analog zur intuitiven Semantik – definiert als Parallelisierung

$$[s] := [R_1] \parallel [R_2] \parallel \cdots \parallel [R_n]$$

Wie auch bei der ausführbaren Semantik der direkten Definition wird gefordert, dass die Regions „von oben nach unten“, d.h. ihrer Reihenfolge nach, berechnet werden. Hierzu ist es erforderlich, dass für jede Region R_i die Menge aller „noch nicht für t definierten“ Kanäle $O_i \cup O_{i+1} \cup \cdots \cup O_n$ bzgl. $[R_i]$ verzögert wirksam sein muss. Das heißt, die Region R_i darf nicht auf Werte von Kanälen zum Zeitpunkt t zugreifen, die in weiter unten stehenden Regions definiert sind. Hintergrund dieser einschränkenden Forderung ist, wie auch bei der ausführbaren Semantik der direkten Definition, dass die eindeutige Berechnungsreihenfolge in der „natürlichen Reihenfolge“ eine unmissverständliche und klar nachvollziehbare ausführbare Semantik gewährleistet.

Die Semantik $[R_i]$ einer einzelnen Region R_i ist eine diskrete Komponente, die zu einem Abtastzeitpunkt $t = i \cdot \delta$ durch folgenden Algorithmus eindeutig bestimmt wird:

A1. Wird der Automat initial (d.h. bei $t = 0$) betreten, dann ist zunächst noch kein aktueller Zustand vorhanden. Es werden alle initialen Zustandsübergänge ermittelt und sofort mit Punkt A8 fortgefahren.

A2. Wird der Automat bei $t > 0$ betreten, dann werden vom aktuellen Zustand, der beim letzten Berechnungsschritt des Automaten durchlaufen wurde, alle existierenden Zustandsübergänge ermittelt. Für all diese Übergänge wird geprüft, ob sie erfüllt sind.

A3. Sind mehrere Übergänge erfüllt, liegt ein Nichtdeterminismus vor und die Ausführung bricht mit einem Laufzeitfehler ab.

A4. Ist keiner der Übergänge erfüllt, wird sofort mit Punkt A7 fortgefahren.

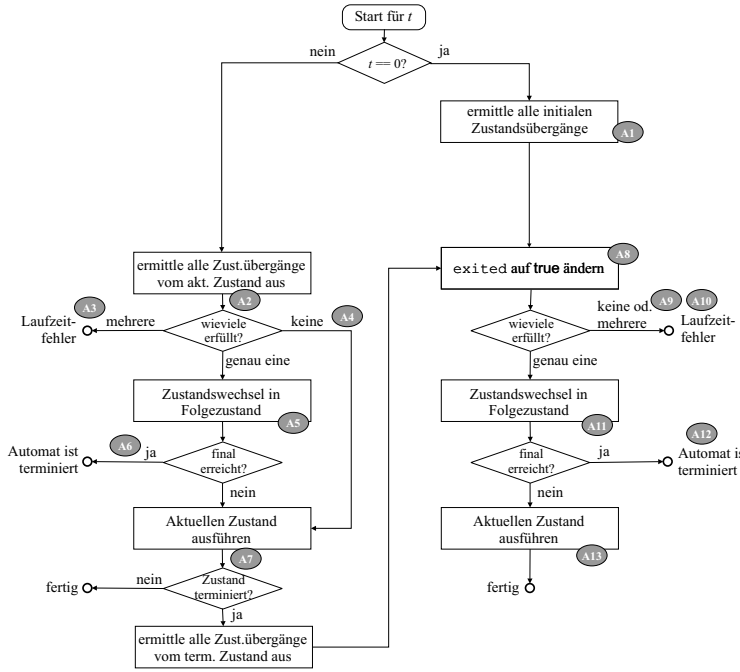

Abbildung 4.11: Ablauf des Automaten-Algorithmus

A5. Ist genau ein Übergang erfüllt, werden die Aktionen aller Transitionen des Übergangs ausgeführt, und der neue aktuelle Zustand ist der Zielzustand des Übergangs.

A6. Ist der neue aktuelle Zustand ein finaler Zustand, ist der Automat damit terminiert. Der Algorithmus bricht mit gesetzten `exited`-Flag entsprechend ab.

A7. Der aktuelle Zustand wird nun ausgeführt und berechnet dabei entsprechend das kontinuierliche Verhalten zum aktuellen Zeitpunkt t. Ist das `exited`-Flag dieses Zustandes nicht gesetzt, d.h., signalisiert der Zustand keine Termination, ist der Algorithmus damit beendet.

Ist das `exited`-Flag des Zustandes hingegen gesetzt, werden alle Zustandsübergänge ermittelt, die vom aktuellen, terminierten Zustand ausgehen und mit Punkt A8 fortgefahren.

A8. Der Wert des Ausdrucks `exited` wird nun von `false` auf `true` geändert, was ggf. dazu führt, dass sich die Wahrheitswerte von Transitionen ändern, wenn sie `exited` in ihrer Bedingung verwenden. Für alle ermittelten Zustandsübergänge wird nun mit `exited` = `true` geprüft, ob sie erfüllt sind.

A9. Sind mehrere Übergänge erfüllt, liegt ein Nichtdeterminismus vor und die Ausführung bricht mit einem Laufzeitfehler ab.

A10. Ist keiner der Übergänge erfüllt, wird gibt es trotz des terminierten Zustandes keinen gültigen Folgezustand: Die Ausführung bricht mit einem Laufzeitfehler ab.

A11. Ist genau ein Übergang erfüllt, werden die Aktionen aller Transitionen des Übergangs ausgeführt, und der neue aktuelle Zustand ist der Zielzustand des Übergangs.

A12. Ist dieser neue aktuelle Zustand ein finaler Zustand, ist der Automat damit terminiert. Der Algorithmus bricht mit gesetzten `exited`-Flag entsprechend ab.

A13. Der aktuelle Zustand wird nun ausgeführt und berechnet dabei entsprechend das kontinuierliche Verhalten zum aktuellen Zeitpunkt t. (Das `exited`-Flag dieses Zustandes kann nicht gesetzt sein, da der Zustand bei t gerade erst betreten wurde.)

Der obige Algorithmus besteht aus zwei Phasen, die je in der linken und rechten Spalte in Abb. 4.11 dargestellt sind. In der ersten Phase (linke Seite) wird als Erstes – ausgehend vom aktuellen Zustand – geprüft, ob es eine schaltende Transition gibt und ggf. der aktuelle Zustand verändert. Erst dann wird die Belegung an den Ausgängen O_i für den aktuellen Zeitpunkt t durch Berechnung des aktuellen Zustands ermittelt.

Die zweite Phase (rechte Seite) arbeitet nahezu analog zur ersten. Der Unterschied besteht darin, dass die Phase 2 nur dann abgearbeitet wird, wenn

kein definierter „aktuellen Zustand“ existiert. Dieser Fall tritt sowohl initial (d.h. beim lokalen Zeitpunkt $t = 0$) als auch immer dann ein, wenn der aktuelle Zustand in Phase 1 berechnet wurde, dieser aber terminiert ist und somit auch die Belegung der Ausgänge O_i zum Zeitpunkt t nicht bestimmt. Zwangsläufig muss in beiden Fällen ein definierter Folgezustand für den Automaten bestimmt werden, der zum Zeitpunkt t das Verhalten festlegt. Um diesen Folgezustand zu ermitteln, werden alle Zustandsübergänge, die vom initialen Knoten bzw. vom gerade terminierten Zustand wegführen, mit gesetztem `exited`-Flag berechnet. Durch dieses Flag kann eine Transition gezielt die Terminationseigenschaft in ihre Übergangsbedingung einbeziehen. Von allen Zustandsübergängen muss genau einer erfüllt sein, der damit eindeutig den Folgezustand bestimmt. Die Belegung an den Ausgängen O_i für den aktuellen Zeitpunkt t wird schließlich durch Berechnung des Folgezustands ermittelt.

Anmerkung: Durch die Eigenschaft T3 auf Seite 81 wurde bereits syntaktisch ausgeschlossen, dass es Transitionen von einem intialen zu einem finalen Knoten gibt. Dadurch stellt der obige Algorithmus sicher, dass die ausführbare Semantik eines TP-Diagramms nicht spontan bei $t = 0$ terminiert.

Diese Definition der ausführbaren Semantik passt exakt zur Idee der hybriden Automaten, da in jedem Fall sichergestellt ist, dass zum Zeitpunkt t eines Zustandswechsels immer der Folgezustand das Verhalten der Ausgänge zum Zeitpunkt t festlegt, so dass die Intervalle, in denen ein Zustand gültig ist, halboffen sind. Da Transitionsbedingungen keine Seiteneffekte haben, ist die hierfür ggf. notwendige „doppelte“ Berechnung der Bedingungen keineswegs problematisch.

4.6 Zusammenfassung

TPT unterstützt zwei unterschiedliche Techniken, mit denen sich kontinuierliches Verhalten beschreiben lässt – die direkte Definition und das Time Partitioning. Diese Techniken ergänzen einander sehr gut und bilden gemeinsam eine kompakte und dennoch ausdrucksstarke Sprache zur Modellierung von kontinuierlichem Verhalten. Die direkte Definition unterstützt dabei die Verhaltensbeschreibung in strukturell einfachen Fällen, während das Time Partitioning durch die Möglichkeiten der hierarchischen Sequenzialisierung und Parallelisierung die strukturellen Zusammenhänge auf höherer Ebene modellieren hilft.

Durch die Kombination von grafischen und textuellen Beschreibungsanteilen ist die Verhaltensbeschreibung auch für solche Domainexperten, Kunden, Qualitätsmanager, Entwickler und Tester intuitiv verständlich, die mit der Modellierungssprache von TPT im Einzelnen nicht vertraut sind. Umfang, Schwerpunkt und Tiefgründigkeit eines Tests können so von allen Beteiligten begutachtet werden und führen zu einem transparenteren Testprozess.

Ein wesentlicher Schwerpunkt beim Design der Modellierungsprache von TPT lag in der einfachen Darstellung von kontinuierlichem Verhalten. Obwohl die ausführbare Semantik letzten Ende eine diskrete, zyklische Berechnung durchführt, ist diese Betrachtungsweise für die Modellierung weitestgehend irrelevant. Die aus der Zeitdiskretisierung resultierenden Fehler sind – wie bereits im Abschnitt 3.5 beschrieben – mit klassischen Wertediskretisierungsfehlern vergleichbar und müssen selbstverständlich bei der Modellierung von Testfällen einkalkuliert werden. Eine Modellierung, die sich die Schrittsemantik explizit zu Nutze macht, ist hingegen nicht notwendig bzw. entspricht auch nicht der zu Grunde liegenden intuitiven, kontinuierlichen Semantik stromverarbeitender Funktionen.

Kapitel 5

Systematische Auswahl

Der Begriff des Tests ist in der Praxis vielfältig belegt. Insofern werden unter der „Durchführung eines Tests“ je nach Kontext sehr unterschiedliche Aktivitäten verstanden, was insbesondere den Vergleich von Vorgehensmodellen zum Test deutlich erschwert. Aus diesem Grund wurde bereits im Abschnitt 1.4 ein abstraktes und universelles Modell des Tests vorgestellt, das unabhängig von der konkreten Testpraxis anwendbar ist.

Anhand dieses Testmodells wurde erläutert, dass die Qualität des Tests eines Systems vorrangig durch zwei Faktoren bestimmt wird: durch die relative Anzahl der aufgedeckten Fehler und durch die Anzahl der benötigten Testfälle. Beide Faktoren – und damit schließlich auch die Testqualität selbst – hängen einzig und allein von der wohlüberlegten Auswahl und Definition der Menge der durchzuführenden Testfälle ab.

Bei der bisherigen Betrachtung von Testfällen wurde im Kapitel 4 zunächst nur festgelegt, wie ein *einzelner* Testfall mit der Modellierungssprache beschrieben werden kann. Für eine optimale Qualität des Tests ist die isolierte Betrachtung eines einzelnen Testfalls jedoch nicht ausreichend. Vielmehr ist die *gemeinsame* Betrachtung einer Menge von Testfällen notwendig, die insgesamt eine hohe Fehleraufdeckungsrate bei angemessenem Testaufwand gewährleisten.

Die Tatsache, dass nur die gemeinsame Betrachtung einer Menge von Testfällen für die Qualität entscheidend ist, erkennt man auch daran, dass die meisten existierenden Testmethoden schwerpunktmäßig beschreiben, wie eine Menge von relevanten Testfällen gezielt ausgewählt werden kann. Die Basis dieser Auswahl ist in Abhängigkeit der jeweiligen Testmethode unterschiedlich. Strukturtestmethoden fokussieren auf die Analyse der inneren Struktur des Systems, das getestet werden soll und werden deshalb oft auch als White-Box-Tests bezeichnet. Funktionstestmethoden beschreiben hingegen Verfah-

ren zur Auswahl von Testfällen anhand der spezifizierten Anforderungen an das zu testende System und werden auch Black-Box-Tests genannt.

Das Time Partition Testing zählt zu den Funktionstestmethoden, d.h., die Auswahl und Definition der Testfälle basiert hauptsächlich auf der Analyse der Anforderungen, die an das System gestellt werden. Zu jeder Anforderung sind in der Regel ein oder mehrere Testfälle erforderlich, die die Anforderung prüfen. Umgekehrt kann zu jedem Testfall die Liste der Requirements angegeben werden, die durch den Testfall geprüft werden, so dass sich eine Relation wie in Tabelle 5.1 dargestellt ergibt.

	Req. 1	Req. 2	Req. 3	Req. 4	Req. 5	Req. 6	...
Testcase 1		•		•	•		
Testcase 2		•					
Testcase 3			•				
Testcase 4					•	•	
...	•	•					

Tabelle 5.1: Relation zwischen Anforderungen und Testfällen

Eine solche formale Gegenüberstellung von Requirements und Testfällen setzt voraus, dass die Auswahl der relevanten Testfälle bereits erfolgt ist. Ziel dieser Auswahl ist es, eine Menge von Testfällen zu finden, die einerseits möglichst wenig Redundanzen enthält und andererseits möglichst viele Requirements unter Berücksichtigung unterschiedlichster Aspekte und Sonderfälle des Systems abdeckt. Die formale Gegenüberstellung von Requirements und Testfällen kann deshalb als Monitoring-Instrument verwendet werden, um beispielsweise verifizieren zu können, ob alle Requirements durch mindestens einen Testfall geprüft werden.

Jede Testmethode muss konstruktiv beschreiben, wie die Auswahl der Testfälle erfolgt. Auch für das Time Partition Testing ist damit die zentrale Frage, wie auf eine systematische und strukturierte Art und Weise eine Menge von Testfällen gefunden werden kann, die für ein zu testendes System eine hohe Testqualität garantieren.

Im weiteren Verlauf dieses Kapitels wird beschrieben, wie die systematische Testfallermittlung mit TPT erfolgt. Es wird dabei insbesondere gezeigt, dass die gewählte Lösung neben einer Verbesserung der Testqualität auch eine starke Aufwandsoptimierung der Testmodellierung – vor allem bei einer großen Anzahl von Testfällen – nach sich zieht, was die praktische Akzeptanz des Testansatzes deutlich verbessert.

5.1 Variation des Verhaltens

Die Grundidee der systematischen Testfallermittlung mit TPT basiert auf einer sehr einfachen Beobachtung aus der Testpraxis: Der grundsätzliche Ablauf vieler Testfälle ähnelt sich, während sich die Testfälle nur in den Details der einzelnen Testschritte unterscheiden.

Diese Beobachtung soll zunächst praktisch illustriert werden, indem das bereits im Abschnitt 2.2 (Seite 21 ff.) verwendete Beispiel einer fiktiven Motorsteuerung ausführlicher diskutiert wird. In diesem Beispiel lässt sich ein Testfall als Folge der Testphasen *Motor starten*, *Gas geben* und *Drehzahlfehler simulieren* modellieren. Hierbei soll die zweite Phase dann beginnen, wenn der Motor bereits gestartet und damit die erste Phase abgeschlossen ist. Die dritte Phase beginnt im Anschluss an Phase 2, sobald die gewünschte Zieldrehzahl erreicht ist, bei der der Drehzahlfehler simuliert werden soll. Nach weiteren 10 Sekunden ist der Testfall abgeschlossen.

Der entstehende Ablauf stellt die Grundlage eines TP-Diagramms mit drei Zuständen und den entsprechenden Transitionen dar (vgl. Abbildung 5.1, Seite 97). Er definiert zunächst aber nur den grundsätzliche Ablauf eines Testfalls. Noch nicht definiert ist das genaue Verhalten während der einzelnen Phasen bzw. die Bedingungen und Aktionen der Transitionen.

Bei der Definition dieser Detailaspekte stellt man fest, dass es einige Zustände und Transitionen gibt, für die nicht nur *eine* mögliche, testrelevante Definition, sondern *mehrere* alternative *Definitionsvarianten* existieren. Für die Phase *Motor starten* liegt es beispielsweise nahe, sowohl einen Kaltstart des Motors als auch alternativ einen Warmstart zu untersuchen, da sich Motor und Motorsteuerung in Abhängigkeit von der Motortemperatur unterschiedlich verhalten. Der Aspekt, wie der Motor beim Test im Einzelnen gestartet wird, ist also aus Sicht des Tests relevant. Die zentrale Frage, die dieser Bildung von Varianten zu Grunde liegt, ist demnach, welche unterschiedlichen Fälle/Varianten für das betrachtete, zu testende Systemverhalten ggf. relevant sein *könnten*. Diese Relevanz wird danach entschieden, ob eine Variante A andere potenzielle Bugs aufdecken kann als eine Variante B. Diese Entscheidung ist jedoch meist eine Schätzung: Im obigen Bespiel liegt nur die Vermutung nahe, dass die Motortemperatur das Schwingungsverhalten beeinflusst und damit die Aufdeckung potenzieller Bugs begünstigt.

In entsprechender Art und Weise können auch alle anderen Zustände und Transitionen des TP-Diagramms untersucht werden. Für den Zustand *Dehzahlfehler simulieren* gibt es trivialerweise unterschiedliche Varianten, da die Auswirkung von Sensorfehlern der Hauptaspekt des Tests ist. Es wird vereinfachend angenommen, dass es folgende relevante Varianten gibt: (a) kurzzeitige niederfrequente Schwingungen, (b) dauerhafte niederfrequente Schwingungen, (c) kurzzeitige hochfrequente Schwingungen, (d) dauerhafte hochfrequente Schwingungen und auch (e) den fehlerfreien Sensor. Es gibt also fünf

unterschiedliche Varianten zu diesem Aspekt.

Ein weiteres Element mit alternativen Definitionen ist die Transition *Sobald Zieldrehzahl erreicht*. Diese Transition legt in Abhängigkeit von der Motordrehzahl den Zeitpunkt fest, bei dem die Beschleunigung des Motors beendet und gleichzeitig die Simulation des Drehzahlfehlers gestartet werden soll. Da der Fehler des Sensors nach Spezifikation abhängig von der Drehzahl ist, ist die Betrachtung unterschiedlicher Drehzahlen für den Test relevant. Es werden exemplarisch die Drehzahlen (a) 5000 U/min, (b) 6500 U/min und (c) 7000 U/min betrachtet, so dass es für die Transition drei unterschiedliche Definitionsvarianten gibt.

Für alle weiteren Zustände und Transitionen des TP-Diagramms wird keine Differenzierung hinsichtlich alternativer Definitionsvarianten benötigt. Beispielsweise schaltet die Transition zwischen den Phasen 1 und 2 immer genau dann, wenn die erste Phase (*Motor starten*) terminiert – unabhängig davon, wie im Einzelnen die erste Phase verlaufen ist. Eine Betrachtung mehrerer Definitionen ist für die Transition deshalb weder erforderlich noch sinnvoll.

Entsprechendes gilt auch für den Zustand *Gas geben*. Ist aus der Domainexpertise heraus bekannt, dass die Bildung alternativer Varianten für diesen Zustand die Fehleraufdeckungsrate nicht oder nur äußerst unwahrscheinlich beeinflusst, kann auf die Variation verzichtet werden. Stattdessen gibt es nur eine Definition für diesen Zustand. Deshalb ist an dieser Stelle ebenfalls keine Unterscheidung mehrerer Varianten notwendig.[1]

Als Resultat der obigen Betrachtungen erhält man für den Test der Motorsteuerung ein TP-Diagramm mit den beschriebenen Zuständen und Transitionen, wobei das Verhalten der Zustände und Transitionen bis auf drei Ausnahmen eindeutig definiert ist. Die Ausnahmen sind die beiden Zustände *Motor starten* und *Drehzahlfehler simulieren* sowie die Transition *Sobald Zieldrehzahl erreicht*, für die es je 2, 5 bzw. 3 Definitionsvarianten gibt (vgl. Abbildung 5.1). Soll nun ein konkreter Testfall A definiert werden, genügt es demnach, zu diesen drei Aspekten je eine Variante auszuwählen, um einen eindeutig und präzise definierten Testfall zu beschreiben.

Der Vorteil dieser Herangehensweise wird bei der Betrachtung eines weiteren Testfalls B deutlich. Auch dieser Testfall durchläuft grundsätzlich dieselben Phasen in derselben Reihenfolge wie der Testfall A. Ebenfalls gelten auch für B die Überlegungen hinsichtlich der unterschiedlichen testrelevanten Varianten für die einzelnen Zustände und Transitionen, so dass auch für B lediglich die Varianten zu den drei oben genannten Aspekten ausgewählt werden müssen.

Das methodische Potenzial, das sich daraus ergibt, ist die Möglichkeit des Vergleiches von A und B. Anhand der (drei) definierten Variationsmöglichkei-

[1] Diese Annahme wurde für das Beispiel vereinfachend getroffen, um das Prinzip der Variation erläutern zu können. Die Praxisrelevanz dieser Annahme soll hier nicht diskutiert werden.

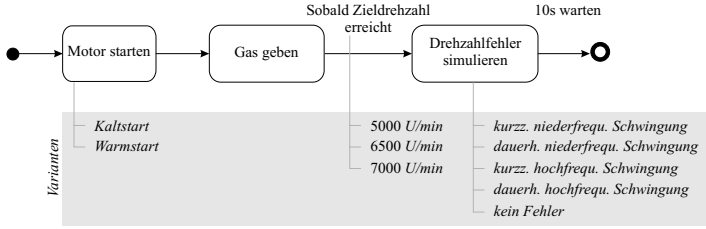

Abbildung 5.1: Beispiel mit Varianten

ten kann sehr einfach verglichen werden, inwiefern sich A und B unterscheiden. Dadurch lässt sich einerseits feststellen, ob A und B redundant sind. Andererseits kann auch analysiert werden, ob der Test ggf. weitere Testfälle erforderlich macht. Für das Beispiel der Motorsteuerung gibt es $2 \cdot 5 \cdot 3 = 30$ mögliche Kombinationen, um Testfälle zu bilden. Für reale Anwendungen ist die Kombinatorik meist deutlich höher, da die Anzahl möglicher Kombinationen exponentiell steigt. Gerade in solchen Fällen ist die Vergleichbarkeit von Testfällen besonders wichtig, um den Testaufwand mit größtmöglichem Nutzen zu optimieren.

Die beschriebene Herangehensweise hat aber auch aus praktischer Sicht Vorteile wegen der sich automatisch ergebenen Wiederverwendung. Die Modellierung des Phasenablaufs im TP-Diagramm sowie die Definition des Verhaltens einzelner Phasen und Transitionen kann für alle Testfälle gemeinsam verwendet werden: Sind die Varianten einmal definiert, ist die Erstellung neuer Testfälle durch einfache Auswahl der gewünschten Varianten zu den drei beschriebenen Aspekten ohne nennenswerten Aufwand möglich.

Verallgemeinert man dieses Beispiel, so bedeutet die Modellierung einer Menge von Testszenarien zu einem gegebenen Testproblem, dass die Struktur der Szenarien von ihrem Inhalt getrennt wird: Jedes Szenario besteht aus einer Reihe einzelner Bausteine (Zustände und Transitionen), die strukturell miteinander verknüpft sind. Die Idee der strukturierten, systematischen Ermittlung einer *Menge* von Szenarien basiert auf der gemeinsamen Definition der Struktur für alle Testfälle und der Bildung von Definitionsvarianten zu den einzelnen Bausteinen, die anschließend zu Szenarien kombiniert werden können.

Aus diesen Überlegungen folgt, dass die im Kapitel 4 vorgestellte Modellierungssprache für Szenarien erweitert werden muss, um die Variantenbildung zu unterstützen. Zu diesem Zweck werden sogenannte *Testlets* eingeführt, die die gemeinsame Struktur der Szenarien und die Modellierung von Definitionsvarianten unterstützen. Ein Szenario selbst legt damit nur noch die eigentliche

Auswahl der Varianten fest, wie in Abbildung 5.2 schematisch dargestellt ist.

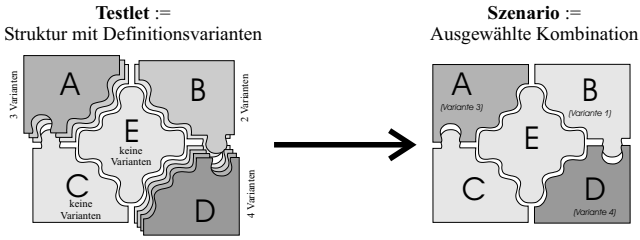

Abbildung 5.2: Struktur mit Varianten

Bevor im Einzelnen auf die Definition von Testlets und die notwendigen Änderungen der Modellierungssprache im Abschnitt 5.1.2 eingegangen wird, soll im Folgenden zunächst erläutert werden, an welchen Stellen die Variation des Verhaltens sinnvoll und notwendig ist.

5.1.1 Formen der Variation

Um zu erläutern, an welchen strukturellen Stellen es sinnvoll ist, Variationsmöglichkeiten vorzusehen, soll ein beliebiger, bereits vollständig und präzise definierter Testfall A angenommen werden, der in Form eines TP-Diagramms modelliert wurde. Nun soll ein weiterer Testfall B zu demselben Testproblem betrachtet werden. Die Frage ist nun, an welchen Stellen sich A und B unterscheiden können. Im Sinne der oben geschilderten Idee müssen all diese Unterschiede durch Varianten über einer gemeinsamen Struktur beschreibbar sein. Aus diesem Grund ergeben sich drei unterschiedliche Arten von Variationen, die im Folgenden kurz erläutert werden sollen.

Variation an Zuständen. Die erste Möglichkeit besteht in der Variation eine Zustandes, wie sie bereits im oben beschriebenen Beispiel für die Zustände *Motor starten* und *Drehzahlfehler simulieren* verwendet wurde. Das bedeutet, dass einem Zustand nicht nur ein definiertes Verhalten zugeordnet wird, sondern mehrere alternative Varianten.

Variation an Transitionen. Analog können auch für Transitionen Varianten gebildet werden. Im Beispiel wurde die Transition *Sobald Zieldrehzahl erreicht* nach diesem Schema mit drei Varianten definiert.

Pfadvariation. Die dritte Möglichkeit der Variation wurde im oben beschriebenen Beispiel noch nicht verwendet. Für zwei Szenarien A und B zu

ein und demselben Testproblem kann es auch vorkommen, dass sich die Pfade von A und B durch das TP-Diagramm an einer oder mehreren Stellen unterscheiden, d.h., die ausgehenden Transitionen an einem Zustand s und damit ggf. auch die nachfolgenden Zustände s'_A und s'_B unterscheiden sich für die Szenarien A und B.

In Abbildung 5.3 wurde das Beispiel der Motorsteuerung erweitert, indem für die zweite Phase zwei alternative Zustände *Gas geben im Leerlauf* und *Gas geben und beschleunigen* betrachtet werden.

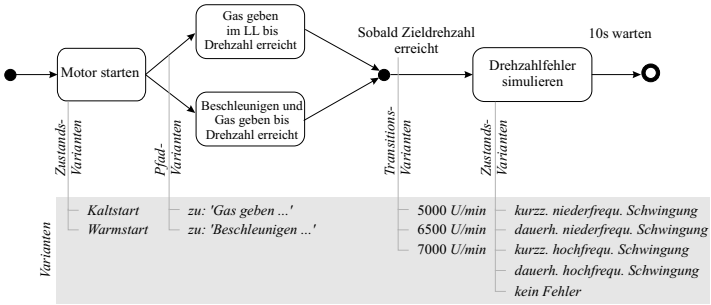

Abbildung 5.3: Pfadvarianten

Die beiden Phasen unterscheiden sich inhaltlich dadurch, dass in einem Fall der Motor im Stillstand (Leerlauf) auf die Drehzahl gebracht wird, während im anderen Fall das Fahrzeug beschleunigt. Um eine der beiden Varianten zur Modellierungszeit eines Testfalls auswählen zu können, müssen die TP-Diagramme erweitert werden: Im Abschnitt 4.5.1 wurde zunächst gefordert, dass von jedem Zustand höchstens eine Transition ausgehen darf (vgl. Ziffer T5, S. 81). Diese Einschränkung wird bei Verwendung der Pfadvariation aufgehoben, d.h., gehen von einem Zustand in einem TP-Diagramm mehrere Transitionen aus, beschreiben diese unterschiedliche Pfadvarianten und *keine* Verzweigungen des Automaten zur Laufzeit. Für jedes einzelne Szenario bedeutet dies, dass für eine Stelle im TP-Diagramm, an der eine Pfadvariation auftritt, eine der Transitionen ausgewählt werden muss, die für das Szenario betrachtet werden soll (analog zur Auswahl einer Variante zu einem Zustand). Die Konsequenz dieser Auswahl ist, dass Testfälle in der Regel nur aus *Teilautomaten* eines TP-Diagramms bestehen, da bestimmte Transitionen und Zustände nicht zur Auswahl gehören.

Aufgrund dieser Erweiterung der TP-Diagramme gibt es zwei verschiedene Arten von Verzweigungen von Transitionen: Pfadvariationen und „Verzweigungen von Transitionen zur Laufzeit“. Die Unterscheidung dieser beiden Arten erfolgt ausschließlich syntaktisch anhand des Knotens im TP-Diagramm,

von dem die verzweigenden Transitionen ausgehen: Mehrere ausgehende Transitionen an einem Zustand bilden eine Pfadvariation und erfordern die Auswahl einer der Transitionen bei der Modellierung jedes Szenarios (vgl. Abb. 5.4, linke Spalte). Mehrere ausgehende Transitionen an einer Junction entsprechen einer Verzweigung zur Laufzeit: Die schaltende Transition wird erst zur Laufzeit entschieden (vgl. Abb. 5.4, rechte Spalte).

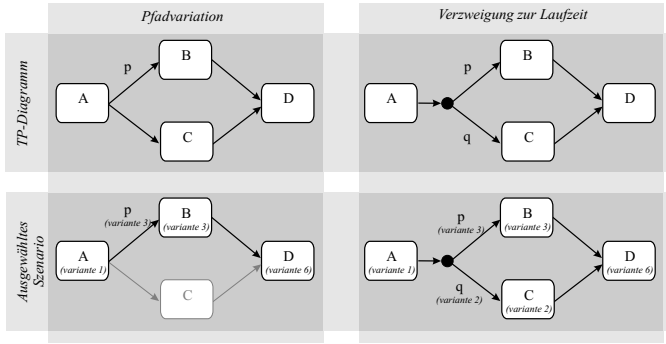

Abbildung 5.4: Pfadvariation und Verzweigung

Man beachte in dem Beispiel der Pfadvariation aus Abbildung 5.4, dass der Zustand C nicht in dem ausgewählten Szenario enthalten ist, d.h., der Testfall besteht nur aus dem Teilautomaten mit den Zuständen A, B und D und den entsprechenden Transitionen. Sobald in einem TP-Diagramm Pfadvarianten auftreten, bilden Szenarien generell nur Teilautomaten, da in jedem Szenario von jedem Zustand nur genau eine Transition ausgehen darf. Umgekehrt ist eine Teilauswahl der Transitionen an Junctions zur Modellierungszeit nicht möglich, um die klare syntaktische Unterscheidung zwischen Pfadvariation und Verzweigungen zur Laufzeit zu bewahren.

Analog ist auch für initiale Knoten die Verwendung der Pfadvariation und der Verzweigung von Transitionen möglich. Um hier die beiden Arten syntaktisch unterscheiden zu können, werden in jeder Region mehrere initiale Knoten zugelassen, d.h., die Einschränkung, dass je Region genau ein initialer Knoten existiert, wird aufgehoben (vgl. Abschn. 4.5.1, S. 84). Jeder initiale Knoten mit seinen nachfolgenden Transitionen stellt dabei eine Variante einer Pfadvariation dar, d.h., in jeder Region muss für jedes Szenario genau ein initialer Knoten ausgewählt werden. Mehrere Transitionen, die von einem initialen Knoten ausgehen, modellieren hingegen eine Verzweigung (vgl. Abb. 5.5).

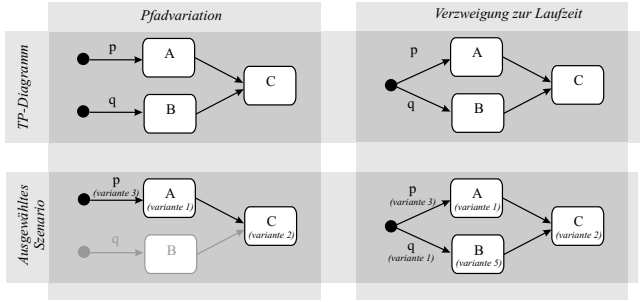

Abbildung 5.5: Pfadvariation und Verzweigung für initiale Knoten

Diskussion der Variationsarten. Von den drei verschiedenen Arten, mit denen das Verhalten variiert werden kann, ist die Pfadvariation die mächtigste. Grundsätzlich kann jede Menge von n beliebigen Szenarien durch eine Pfadvariation dargestellt werden, indem jedes Szenario in einem Zustand TC_i modelliert wird und für jedes dieser Szenarien ein separater initialer Knoten mit einer Transition $\bullet \rightarrow TC_i$ existiert. Entsprechend der oben beschriebenen Pfadvariation bei initialen Knoten gibt es demnach genau n Varianten, die trivialerweise den Szenarien entsprechen.

In ähnlicher Form lässt sich jede Zustandsvariation und jede Transitionsvariation theoretisch durch eine Pfadvariation ersetzen, indem die Pfade entsprechend „aufgefaltet“ werden. Dies ist jedoch nicht sinnvoll, da die Gemeinsamkeiten mehrerer Szenarien in der Modellierung nicht mehr durch gemeinsame Elemente dargestellt werden. Deshalb ist es bei der Definition von Varianten besonders wichtig, die Art der verwendeten Variation situationsabhängig zu wählen: Die Pfadvariation ist zwar mächtig, jedoch ist es häufig ratsam, die anderen beiden Variationsformen vorzuziehen, da sie die Struktur der Modellierung einfach halten, indem sie Gemeinsamkeiten der Szenarien zusammenfassen.

Offenbar gibt es in einem TP-Diagramm Elemente mit und ohne Variation. Die interessante Beobachtung bzgl. dieser Tatsache ist, dass genau die Elemente, für die alternative Varianten definiert wurden, die aus Sicht des Tests tatsächlich *relevanten* Bestandteile sind. In dem Beispiel aus Abbildung 5.3 gibt es beispielsweise keine Variation für den Zustand *Gas geben im Leerlauf*, d.h., für das Verhalten dieses Zustandes gibt es nur eine einzige Definition. Bei der Modellierung des Tests wurde also offensichtlich entschieden, dass eine Variation dieses Zustands auf die Fehleraufdeckungswahrscheinlichkeit nur einen vernachlässigbaren Einfluss hat. Anders verhält es sich beispielsweise bei

dem Zustand *Motor starten*. Für diesen Zustand gibt es zwei Alternativen, d.h., die Differenzierung der beiden Fälle ist bei der Modellierung offenbar als relevant erachtet worden.

5.1.2 Testlets

Die bisherige im Kapitel 4 vorgestellte Modellierungssprache für Szenarien unterstützt keine Sprachelemente, um die strukturierte Ermittlung der Menge aller testrelevanten Szenarien für ein gegebenes Testproblem basierend auf den oben beschriebenen Variationsformen zu ermöglichen. Insofern können bei der Ermittlung der relevanten Testszenarien die Variationen zwar grundsätzlich betrachtet werden – bei der konkreten Modellierung dieser Szenarien mit Hilfe der Modellierungssprache geht diese Information jedoch verloren.

Deshalb wird die Modellierungssprache nunmehr erweitert, um die für einen systematischen Test zwingend notwendige gemeinsame Betrachtung der Menge *aller* relevanten Testfälle (also der Stichprobe I; vgl. Abschn. 1.4) zu ermöglichen. Hierzu werden *Testlets* als zusätzliche Sprachelemente neben den Szenarien eingeführt. Mit Testlets werden also nicht mehr nur *einzelne* Szenarien modelliert, sondern immer die Menge aller relevanten Szenarien für ein gegebenes (Teil-)Testproblem gemeinsam betrachtet.

Es wird im Folgenden zwischen Testlets für die direkte Definition und Testlets für das Time Partitioning unterschieden. Dennoch haben alle Testlets die gleiche Grundidee, die vor der detaillierten Erläuterung der beiden Testlet-Arten zunächst beschrieben werden soll:

Gruppierung von Szenarien. Jedes Testlet gruppiert die Menge aller testrelevanten Szenarien für ein gegebenes Testproblem. Insofern sind alle Szenarien für dieses Testproblem dem Testlet fest zugeordnet. Im Beispiel aus Abbildung 5.3 wird das Testproblem „*Motor starten*“ durch ein entsprechendes Testlet modelliert, dem zwei testrelevante Szenarien zugeordnet sind: *Kaltstart* und *Warmstart*.

In vielen praktischen Fällen ist die Anzahl der einem Testlet zuzuordnenden relevanten Szenarien sehr groß (Größenordnung: mehrere hundert Szenarien). Um hierbei dennoch den Überblick zu garantieren, werden alle Szenarien eines Testlets natürlichsprachlich benannt und können bei Bedarf hierarchisch gruppiert werden. Zu diesem Zweck lassen sich Szenarien in *Szenariogruppen* zusammenfassen, die ebenfalls natürlichsprachlich benannt werden und aus einer Sequenz von Szenarien und ggf. weiteren Unterszenariogruppen bestehen können. Auf diese Weise entsteht eine hierarchische Struktur von Szenarien. Szenariogruppen sind reine Strukturierungsmittel und haben auf die Semantik der einzelnen Szenarien keinen Einfluss.

Signaturdefinition. Da ein Testlet jeweils ein bestimmtes (Teil-)Testproblem mit all seinen Szenarien modelliert, ist die Signatur aller Szenarien des Testlets immer gleich. Wenn also ein Szenario für den Zustand „*Motor starten*“ aus Abbildung 5.3 beispielsweise den Zündschlüssel als Ausgangskanal modellieren muss, um den Motorstart formal zu beschreiben, so gilt dies auch für alle anderen Szenariovarianten für diesen Zustand und damit auch für das Testlet, zu dem diese Szenarien gehören. Mit anderen Worten: Die Signatur aller Szenarien eines Testlets ist gleich und wird deshalb dem Testlet und nicht den einzelnen Szenarien zugeordnet. Alle Szenarien des Testlets müssen demnach dieselben Größen stimulieren (Stimulationsgrößen) und können potenziell dieselben Größen beobachten (Observationsgrößen).

Abbildung weiterer struktureller Gemeinsamkeiten. Ziel der Einführung von Testlets ist es, neben der Signaturdefinition auch andere strukturelle Gemeinsamkeiten aller Szenarien dem Testlet selbst zuzuordnen, während den Szenarien ausschließlich die charakteristischen inhaltlichen Unterschiede der Szenarien zugeordnet werden, anhand derer sich ein Szenario von den anderen Szenarien unterscheidet.

Im Folgenden wird diskutiert, welche strukturelle Gemeinsamkeiten dem Testlet für das Time Partitioning zugeordnet werden und warum bei der direkten Definition keine solche Gemeinsamkeiten betrachtet werden.

Testlets für die direkte Definition

Bei der direkten Definition werden die Szenarien basierend auf Gleichungssystemen definiert. Alle Szenarien eines Testlets haben dieselbe Signatur, die dem Testlet zugeordnet ist. Das heißt, die Mengen der Ein- und Ausgangskanäle sind für alle Szenarien und damit für alle Gleichungssysteme gleich. Die Szenarien müssen demnach alle dieselben Größen stimulieren (Stimulationsgrößen) und können potenziell dieselben Größen beobachten (Observationsgrößen). Bei der direkten Definition bedeutet dies, dass alle Szenarien eines Testlets jeweils Gleichungssysteme für dieselben lokalen Kanäle und Ausgangskanäle bilden. Lediglich die Definitionsterme auf den rechten Seiten der Gleichungen können sich von Szenario zu Szenario unterscheiden.

Weitere strukturelle Gemeinsamkeiten von Szenarien – neben der Signatur – werden bei der direkten Definition nicht betrachtet. Da sich die einzelnen Szenarien eines Testlets bei der direkten Definition ohnehin nur in den Definitionstermen der Gleichungen unterscheiden, müsste eine weitere Vereinheitlichung die Termstruktur der Definitionsterme betreffen (z.B. durch Vorgabe einer Gleichung „$o(t) = \mathbf{P} \cdot \sin(\mathbf{Q} \cdot t)$“ für das gesamte Testlet, so dass für jedes Szenario nur die Parameter $\mathbf{P}$ und $\mathbf{Q}$ individuell festgelegt werden können). Eine solche Vereinheitlichung wäre in den meisten Fällen zu restriktiv und wird deshalb nicht durchgeführt.

Zusammenfassend kann festgestellt werden, dass die Einführung von Testlets die Modellierung mit der direkten Definition nur maginal verändert, indem erstens alle Szenarien hierarchisch strukturiert einem Testlet zugeordnet werden und zweitens dieses Testlet die Signatur für alle Szenarien gemeinsam festlegt. Insofern ist der Vorteil der Einführung von Testlets bei der isolierten Betrachtung der direkten Definition noch nicht ersichtlich. Erst zusammen mit dem Time Partitioning kommt der wesentliche Vorteil der Testlets zum Tragen.

Testlets für das Time Partitioning

Auch beim Time Paritioning ist die Signatur eine der strukturellen Gemeinsamkeiten, die dem Testlet zugeordnet werden. Darüber hinaus wird das Verhalten beim Time Partitioning mit Hilfe eines TP-Diagramms bestimmt, für das die oben beschriebenen Formen der Variation existieren. Um diese Verhaltensvariation abbilden zu können, wird das TP-Diagramm in vier Bestandteile zergliedert:

1. **Strukturelle Definition des TP-Diagramms.** Es wird festgelegt, welche Knoten und Kanten (Zustände, initiale Knoten, finale Knoten, Junctions, Transitionen) im TP-Diagramm für die Modellierung der Szenarien benötigt werden. Existieren hierbei mehrere Transitionen, die von ein und demselben Zustand ausgehen, bilden diese automatisch eine Pfadvariation.

2. **Zuordnung des Verhaltens zu den Zuständen.** Jedem Zustand muss ein Verhalten zugeordnet werden. Da für dieses Verhalten entsprechend der Idee der Zustandsvariation potenziell mehrere Varianten existieren können, wird diese Verhaltenszuordnung dadurch modelliert, dass jedem Zustand nunmehr nicht nur ein *Subszenario* – wie im Abschnitt 4.5 beschrieben – sondern ein *Subtestlet* zugeordnet wird. Da jedes Testlet eine Menge relevanter Szenarien für ein gegebenes (Teil-)Testproblem modelliert, beschreiben die Szenarien, die zu einem solchen „Zustandstestlet“ gehören, die alternativen Varianten für das Verhalten des Zustandes.

3. **Zuordnung des Verhaltens zu den Transitionen.** Auch jeder Transition muss ein Verhalten zugeordnet werden. Wegen der Transitionsvariation genügt es nicht, das Verhalten durch die Zuordnung einer einzigen Transitionsbedingung und einer entsprechenden Aktionsdefinition zu modellieren, wie es im Abschnitt 4.5.1 beschrieben wurde. Deshalb wird das Verhalten dadurch modelliert, dass – in Analogie zur direkten Definition – mehrere alternative *Transitionsspezifikationen* gebildet werden, die natürlichsprachlich benannt und gruppiert werden können

und gemeinsam das Verhalten der Transition modellieren. Jede Transitionsspezifikation besteht aus einer Transitionsbedingung und aus der Definition der entsprechenden Aktion der Transition.

4. **Kombination von Varianten zu Szenarien.** Für jedes konkrete, relevante Szenario muss nun nur noch festgelegt werden, welche der Varianten bzgl. des Pfades, der Zustände bzw. der Transitionen ausgewählt werden sollen.

Die ersten drei Punkte sind unabhängig von einem konkreten Szenario und werden deshalb gemeinsam dem übergeordneten Testlet zugeordnet. Das Testlet legt beim Time Partitioning also neben der Signatur für alle Szenarien auch die Struktur des Automaten mit allen Zuständen, Transitionen etc. sowie die Varianten des Verhaltens für Zustände und Transitionen fest. Für ein einzelnes Szenario, das diesem Testlet zugeordnet ist, muss entsprechend nur die Variantenauswahl bzgl. der Pfad-, Zustands- und Transitionsvarianten getroffen werden.

Durch die Zuordnung von Testlets zu Zuständen ergibt sich implizit eine Hierarchie von Testlets. Wird ein Testlet eines Zustands wiederum mit dem Time Partitioning modelliert, resultiert daraus ein hierarchischer Automat. Das Testlet auf der obersten Ebene modelliert das gesamte Testproblem. Alle ausgewählten Kombinationen von Pfad-, Zustands- und Transitionsvarianten bilden zusammen die Menge der für das Testproblem relevanten Testfälle.

Zusammenfassung

Durch die Einführung von Testlets und die damit verbundene Bildung von Varianten für bestimmte Teilaspekte des Tests ist es möglich, eine Menge relevanter Testszenarien für ein gegebenes Testproblem vergleichsweise einfach zu modellieren, wobei die Gemeinsamkeiten und Unterschiede mit Hilfe der Variationen explizit gemacht werden. Die resultierende Differenzierung zwischen Testlets und Szenarien ist in Tabelle 5.2 zusammenfassend dargestellt.

5.1.3 Beispielanwendung

Die Trennung zwischen Testlets und Szenarien soll im Folgenden anhand des am Anfang dieses Kapitels diskutierten Beispiels kurz illustriert werden. Auf der obersten Ebene wird das gesamte Testproblem in Form eines Testlets dargestellt, das mit dem Time Partitioning modelliert wird. Zunächst muss hierzu die Signatur festgelegt werden. Die entsprechenden Mengen sind hier $I = \{\texttt{Drehzahl\_roh}\}$, $L = \emptyset$ sowie $O = \{\texttt{Gaspedal}, \texttt{Drehzahl}\}$. Weiterhin muss ein TP-Diagramm modelliert werden, das den Ablauf der Testfälle beschreibt. Hierzu soll das oben bereits erläuterte TP-Diagramm (Abbildung 5.6) wiederverwendet werden. Für jeden der vier Zustände des Diagramms

Element	Rolle
Testlets (allg.)	Festlegung der Signatur; Zuordnung einer (hierarchischen) Sequenz von zugehörigen Szenarien
Testlets bei direkter Definition	(keine weiteren strukturellen Gemeinsamkeiten)
Testlets beim Time Partitioning	Struktur des TP-Diagramms, Verhalten der Zustände durch Subtestlets, Verhalten der Transitionen durch (hierarchische) Sequenz von Transitionsspezifikationen
Szenario (allg.)	natürlichsprachlicher Name; Zuordnung zu einem Testlet; hierarchisch strukturiert
Szenario bei direkter Definition	Definitionsgleichungen für alle lokalen und alle Ausgangskanäle
Szenario beim Time Partitioning	Auswahl von Varianten für Pfad-, Zustands- und Transitionsvariationen

Tabelle 5.2: Testlets und Szenarien

und für jede der sieben Transitionen muss nun das Verhalten mit den jeweiligen Varianten definiert werden. Dies wird im Folgenden exemplarisch erläutert.

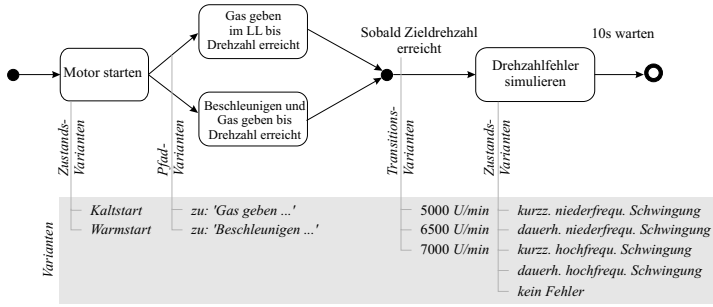

Abbildung 5.6: Pfadvarianten

Zustand *Motor starten.* Dem Zustand *Motor starten* wird ein Testlet zugeordnet, das wiederum mit Hilfe des Time Partitioning modelliert wird. Das zugehörige TP-Diagramm ist in Abbildung 5.7 dargestellt; die Bestandteile werden im Folgenden genauer beschrieben.

Im Diagramm sind zwei Zustände enthalten. Der erste Zustand *Motortemp. einstellen* regelt die Motortemperatur auf den gewünschten Wert. Das

Verhalten dieses Zustands wird durch ein Subtestlet definiert. Unabhängig davon, mit welcher der beiden Techniken das zugehörige Testlet modelliert wird und wie diese Temperaturregulierung im Einzelnen abläuft[2], müssen dem Testlet zwei alternative Szenarien zugeordnet werden: *kalt* und *warm*, um einen Kaltstart bzw. Warmstart beim Test betrachten zu können. Damit modelliert der Zustand *Motortemp. einstellen* also eine Zustandsvariation.

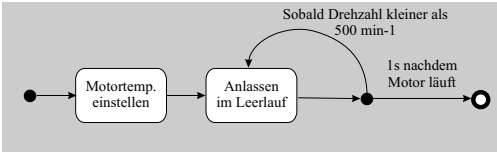

Abbildung 5.7: TP-Diagramm für *Motor starten*

Das Verhalten des zweiten Zustands *Anlassen im Leerlauf* wird ebenfalls durch ein Testlet definiert, das mit der direkten Definition modelliert wird, d.h., die zugehörigen Szenarien zu diesem Testlet werden mit Hilfe von Gleichungssystemen modelliert. Für den konkreten Zustand gibt es jedoch nur eine relevante Definition, so dass dem Testlet nur genau ein Szenario zugeordnet wird, das die Stellung des Gaspedals während des Anlassens reguliert. Wie das Gleichungssystem im Einzelnen definiert ist, soll hier nicht betrachtet werden.

Die beiden verzweigenden Transitionen im Anschluss an den Zustand *Anlassen im Leerlauf* sorgen dafür, dass der Automat genau dann terminiert, wenn der Motor die Drehzahl eine Sekunde lang über 500 min^{-1} hält. (Andernfalls sorgt die zyklische Transition dafür, dass die Phase *Anlassen im Leerlauf* neu betreten wird und damit auch die lokale Zeit zurückgesetzt wird.) Die Transitionen des TP-Diagramms unterliegen keiner Transitionsvariation, d.h., die jeder Transition zugeordnete Sequenz von Transitionsspezifikationen enthält jeweils nur genau eine Spezifikation, die die Transitionsbedingung definiert. Aktionen werden in diesem Beispiel nicht benötigt.

Damit definiert das TP-Diagramm eine Zustandsvariation, keine Transitionsvariation und auch keine Pfadvariation. Mit dieser Definition des Testlets lassen sich nun trivialerweise genau zwei Szenarien für dieses Testlet „kombinieren“: *Kaltstart* und *Warmstart* (vgl. Abb. 5.8). Diese beiden Szenarien sind die Alternativen, die auf oberer Ebene als Definitionsvarianten des Verhaltens des Zustands *Motor starten* zur Verfügung stehen, wie es bereits in der Abbildung 5.3 vorweggenommen wurde.

[2]Das hängt im Wesentlichen von der Art der Testumgebung ab. Bei den meisten Motormodellen lässt sich die Motortemperatur als explizite Größen vorgeben.

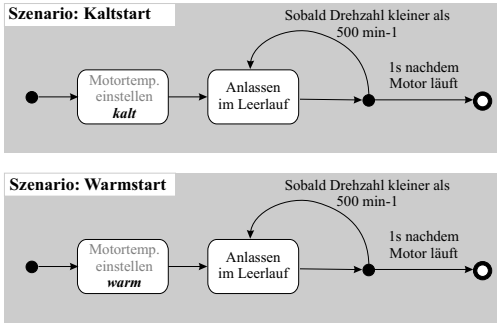

Abbildung 5.8: Szenarien für *Motor starten*

Transition *Sobald Zieldrehzahl erreicht.* Als zweites Beispiel der Modellierung des Verhaltens soll die Transition *Sobald Zieldrehzahl erreicht* betrachtet werden. Hier sollen drei Transitionsspezifikationen betrachtet werden, die sich jeweils in der konkreten Drehzahl unterscheiden, bei der die Transition schaltet. Jede der Transitionsspezifikationen enthält nur eine Transitionsbedingung, jedoch keine Aktionen. Die Bedingungen haben jeweils die Form `Drehzahl_roh >=` *limit*, wobei *limit* eine der Konstanten 5000, 6500 bzw. 7000 ist. Die entsprechenden Transitionsspezifikationen werden als *5000 U/min*, *6500 U/min* bzw. *7000 U/min* bezeichnet und entsprechen den Definitionsvarianten aus Abbildung 5.3.

Modellierung von Gesamttestfällen. Auf analoge Weise lassen sich auch die anderen Elemente des TP-Diagramms modellieren. Sind alle Zustände und Transitionen mit Hilfe von Testlets bzw. Sequenzen von Transitionsspezifikationen mit den Varianten entsprechend der Abbildung 5.3 definiert, können Gesamttestfälle – also Szenarien auf oberster Ebene – kombiniert werden. Die maximal mögliche Anzahl möglicher Kombinationen ist $2 \cdot 2 \cdot 3 \cdot 5 = 60$, d.h., es lassen sich bis zu 60 unterschiedliche Testfälle modellieren.

Die Tatsache, dass bereits bei diesem sehr einfachen Beispiel die Komplexität möglicher Kombinationen erheblich ist, macht deutlich, dass neben der wohlüberlegten Definition von Varianten zu einzelnen Modellbausteinen insbesondere auch die gezielte und systematische *Selektion* von relevanten Kombinationen ein wichtiges Problem darstellt. Dieser Aspekt soll deshalb im folgenden Abschnitt diskutiert werden.

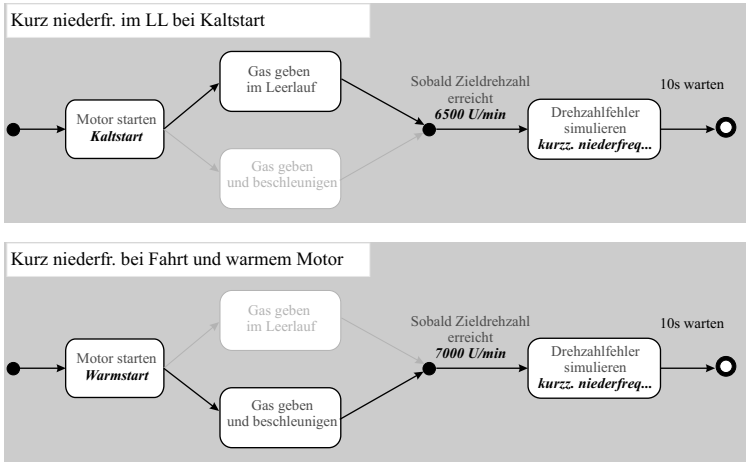

Abbildung 5.9: Szenarien für den Gesamtablauf

5.2 Kombination von Szenarien

Die Bildung von Varianten an klar definierten Stellen der Gesamtstruktur und die daraus resultierende Vergleichbarkeit von Testfällen ist eine wesentliche Voraussetzung, um bei einer großen Anzahl von Testfällen bewerten zu können, ob Testfälle redundant sind bzw. ob wesentliche Aspekte noch nicht berücksichtigt wurden. Dennoch reicht diese direkte Vergleichbarkeit in vielen Fällen nicht aus. In der bisherigen Betrachtung wurde jedes Szenario jeweils einzeln, d.h. unabhängig von den anderen Szenarien durch Auswahl entsprechender Varianten an den Variationsstellen modelliert. Die gezielte Optimierung der Menge der gewählten Testszenarien ist bislang nur durch den direktem Vergleich zweier Szenarien möglich. Der Überblick über die insgesamt ausgewählten Testszenarien wird hierbei nur unzureichend unterstützt.

Für die systematische und strukturierte Testfallermittlung wird deshalb neben der Darstellung einzelner Szenarien als „Spezialisierung“ des generischen Ablaufs (vgl. Abb. 5.9) noch eine weitere Sicht auf die Testszenarien durch TPT unterstützt, die die gemeinsame Betrachtung aller Szenarien ermöglicht. Für diese Sicht ist die Klassifikationsbaum-Methode [Gro94, Gri95] sehr gut geeignet, da sie insbesondere die Kombinatorik von Varianten, wie sie bei der Szenariodefinition benötigt wird, intuitiv und anschaulich unterstützt. Im folgenden Abschnitt wird die Klassifikationsbaum-Methode kurz erläutert. Im Anschluss wird im Abschnitt 5.2.2 vorgestellt, wie die Methode für das Time Partition Testing konkret eingesetzt wird.

5.2.1 Klassifikationsbaum-Methode

Die Klassifikationsbaum-Methode (kurz: K-Baum-Methode) wurde erstmalig in [GG93] vorgestellt und ist eine Weiterentwicklung der Category-Partition Method [OB88]. Die Methode unterstützt die funktionale Testfallermittlung und zeichnet sich insbesondere durch ein anschauliches und systematisches Vorgehen aus, bei dem der Raum möglicher Eingangsdaten für ein gegebenes Testobjekt nach verschiedenen, testrelevanten Gesichtspunkten in eine endliche Menge disjunkter Äquivalenzklassen zerlegt wird, die anschließend durch geeignete Kombinationen zu Testfällen kombiniert werden.

Der erste Schritt bei der Anwendung der K-Baum-Methode ist stets die Identifikation testrelevanter Aspekte, die auch als *Klassifikationen* bezeichnet werden. Hierfür wird die funktionale Spezifikation als Informationsquelle verwendet, um zu ermitteln, welche Klassifikationen für den Test genauer untersucht werden müssen und welche Aspekte mit hoher Wahrscheinlichkeit keinen Einfluss auf die Fehleraufdeckung haben und deshalb nicht betrachtet werden müssen. Neben der Spezifikation sind hierfür natürlich auch Erfahrungswissen aus dem Anwendungskontext und Kreativität erforderlich. Zu jedem Aspekt wird anschließend der Eingabedatenraum vollständig in disjunkte Teilmengen zerlegt. Diese Teilmengen werden auch als *Klassen* bezeichnet. Die Zerlegung wird für jeden Aspekt separat durchgeführt und ist deshalb meist einfach durchzuführen. Die Differenzierung in einzelne Klassen erfolgt so, dass für jede gebildete Klasse ein potenziell anderes testrelevantes Verhalten vermutet wird, so dass es für die Fehleraufdeckung notwendig ist, diese Klasse explizit beim Test zu betrachten.

In vielen Fällen ist es nützlich, Sub-Klassifikationen einzuführen, die nicht den gesamten Eingabedatenraum betrachten, sondern sich nur auf eine Klasse einer bereits existierenden Klassifikation beziehen. Diese rekursive Anwendung von Klassifikationen und Klassen kann solange über beliebig viele Ebenen fortgesetzt werden, bis eine präzise Differenzierung aller testrelevanten Eingabesituationen erzielt wurde. Das Resultat ist ein Baum, der aus Klassen und Klassifikationen besteht. Der Baum wird grafisch dargestellt (siehe Abb. 5.10). Die Wurzel dieses Baumes repräsentiert den gesamten Eingabedatenraum des Testobjektes, der sukzessive in Klassen zerlegt wird.

In Abbildung 5.10 ist ein Beispiel für ein Kamerasystem dargestellt, das die Aufgabe hat, auf einem Förderband vorbeifahrende Objekte hinsichtlich ihrer Größe zu klassifizieren (groß und klein). Für dieses Standardbeispiel der K-Baum-Methode werden die drei Aspekte *Farbe*, *Größe* und *Form* der Objekte betrachtet. Andere mögliche Aspekte wie beispielsweise das Gewicht der Objekte werden bewusst nicht betrachtet, da nicht zu vermuten ist, dass das Gewicht einen Einfluss auf die Größenerkennung des Kamerasystems hat. Für jeden Aspekt werden nun jeweils relevante Klassen gebildet, die beim Test explizit berücksichtigt werden sollen. Die Klasse *Dreieck* wird durch ei-

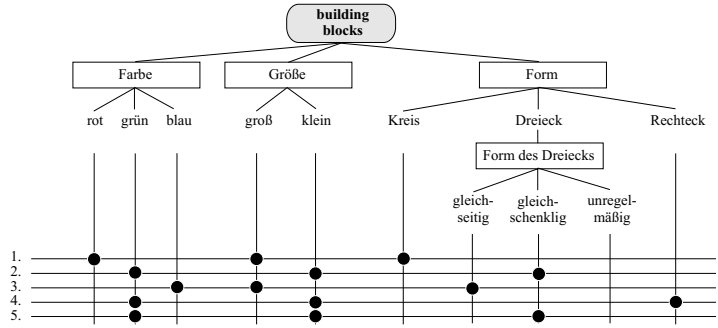

Abbildung 5.10: Klassifikationsbaum

ne Sub-Klassifikation *Form des Dreiecks* weiter verfeinert, indem drei unterschiedliche Dreiecksformen differenziert werden. Damit ergeben sich in Bezug auf die Form des Objektes insgesamt fünf alternative Möglichkeiten.

Der zweite Schritt ist nun die Bildung von Testfällen auf der Basis des Klassifikationsbaums. Ein Testfall wird hierbei durch die Kombination von Klassen unterschiedlicher Klassifikationen gebildet. Für jeden Testfall muss zu jeder Klassifikation genau eine Klasse ausgewählt werden. Zu diesem Zweck bildet der Klassifikationsbaum den Kopf einer Kombinationstabelle, in der die zu kombinierenden Klassen markiert werden. Jede Zeile in der Tabelle repräsentiert einen Testfall. Jede Spalte repräsentiert eine nicht weiter verfeinerte Klasse des Klassifikationsbaums. Die Anzahl der gebildeten Testfälle ist nicht eindeutig, sondern hängt von der Auswahl von Kombinationen des Testers ab. Die Zahl der benötigten Kombinationen wird in jedem Fall durch das Minimal- und das Maximalkriterium eingegrenzt [GG93]. Das Minimalkriterium beschreibt die Anzahl von Testfällen, die notwendig ist, um jede Klasse mindestens in einem Testfall berücksichtigt zu haben. Das Maximalkriterium ist erfüllt, wenn alle möglichen Kombinationen betrachtet wurden.

Im Beispiel in Abbildung 5.10 wurden exemplarisch fünf Testfälle gebildet. Jede Zeile der Tabelle legt zu jeder der drei Klassifikationen genau eine Klasse fest. Ein wesentlicher Vorteil der K-Baum-Methode liegt in der einfachen Möglichkeit, die gebildete Kombinatorik kompakt und anschaulich darstellen zu können. Im Beispiel ist sofort ersichtlich, dass es keinen Testfall gibt, der ein unregelmäßiges Dreieck testet, während gleichschenklige Dreiecke in zwei Fällen betrachtet werden. Außerdem erkennt man, dass vorrangig grüne Objekte betrachtet werden usw.

Auf Grund der schrittweisen Vorgehensweise sowie der grafischen Repräsentation von Klassifikationsbäumen und Kombinationstabellen kann die

K-Baum-Methode sehr gut werkzeugunterstützt werden. Hierfür wurde bereits 1995 der Klassifikationsbaum-Editor CTE [GW95] entwickelt. Im Kontext der vorliegenden Arbeit wurde dieser Editor für die Kopplung mit dem Time Partition Testing reimplementiert und um eine Reihe neuer Funktionen angereichert, die im Abschnitt 5.2.4 kurz angesprochen werden [LW00].

5.2.2 Testlets und Klassifikationsbäume

Die K-Baum-Methode soll im Zusammenhang mit dem Time Partition Testing dazu verwendet werden, um die systematische Szenariodefinition bei der Bildung von Kombinationen zu unterstützen. Da dieses Kombinatorikproblem nur für Testlets auftritt, die mit dem Time Partitioning modelliert wurden, genügt hierbei die Betrachtung solcher Testlets und ihrer zugehörigen TP-Diagramme.

Die entscheidende Idee ist nun, die Varianten, die zu einem Testlet existieren, das mit dem Time Partitioning modelliert wurde, ein einem Klassifikationsbaum zu repräsentieren und die Kombinationen der Varianten und damit die modellierten Szenarien in der zugehörigen Kombinationstabelle abzubilden. Hierfür wird für ein Testlet, das mit dem Time Partitioning modelliert wurde, jeweils ein Klassifikationsbaum nach folgendem Schema generiert:

§1: Wurzel des Baumes. Die Wurzel dieses Baumes repräsentiert das Testlet selbst.

§2: Abbildung von Zustandsvariationen. Für jede in einem TP-Diagramm enthaltene Zustandsvariation existiert im Klassifikationsbaum eine Klassifikation, die unmittelbar unter der Wurzel des Baumes angeordnet ist. Der Name der Klassifikation entspricht dem Namen des variierten Zustandes.

Bei der Zustandsvariation entsprechen die Definitionsvarianten den Szenarien des Subtestlets, das dem Zustand zugeordnet ist. Diese Szenarien können ggf. hierarchisch gruppiert werden, um auch bei einer großen Anzahl von Szenarien eine bessere Strukturierung zu ermöglichen (vgl. Abschn. 5.1.2). Aus diesem Grund sollen diese Szenariogruppen auch im Klassifikationsbaum abgebildet werden. Hierzu wird für allen Szenarien und Szenariogruppen der obersten Hierarchieebene jeweils eine Klasse im Klassifikationsbaum erzeugt, die unter der Klassifikation des Zustandes angeordnet ist. Für jede Szenariogruppe wird die zugehörige Klasse im Baum durch eine Klassifikation weiter verfeinert, die denselben Namen wie die Klasse hat, da sie lediglich den Aspekt der Zerlegung dieser Gruppe in die Einzelszenarien beschreibt. Unter dieser Klassifikation sind dann die Klassen für alle zur Gruppe gehörenden Szenarien und Szenariogruppen angeordnet (vgl. Abb. 5.11). Dieses Konstruktionsprinzip kann sich rekursiv wiederholen.

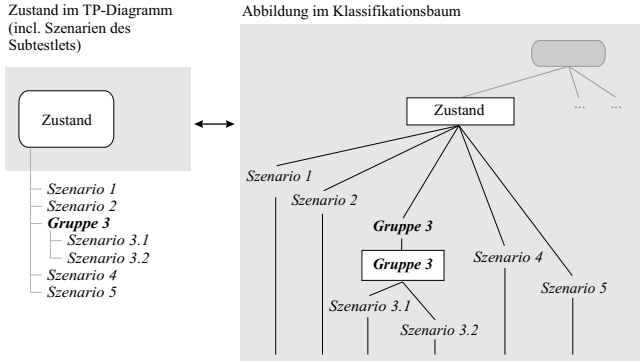

Abbildung 5.11: Abbildung von Zustandsvariationen

§3: Abbildung von Transitionsvaritionen. Für jede in einem TP-Diagramm enthaltene Transitionsvariation existiert im Klassifikationsbaum ebenfalls eine Klassifikation, die unmittelbar unter der Wurzel des Baumes liegt. Der Name der Klassifikation entspricht dem annotierten Namen an der Transition. Die Abbildung von Transitionsspezifikationen auf Klassen im Klassifikationsbaum erfolgt analog zur Abbildung von Szenarien auf Klassen für die Zustandsvariationen. Auch hier werden demnach Gruppen von Transitionsspezifikationen als Teilbäume repräsentiert (vgl. Abb. 5.12).

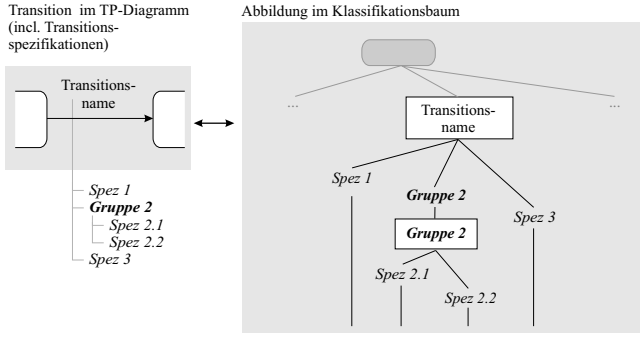

Abbildung 5.12: Abbildung von Transitionsvariationen

§4: Abbildung von Pfadvariationen. Auch für die Pfadvariationen existiert direkt unter dem Wurzelelement des K-Baums eine Klassifikation, die

den Namen „Transition from ⟨*state*⟩“ hat, wobei ⟨*state*⟩ der Name des Zustandes ist, von dem die verzweigenden Transitionen ausgehen. Da von jedem Zustand höchstens eine Pfadvariation ausgehen kann, beschreibt dieser Name die Pfadvariation eindeutig. Die Varianten der Pfadvariation werden durch die verzweigenden Transitionen beschrieben. Da diese Transitionen nur teilweise einen Namen haben, müssen für die zugehörigen Klassen im Klassifikationsbaum Namen so generiert werden, dass die eindeutige und intuitive Identifikation der Varianten im Klassifikationsbaum möglich ist. Hierzu wird bei namenlosen Transitionen der Name „to ⟨*state*⟩“ generiert, wobei ⟨*state*⟩ der Name des Zielzustandes ist. Endet die Transition an einer Junction kann sie ggf. verzweigen, so dass in diesem Fall der Name „to ⟨$state_1$⟩ & ⟨$state_2$⟩ . . . “ gewählt wird. Für Transitionen zu finalen Knoten lautet der Name „to final“. Auf diese pragmatische Art und Weise lassen sich die praktisch relevanten Fälle problemlos abbilden. Sollte es in Einzelfällen dennoch zu Mehrdeutigkeiten der Benennung kommen, ist diese jederzeit durch explizite Vergabe eindeutiger Namen für die Transitionen auszuräumen, so dass das Konzept auch in diesen Sonderfällen trägt.

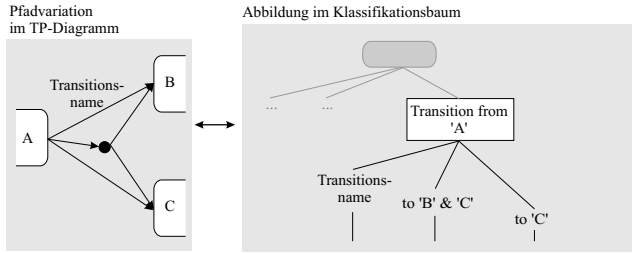

Abbildung 5.13: Abbildung von Pfadvariationen

Für jedes Testlet, das mit dem Time Partitioning modelliert wurde, kann ein K-Baum entsprechend der oben beschriebenen Regeln generiert werden. Jedes Element des Baumes hat eine Entsprechung im zugehörigen Testlet, so dass der K-Baum nur eine alternative Sicht auf ein TP-Diagramm darstellt, bei der ausschließlich der Kombinatorikaspekt abgebildet ist.

Da die Darstellung als K-Baum eine Abstraktion der Informationen des TP-Diagramms darstellt, ist die umgekehrte Richtung der Generierung von TP-Diagrammen auf Basis eines K-Baums nur durch Anreicherung der Baumin formationen möglich. So muss beispielsweise für jede Klassifikation angegeben werden, ob sie einen Zustand (mit seinen Varianten), eine Transition oder eine Pfadvariation symbolisiert. Das heißt, der Import der Informationen aus einem K-Baum in einen Automaten ist möglich, die vollständige Generierung

aller Informationen eines TP-Diagramms aus dem Baum jedoch nicht.

5.2.3 Szenarien in der Kombinationstabelle

Mit Hilfe des generierten Klassifikationsbaums lassen sich die Szenarien nun sehr einfach und übersichtlich in der zugehörigen Kombinationstabelle darstellen. Für das Beispiel aus Abbildung 5.3 enthält die Abbildung 5.14 eine (stark verkürzte) Tabelle mit ausgewählten Kombinationen. Anhand dieser Tabelle kann man sehr einfach analysieren, welche Kombinationen von Varianten bereits in Testfällen abgedeckt wurden bzw. welche Kombinationen noch betrachtet werden müssen. Im einfachsten Falle kann man beispielsweise auch erkennen, ob bestimmte Varianten noch gar nicht oder nicht in ausreichendem Maße in Testfällen berücksichtigt wurden. In der abgebildeten Beispieltabelle ist sofort ersichtlich, dass bislang kein Testfall die Zieldrehzahlen 5000 U/min bzw. 7500 U/min betrachtet hat.

Da die Kombinationstabelle nur eine alternative Sicht auf die Szenarien zu einem Testlet darstellt, das mit dem Time Partitioning modelliert wurde, können Szenarien sowohl in der „Ablaufsicht“ des TP-Diagramms (vgl. Abb. 5.9) als auch in der „Kombinationssicht“ der Kombinationstabelle definiert und visualisiert werden. In der TPT-Werkzeugumgebung ist eine aktive Kopplung der beiden Werkzeuge umgesetzt, die es jederzeit gestattet, zwischen den Sichten zu wechseln (vgl. Kapitel 8). Diese Möglichkeit ist insbesondere bei der praktischen Arbeit mit dem Time Partition Testing von erheblichem Vorteil.

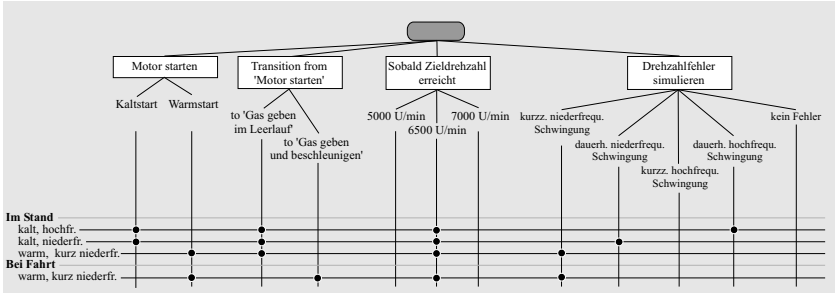

Abbildung 5.14: Kombinationstabelle

5.2.4 Automatisierung der Kombinatorik

Im Kontext der vorliegenden Arbeit wurde insbesondere auch die Frage untersucht, inwiefern die Klassifikationsbaum-Methode ohne Einbußen hinsichtlich

der Flexibilität erweitert werden kann, um die Handhabbarkeit noch weiter zu erleichtern. Insbesondere die Bildung von Kombinationen in der Kombinationstabelle ist bei komplexen, stark verzweigten Bäumen aufwändig und erfordert ein sorgfältiges Vorgehen, damit keine relevanten Fälle und Aspekte versehentlich vergessen werden.

Um den Aufwand und auch die möglichen Fehlerquellen bei der Kombination zu minimieren, wäre eine vollautomatische Generierung von Testszenarien wünschenswert, die die Relevanz bestimmter Klassen und Kombinationen automatisch berücksichtigt. Da diese Relevanz domain- und kontextabhängig ist, ist für einen solchen Vollautomatismus die Angabe spezifischer *Kombinationsregeln* erforderlich, die eine präsize Beschreibung der gewünschten Kombinatorik ermöglichen. In [LW00] wurde beschrieben, wie in dem Werkzeug CTE XL die K-Baum-Methode um die Möglichkeit der automatischen Testfallgenerierung erweitert wurde.

Die Kombinationsregeln sind eine Weiterentwicklung des Combinatorial Design Approach [CD+96b] und beruhen darauf, Terme über den Klassifikationsbaum-Elementen zu beschreiben, die eine bestimmte Kombinatorik spezifizieren. Der Name jedes Klassifikationsbaum-Elements ist hierbei ein Term und repräsentiert alle möglichen Kombinationen, die sich unterhalb des Elementes bilden lassen. Der Term *Kaltstart* repräsentiert also genau eine Kombination, die nur die Klasse *Kaltstart* selbst markiert. Der Term *Motor starten* repräsentiert zwei Kombinationen, wobei die eine die Klasse *Kaltstart* und die andere die Klasse *Warmstart* markiert. Der Wurzelknoten des Baumes beschreibt alle möglichen 60 Kombinationen. Mit diesen Basistermen lassen sich mit Hilfe zweier Operatoren $\star\star$ sowie $+\!+$ spezifische Kombinationen beschreiben. Der Ausdruck $A \star\star B$ beschreibt die Menge aller möglichen Kombinationen, die sich aus Kombinationen aus A und B bilden lassen. Der Ausdruck $A +\!+ B$ beschreibt nicht deterministisch eine Menge von Kombinationen, die alle Kombinationen von A und B mindestens einmal enthält. So legt die Generierungsregel

(“Motor starten” $\star\star$ “6500 U/min”) $+\!+$
(“Sobald Zieldrehzahl erreicht” $\star\star$ “Kaltstart”)

beispielsweise fest, dass einerseits jede Klasse zum Motorstart mit der Drehzahl 6500 zu kombinieren ist (2 Kombinationen), andererseits sollen alle Drehzahlen mit Kaltstart kombiniert werden (3 Kombinationen). Der $+\!+$-Operator sagt aus, dass die gewünschten Kombinationen beide Aspekte berücksichtigen müssen. Dies führt zu einer Menge von mindestens 4 Kombinationen[3].

Um bei dieser Generierung bestimmte Kombinationen explizit ausschließen zu können, weil sie logisch unmöglich oder fachlich irrelevant sind, können zusätzlich *logische Abhängigkeiten* formuliert werden, die in Form von aussagenlogischen Termen formuliert werden und beschreiben, welche Kombinationen von Klassen zulässig sind [LW00].

[3](Warmstart, 6500); (Kaltstart, 6500); (Kaltstart, 5000); (Kaltstart, 7000)

Mit Hilfe der Generierungsregeln und der logischen Abhängigkeiten können sehr individuelle Regeln formuliert werden, die anschließend als Basis der automatischen Generierung einer Menge von Testfällen dienen. Der Testfallgenerator des CTE XL implementiert eine heuristische kombinatorische Suche [Hal86], mit der nach einer minimalen Anzahl von Kombinationen gesucht wird, die die logischen als auch die Generierungsregeln erfüllt [LW00]. Durch diesen Automatismus kann vor allem bei großen K-Bäumen der Aufwand für die Kombination einzelner Testfälle erheblich reduziert werden. Statt in die Definition individueller Kombinationen fließen die kombinatorischen Anforderungen direkt und explizit in die Generierungsregeln ein, was die Transparenz und Qualität der Testfälle verbessert. Damit trägt die Testfallgenerierung unmittelbar auch zur Verbesserung der mit TPT modellierten Testfälle bei. Eine ausführlichere Darstellung der Details zu den Generierungsregeln und zu den logischen Abhängigkeiten ist im Rahmen dieser Arbeit leider nicht möglich.

5.3 Zusammenfassung

Neben klar definierten Sprachmitteln zur Modellierung von Testfällen ist für die Testfallermittlung insbesondere auch der Aspekt der Modellierung einer Menge von Testfällen relevant. Dieser Aspekt wird beim Time Partition Testing mit Hilfe von Variationen unterstützt, die zu den einzelnen Elementen eines TP-Diagramms gebildet und anschließend zu Testfällen kombiniert werden können. Für die Kombination, bei der insbesondere die Übersicht und Anschaulichkeit der Gesamtmenge der Testfälle im Vordergrund steht, wird das Kombinatorikproblem in einen Klassifikationsbaum eindeutig abgebildet, in dessen zugehöriger Kombinationstabelle die Definition von Testszenarien sehr einfach möglich ist.

Für die Bildung von Variationen wurde die Modellierungssprache von TPT erweitert. Die zusätzliche Ebene der Testlets dient zur Repräsentation einer Menge relevanter Szenarien für ein gegebenes Testproblem. Testlets setzen sich – analog zu Szenarien – hierarchisch zusammen, so dass die modulare Beschreibung von Testfällen gewahrt bleibt.

Mit den beschriebenen Sprachmitteln lassen sich auf jeder Ebene der Testmodellierung präzise und systematisch die relevanten Szenarien herausarbeiten, die für den Test des zugehörigen Testproblems notwendig sind. Nichtsdestotrotz obliegt es der Sorgfalt und Expertise des Testers, wie gut die gebildeten Testfälle das Testproblem tatsächlich abdecken und wie hoch somit die Testqualität ist.

Kapitel 6

Testdurchführung

Systematisch ermittelte Testfälle sind die Grundvoraussetzung für eine hohe Testaufdeckung und für einen effektiven Test. Ziel ist es hierbei, mit einer möglichst geringen Menge von systematisch ermittelten Testfällen eine hohe Menge potenzieller Fehler abzuprüfen. Aus theoretischer Sicht ist die Testfallermittlung deshalb die wichtigste Testaktivität.

In der Praxis nimmt jedoch die Testdurchführung meist einen wesentlich größeren Stellenwert als die Testfallermittlung ein, da der Prozess der Testdurchführung die Effizienz des Tests entscheidend beeinflusst. Der Gesamtaufwand des Tests gliedert sich im Wesentlichen in zwei Bereiche: den Aufwand für die Testfallermittlung sowie den Aufwand für die Testdurchführung bzw. Testauswertung. Während der erste Teil „nur“ abhängig von der Komplexität des Testobjekts sowie der Anzahl der relevanten Testfälle ist, wächst der zweite Teil zusätzlich nahezu linear mit der Anzahl der notwendigen Testwiederholungen. Die Anzahl dieser Wiederholungen hängt hierbei von der Art des Entwicklungsprozesses und nicht von der verwendeten Testmethodik ab. Bei iterativen Prozessen, wie sie bei eingebetteten Systemen wegen des Hardware-Software-Codesigns häufig auftreten, ist die Anzahl der benötigten Testwiederholungen naturgemäß sehr hoch. In solchen Fällen kann der Testaufwand für die Durchführung nur durch Automatisierung minimiert werden. Diese Tatsache erklärt den hohen Stellenwert der Testautomatisierung in der Praxis.

Um diesem Aspekt auch bei der Entwicklung des Time Partition Testing gerecht zu werden, wurde bereits bei der Definition der Modellierungssprache von TPT auf eine ausführbare Semantik Wert gelegt. In diesem Kapitel wird beschrieben, welche zusätzlichen Anforderungen sich an die Testautomatisierung aus der Praxis ergeben und wie diese Anforderungen mit TPT umgesetzt wurden.

Plattformunabhängigkeit. Die wichtigste Anforderung an das TIME PARTITION TESTING ist die Plattformunabhängigkeit, d.h., die modellierten Testfälle sollen auf unterschiedlichen Plattformen ablaufen können. Diese Anforderung ist aus zweierlei Hinsicht relevant. Erstens soll TPT keine Speziallösung für den Test mit einer bestimmten Testumgebung sein. Zweitens gibt es bei eingebetteten Systemen in unterschiedlichen Entwicklungsstadien eines Systems meist auch unterschiedliche Plattformen, auf denen das System getestet werden soll. Hierzu zählen der Entwicklungsrechner (Host), Prototypen-Plattformen (meist Echtzeit-Entwicklungsrechner) sowie die Zielplattform (Target). Daraus resultiert die Anforderung an die Testdurchführung, dass Testfälle beim Wechsel der Plattform von Plattform A nach Plattform B einfach übernommen und automatisch durchgeführt werden können, ohne dass hierzu ein Redesign der Testfälle notwendig ist.

Echtzeitfähigkeit. Eine weitere Anforderung an die Testdurchführung ist, dass die modellierten Testfälle bei der Ausführung potenziell echtzeitfähig sind. Das heißt, wenn eine Testumgebung oder das zu testende System Echtzeitfähigkeit erfordert, so soll TPT bei der Testdurchführung diese Echtzeitanforderung berücksichtigen können.

Reaktivität. Testfälle für eingebettete Systeme sind häufig reaktiv, d.h., während der Durchführung des Tests soll der Testfall auf das Verhalten des Systems reagieren können. In diesen Fällen legt ein Testfall die zu testende Situation nicht für einen definierten Zeitpunkt, sondern für einen definierten Systemzustand fest: Hat das System den Systemzustand erreicht, muss der Testfall darauf – unabhängig vom konkreten Zeitpunkt des Auftretens – reagieren können.

6.1 Architektur

Um die oben beschriebenen Anforderungen umsetzen zu können, gibt es in TPT eine strikte Trennung zwischen der Modellierungsebene und der Ausführungsebene. Die Ausführungsebene ist ausschließlich für die Testautomatisierung zuständig und stellt die technische Schnittstelle zum eigentlichen Testobjekt dar (vgl. Abb. 6.1). In der Modellierungsebene laufen alle sonstigen Aktivitäten von der Testmodellierung bis zur Testauswertung und Testdokumentation ab. Der Vorteil dieser Trennung liegt darin, dass nur die Ausführungsebene plattformabhängige und systemspezifische Gegebenheiten berücksichtigen muss.

Für die Kommunikation zwischen Modellierungs- und Ausführungsebene gibt es klar definierte Schnittstellen. Die Ausführungsebene benötigt für die automatisierte Ausführung von Testfällen eine semantisch eindeutige Be-

schreibung der Testfälle. Das Resultat der Durchführung ist die Aufzeichnung der relevanten Daten, die während des Testdurchlaufs protokolliert wurden. Diese Daten werden als *Testrecord* bezeichnet und werden in umgekehrter Richtung von der Ausführungsebene an die Modellierungsebene zurückgeliefert.

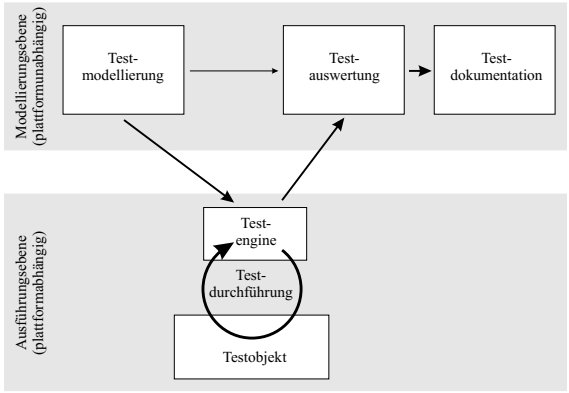

Abbildung 6.1: Modellierungs- und Ausführungsebene von TPT

Die eigentliche Testdurchführung wird innerhalb der Ausführungsebene von einer sogenannten *Testengine* durchgeführt, die als Treiber für den Testablauf dient und die Abarbeitung der Testfälle übernimmt. Diese Testengine ist plattformabhängig: Je nach Testplattform kann sowohl die Architektur und Implementierung der Automatisierung als auch das Kommmunikationsprotokoll mit dem Testobjekt verschieden sein (vgl. Abschn. 4.3.1). Grundsätzlich lassen sich zwei Arten von Testengines unterscheiden:

Vorverarbeitende Testengines führen die Testfälle nicht selbst aus, sondern übersetzen die Testfallbeschreibung in eine plattformspezifische Sprache, die anschließend von einem existierenden Compiler übersetzt oder direkt von einem Interpreter ausgeführt werden kann. Solche Testengines sind vorrangig für Targetplattformen sinnvoll, auf denen Speicher- und Zeitressourcen stark beschränkt sind. Durch die Vorverarbeitung kann der Testfall so optimiert werden, dass er nur noch die für den Testfall notwendigen Elemente der Sprachdefinition enthält.

Interpretierende Testengines implementieren unmittelbar den Algorithmus zur Ausführung eines Testfalls, ohne ein weiteres „Zwischenformat“ zu generieren. Eine solche Engine ist auch die Referenz-Testengine, die in Java implementiert ist und den Referenzalgorithmus der ausführbaren

Semantik enthält. Der Vorteil interpretierender Testengines liegt – wie bei allen Interpretern – in der schnellen Verfügbarkeit, da ein Übersetzungsschritt wegfällt. Außerdem kann für interpretierende Testengines der Referenzalgorithmus mehr oder weniger direkt übernommen werden, womit die Gefahr von Fehlern, die bei der Umsetzung einer spezifischen Testengine durchaus existiert, relativ gering ist.

6.2 Repräsentation von Testfällen

Bei der Modellierung von Testfällen werden alle Testfälle für ein gegebenes Testproblem gemeinsam modelliert, um systematisch und strukturiert eine *Menge* von Testfällen zu definieren (vgl. Kapitel 5). Aus diesem Grund liegt die Repräsentation einzelner Testfälle zunächst nicht isoliert, sondern nur im Verbund der Gesamtheit aller Testfälle zusammen mit allen grafischen Informationen zur visuellen Darstellung der Diagramme vor.

Für die Ausführung von Testfällen in einer Testengine ist es sinnvoll, die einzelnen Testfälle aus diesem Modell zu extrahieren und auf die wesentlichen, für die Ausführung relevanten Informationen zu reduzieren. Hierbei werden zusätzlich auch alle textuellen Sprachanteile parsiert und kontextgeprüft, so dass Kontexteigenschaften der Sprache in der Ausführungsebene als erfüllt vorausgesetzt werden können.

Die Repräsentation der ausführbaren Testfälle erfolgt in einer XML-basierten Zwischensprache, in der ein Testfall als abstrakter Syntaxbaum abgelegt wird. Diese XML-Struktur dient als plattformunabhängiges Austauschformat für die Beschreibung von Testfällen und wird an die Testengine übermittelt, die den Testfall ausführen soll. Die Größe dieser Repräsentationen liegt selbst bei komplexen praktischen Testfällen bei ca. 50 bis 100 kByte (unkomprimiert) und ist damit für die meisten modernen Plattformen der Testdurchführung unkritisch.

Um beide Kategorien der oben beschriebenen Testengines möglichst optimal unterstützen zu können, wurde für die Zwischensprache keine abstrakte Maschine definiert, sondern jeder XML-Testfall bildet die logische Struktur der hierarchischen, parallelen Automaten ab. Durch diese Darstellung von Testfällen auf einer sehr hohen Abstraktionsebene können insbesondere die vorverarbeitenden Testengines die Transformation optimal an die Zielsprache anpassen, so dass der Speicher- und Zeitbedarf bei der Ausführung der Testfälle minimiert werden kann. Soll eine Testengine beispielsweise für eine automatenorientierte Modellierungssprache (z.B. Stateflow von 'The Mathworks' oder Statecharts von i-Logix) implementiert werden, kann die Automatensemantik dieser Zielsprachen die Transformation von Testfällen optimieren. Die Details der XML-Zwischensprache sind im Anhang B.2 in Form der DTD beschrieben.

6.3 Testengines

Durch die Einführung von Testengines ist die Anforderung der Plattformunabhängigkeit bereits erfüllt, da jede Testengine für eine bestimmte Testumgebung die automatisierte Durchführung von Testfällen ermöglicht.

Die Echtzeitfähigkeit ist für Testengines ebenfalls gewährleistet, da der Algorithmus zur Berechnung des Verhaltens zu einem Zeitpunkt t keinerlei Schleifen dynamischer Länge enthält und somit für die notwendige Ausführungszeit eines Berechnungsschritts eine obere Schranke angegeben werden kann. Die Echtzeitfähigkeit wird jedoch nicht für alle Testengines zwingend gefordert, da sie nicht für alle Testumgebungen relevant ist. Entscheidend ist nur, dass die ausführbare Semantik von TPT-Szenarien potenziell in Echtzeit berechnet werden *kann*, wenn dies erforderlich ist.

Für die Ausführbarkeit reaktiver Testfälle muss eine Testengine in der Lage sein, Daten vom zu testenden System zu lesen. Die Semantik von Szenarien ermöglicht die Modellierung dieser Reaktivität; es hängt jedoch von der konkreten Testumgebung ab, ob die Reaktivität auch für die konkrete Testplattform verfügbar ist. Gibt es für bestimmte Systeme nicht die Möglichkeit, auf Systemgrößen zuzugreifen, können reaktive Testfälle zwangsläufig nicht durchgeführt werden. In diesem Fall ist die Menge der Eingänge für einen Testfall also leer, so dass das Verhalten von keinen Parametern abhängt und damit vorab berechnet werden kann (z.B. mit Hilfe der Referenz-Testengine). Das heißt, für Plattformen, die keine Reaktivität unterstützen, ist auch keine Implementierung einer Testengine notwendig. Stattdessen muss nur eine Möglichkeit geschaffen werden, das System mit vorab berechneten Signalverläufen für alle Ausgänge des Testfalls zu stimulieren.

Semantik von Funktionen in Testengines. Bei der Modellierung von Ausdrücken wurden Funktionen eingeführt (vgl. Abschn. 4.4.3, S. 72), die ein sehr mächtiges Instrument zur Erweiterung der Ausdrucksmöglichkeiten für Testfälle darstellen. Wie bereits beschrieben, gehört zu jeder Funktion der eindeutige Name, die Beschreibung der Ausführungssemantik sowie die Beschreibung der Kontextbedingungen.

Funktionen werden sowohl in der Modellierungsebene als auch in der Ausführungsebene benötigt: In der Modellierungsebene ist festgelegt, welche Funktionen überhaupt existieren und in Testfällen verwendet werden können. Jeder Funktionsbaustein, der eine Funktion beschreibt, legt hierzu in der Modellierungsebene den Namen der Funktion fest. Zusätzlich wird hier für die Übersetzung und Kontextanalyse eines Szenarios die Definition der Kontextbedingungen der Funktion benötigt (z.B. Anzahl und Typ der Parameter). Mit diesen Informationen lässt sich ein Testfall in die XML-Zwischensprache übersetzen.

In der Ausführungsebene ist für jede Testengine eine bestimmte Men-

ge von Funktionen implementiert, d.h., die Testengine definiert die Ausführungssemantik dieser Funktionen. Es ist jedoch nicht erforderlich und auch nicht praktikabel, dass jede Testengine *alle* Funktionen implementiert. Wird ein Testfall in eine Testengine geladen, wird geprüft, ob alle verwendeten Funktionen in der Testengine verfügbar sind oder ob der Testfall gar nicht ausgeführt werden kann. Die Identifikation der Funktionen erfolgt dabei über ihren (eindeutigen) Namen.

Ein Funktionsbaustein besteht demnach aus mehreren Teilen, die sich auf die Testmodellierungsebene (für die Modellierung und Kontextanalyse von Testfällen) und die Ausführungsebene (für die Ausführung der Funktion) verteilen. Für jede Funktion sollte – wenn technisch sinnvoll und machbar – mindestens eine ausführbare Semantik für die Referenz-Testengine implementiert sein, die als Vorlage für die Implementierung der Funktion in weiteren Testengines dient (siehe Abb. 6.2).

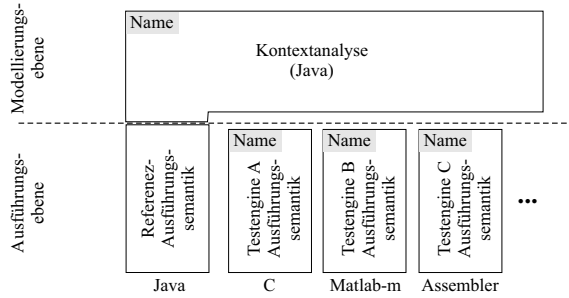

Abbildung 6.2: Architektur von Funktionsbausteinen

6.4 Repräsentation der Testrecords

Während der Ausführung eines Testfalls werden alle relevanten Daten protokolliert, sofern dies technisch möglich ist. Wie bereits erwähnt, werden diese Daten als *Testrecords* bezeichnet. Die Testrecords werden im Anschluss an die Testdurchführung für die manuelle oder automatische Auswertung des getesteten Verhaltens benötigt (vgl. Kapitel 7). Da die Testauswertung unabhängig von der konkreten Ausführungsplattform ist und demnach in der Modellierungsebene erfolgen kann (vgl. Abb. 6.1), müssen die Testrecords von der Ausführungsebene an die Modellierungsebene zurückgeliefert werden. Diese Daten beschreiben inhaltlich die durch die Testengine berechneten bzw. vom System beobachteten Ströme der einzelnen Kanäle – also Funktionen über der Zeit. Auf Grund der äquidistanten Berechnungsschrittweite des Ausführungs-

modells (vgl. Abschn. 4.3.5) können diese Daten als Folge von Stützstellenwerten tabellarisch repräsentiert werden. Das definierte Austauschformat ist entsprechend einfach und soll hier nicht genauer betrachtet werden. Wegen der meist großen Datenmengen handelt es sich um ein Binärformat.

6.5 Zusammenfassung

Die Testdurchführung nimmt in der Praxis des Tests einen hohen Stellenwert ein, da die Effizienz der Durchführung in erheblichem Maße über die erforderlichen Testaufwände entscheidet. Aus diesem Grund wurde bereits bei der Modellierungssprache für Testfälle Wert auf die Existenz einer ausführbaren Semantik gelegt, um eine automatische Testdurchführung gewährleisten zu können.

Mit Hilfe von Testengines ist es nunmehr konkret möglich, diese Testfälle in unterschiedlichen Testumgebungen und auf verschiedenen Plattformen ablaufen zu lassen. Hierbei enthalten die Testengines nur genau den plattformabhängigen Teil der Prozesskette des Tests. Die Modellierung und Übersetzung sowie die an die Testdurchführung anschließende Testauswertung und Dokumentation sind in der plattformunabhängigen Modellierungsebene umgesetzt.

Die Kommunikation zwischen der Modellierungsebene und den Testengines basiert auf zwei definierten Austauschformaten: Ausführbare Testfälle werden in einer XML-basierten Zwischensprache abgelegt und an die Testengine übertragen. Die Testrecords werden in umgekehrter Richtung in Form eines einfachen Binärformats zurückgeliefert. Diese wohldefinierten Schnittstellen gewährleisten, dass die Neuentwicklung und Integration weiterer Testengines in das Gesamtframework von TPT sehr einfach erfolgen kann.

Kapitel 7

Testauswertung

Während der Testdurchführung werden für eine gegebene Eingangssituation für ein System die zugehörigen Ausgänge beobachtet. Um entscheiden zu können, ob der Testfall einen Fehler aufgedeckt hat, wird ein Testorakel benötigt, das anhand der Ausgänge zu den gegebenen Eingängen entscheidet, ob das Systemverhalten korrekt oder fehlerhaft ist (vgl. Abschn. 1.4).

Die Definition eines solchen Testorakels ist in vielen Fällen problematisch. Hierfür gibt es vorrangig zwei Gründe. Erstens ist es in vielen Fällen nur unzureichend möglich, das Verhalten des Systems anhand der beobachtbaren Ausgänge zu beurteilen. Dieses Problem tritt beispielsweise im Zusammenhang mit eingebetteten Systemen häufig ein, wenn das System auf der Zielplattform getestet wird, auf der die Beobachtung interner Vorgänge des Systems nicht möglich ist. Zweitens ist die Angabe eines Testorakels dann schwierig, wenn die beobachteten Daten sehr komplex sind und nicht ohne weiteres eine Differenzierung in korrektes und fehlerhaftes Verhalten zulassen.

In beiden Fällen führen die beschriebenen Probleme dazu, dass bei der Testauswertung in der Praxis häufig manuell vorgegangen wird, so dass letzten Endes bei jeder Auswertung das Systemverständnis sowie eine individuelle und situationsabhängige Analyse durch einen Systemexperten oder Tester erforderlich ist. Folglich ist der notwendige Aufwand für die Auswertung der Testergebnisse entsprechend hoch, was insbesondere bei umfangreichen Wiederholungstests den Gesamtaufwand des Tests deutlich erhöht.

Für das Time Partition Testing wurde zur Minimierung des Aufwandes bei der Testauswertung eine spezielle Sprache definiert, mit der Eigenschaften zur Bewertung des Systemverhaltens formuliert und automatisch berechnet werden können. Obwohl die Darstellung und Diskussion aller Details dieser Sprache den Rahmen der vorliegenden Arbeit deutlich sprengen würde, soll in diesem Kapitel dennoch die zu Grunde liegende Idee beleuchtet wer-

den, da sie für die Gewährleistung der Durchgängigkeit der TPT-Methode zum Gesamtkonzept dazugehört.

7.1 Auswertung als Abstraktion

Das Ziel der Testauswertung ist es, aus dem Testrecord, das für gewöhnlich eine große Menge von Daten enthält, die Information zu extrahieren, ob der Testfall erfolgreich verlaufen ist oder ob ein Fehler aufgedeckt wurde. So gesehen kann die Testauswertung als eine Abstraktionsaufgabe angesehen werden, bei der aus den detaillierten, meist umfangreichen Daten zur Testdurchführung generellere Aussagen abgeleitet werden.

Wie bereits erwähnt, ist ein automatisches Testorakel, das eindeutig entscheidet, ob das Systemverhalten für einen Testfall korrekt ist oder nicht, in der Regel schwierig zu definieren. Die Daten der Testdurchführung lassen sich also häufig nicht bis zu einer klaren Ja/Nein-Aussage automatisch abstrahieren. Nichtsdestotrotz lassen sich in den meisten Fällen bestimmte Teilanalysen automatisch durchführen, die ein Indiz für die Korrektheit des Systemverhaltens darstellen. Dadurch ergibt sich im allgemeinen Fall ein Abstraktionsbaum, wie er in Abbildung 7.1 dargestellt ist, bei dem aus einem Testrecord durch sukzessive Abstraktion generelle Aussagen abgeleitet werden. Dieser Baum kann in der Regel nicht bis zu seiner Wurzel – also bis zu der Korrektheitsaussage – automatisiert werden. Dennoch ist die Teilautomatisierung sinnvoll und hilfreich, um den Testaufwand bei der Auswertung zu minimieren. Bis zu welchem Grad die Automatisierung möglich ist, hängt letzten Endes vom konkreten Testobjekt und den modellierten Testfällen ab.

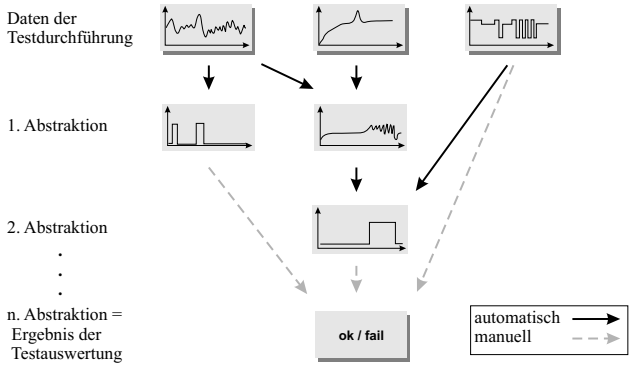

Abbildung 7.1: Abstraktion während der Testauswertung

Ziel der Auswertungssprache von TPT ist es, diesen Abstraktionsgedanken

aufzugreifen, so dass nicht zwingend die Berechnung einer Ja/Nein-Aussage zur Korrektheit im Vordergrund steht. Vielmehr soll die Sprache die Berechnung von Teilaussagen gestatten. Solche Teilaussagen sind auch für die Fehleridentifikation hilfreich, um auf die Fehlerursache rückschließen zu können, die ein Fehlverhalten provoziert hat.

7.2 Auswertungssprache

Die zentrale Designentscheidung für die Auswertungssprache hängt mit der Frage zusammen, welche Formen von Auswertungseigenschaften es geben kann bzw. welche dieser Formen unterstützt werden sollen. Um diese Frage beantworten zu können, wird im Folgenden eine Auswahl praktischer Fälle solcher Eigenschaften kurz vorgestellt.

Fall 1: Schwellwertüberschreitung. Es soll untersucht werden, ob ein Signal zu einem Kanal x einen bestimmten Schwellwert überschritten hat. Hierbei ist zum einen interessant, *ob* der Schwellwert überschritten wurde. Unter Umständen ist aber auch relevant, *wie lange* die Überschreitung auftrat bzw. *wie häufig* es zu der Schwellwertverletzung kam.

Fall 2: Vergleich mit Referenzsignalen. Ein Signal x soll bei der Auswertung mit einem Referenzsignal x_{ref} verglichen werden. Bei einem solchen Vergleich genügt teilweise die Aussage, *ob* sich die Signale unterscheiden. In anderen Fällen ist zusätzlich relevant, *wie stark* sich die Signale unterscheiden (z.B. Integral über der Differenz). Schließlich gibt es auch Fälle, in denen der Abstand der Signale selbst als Signal über der Zeit interessant ist, um kritische Stellen mit großem Abstand identifizieren zu können.

Fall 3: Folgen von Ereignissen. Es soll überprüft werden, wie viel Zeit zwischen dem Eintreten eines Ereignisses A (z.B. Signal x hat den Wert 1) und dem Eintreten eines anderen Ereignisses B vergangen ist.

Fall 4: Intervallbezogene Auswertung. Die Analysen aus den oben genannten drei Fällen können sich ggf. auch nur auf Teilintervalle der gesamten Testdurchführung beziehen, d.h., nur innerhalb bestimmter Intervalle ist die Analyse der Schwellwertüberschreitung, der Vergleich zweier Signale oder der Abstand zweier Ereignisse relevant.

Fall 5: Qualitative Aussagen. Während der Testauswertung soll entschieden werden, in welchem Modus sich eine Komponente K des System befunden hat, wobei die Modi `NORMAL`, `OVERHEAT` und `OFF` unterschieden werden. Da sich der Modus des Systems während der Testdurchführung ändern kann, soll jeweils analysiert werden, in welchem Intervall welcher Modus vorliegt.

Die Liste dieser Fälle ist nicht vollständig, zeigt aber bereits, dass es eine Vielzahl unterschiedlicher Formen von Aussagen geben kann, die für die Testauswertung relevant sind. Um all diese Fälle mit TPT berücksichtigen zu können, werden sogenannte *Assessments* eingeführt, die zur Repräsentation der abstrahierten Aussagen dienen. Assessments ordnen Informationen bestimmten Zeitintervallen zu, wobei jedes Assessment zu mehreren disjunkten Intervallen Informationen enthalten kann. Es gibt zwei Arten von Assessments: *Kanäle* und *Variablen*.

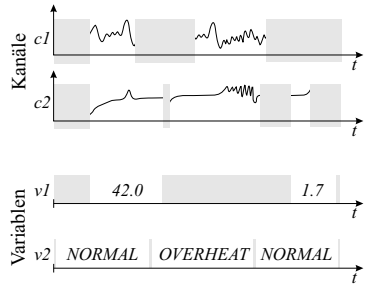

Abbildung 7.2: Kanäle und Variablen

Kanäle beschreiben im Kontext von Intervallen kontinuierliche Verläufe, d.h., sie sind eine Verallgemeinerung des Kanalbegriffs der Testdurchführung. Variablen beschreiben hingeben ausschließlich konstante Werte, die Intervallen zugeordnet werden (vgl. Abb. 7.2). So gesehen, sind Variablen spezielle Kanäle, deren Verläufe je Intervall konstant sind. Es wird sich jedoch zeigen, dass Variablen auf Grund ihrer Konstanz anders verwendet werden können als Kanäle, was die klare Differenzierung von Variablen und Kanälen erforderlich macht.

Kanäle können wie bei der Testdurchführung die Typen `float`, `int` und `boolean` haben. Für Variablen gibt es zusätzlich noch die Typen `string` und `void` sowie die Möglichkeit zur Definition eigener Aufzählungstypen. Alle Assessments müssen explizit deklariert werden:

```
channel float c1;    // Float-Kanal
channel int c2;      // Integer-Kanal
channel boolean c3;  // boolscher Kanal
float x;          // Float-Variable
boolean b;        // boolsche Variable
string name;      // String-Variable
void xInterval;   // Var. ohne assoz. Wert

enum { NORMAL, OVERHEAT, OFF } MODE; // Typdekl.
MODE componentMode;
```

7.2.1 Intervalle der Auswertung

Mit Hilfe von Assessments (Kanälen und Variablen) lassen sich abstrahierte Aussagen der Testauswertung einzelnen Intervallen zuordnen, wobei die Form dieser Aussagen von einfachen Werten über Aufzählungen bis hin zu kontinuierlichen Verläufen variieren kann. Das heißt, Assessments repräsentieren sowohl Werte als auch die Definitionsintervalle, zu denen die Werte gehören. Die Zuweisung von Werten zu Assessments erfolgt durch einfache Gleichungen der Form $\langle assessment \rangle = \langle value \rangle$, wobei das Definitionsintervall implizit aus dem Kontext der Zuweisung hervorgeht. Zur Festlegung dieses so genannten *Kontextintervalls* gibt es eine spezielle Anweisung „`during` $\langle assessment \rangle$ $\{\langle body \rangle\}$“. Das Assessment in dieser Anweisung legt mit seinen Definitionsintervallen fest, in welchen Kontextintervallen der Body (chronologisch sortiert) wiederholt abgearbeitet werden soll.

Ein Beispiel: Sei `i` ein Assessment, das in den Intervallen $[1, 4)$ und $[6, 9)$ definiert ist. Seien weiterhin `x`, `c1` und `c2` wie oben deklarierte Assessments. Die Anweisung

```
during i {
  x = 2; // Zuweisung in allen Def.intervallen von i
  c1(t) = sin(t);
  c2(t) = (int)(x * c1(t));
}
```

bewirkt, dass die geklammerten Anweisungen erst im Kontext von $[1, 4)$ und dann von $[6, 9)$ ausgeführt werden. Dadurch wird der Variablen `x` der Wert 2 in beiden Intervallen zugeordnet. Der Kanal `c1` wird in beiden Intervallen als Sinusfunktion definiert, wobei an der linken Intervallgrenze jeweils $t = 0$ gilt (lokale Zeitachse). Analog wird $c2$ definiert, wobei in der Definition auf die Werte von x und $c1$ Bezug genommen wird. `during`-Anweisungen können auch verschachtelt werden und schränken dadurch das Kontextintervall weiter ein.

Um die Hierarchie von Abstraktionen zu unterstützen, kann bei der Definition von Assessments – wie bei `c2` oben bereits gezeigt – auf andere Assessments Bezug genommen werden. Ist der Wert eines Assessments `x` im aktuellen Kontextintervall eindeutig, kann der Wert direkt referenziert werden. Ist der Wert nicht eindeutig, weil es im Kontextintervall mehrere Teilintervalle gibt, denen ein Wert für `x` zugeordnet wurde, muss die Form `x`$[n]$ verwendet werden. Dieser Ausdruck beschreibt den Wert von `x` im n-ten Definitionsintervall von `x` innerhalb des Kontextintervalls:

```
x = 2;
during i {
  y = x; // okay: x ist eindeutig
}
z = y; // nicht okay: y in zwei Interv. def.
z = y[0]; // okay: entspr. y im Interv. [1,4)
```

In ähnlicher Form kann neben dem Zugriff auf die Werte der Assessments auch auf die Intervalle durch Angabe spezieller Postfixe Bezug genommen werden. Wenn *v* den Wert eines Assessments in einem bestimmten Intervall beschreibt, dann beschreiben *v*.`length` die Länge dieses Intervalls sowie *v*.`left` bzw. *v*.`right` den linken bzw. rechten Randzeitpunkt des Intervalls (relativ zum aktuellen Kontextintervall). Weiterhin beschreibt `x.size` für ein Assessment `x` die Anzahl der Intervalle, in denen im aktuellen Kontextintervall ein Wert für `x` definiert wurde:

```
during i {
  x = i.length; // in beiden Durchläufen gilt x==3
  y1 = i.size; // y1==1: je nur ein Interv. sichtb.
}
y2 = i.size; // y2==2: beide Interv. sichtbar
z = i[1].left; // linker Rand des zweiten Interv.: z==6
```

Um bei der Testauswertung auf die Werte der Testdurchführung Bezug nehmen zu können, können Signalverläufe mit einem speziellen `import`-Statement importiert und Kanälen zugeordnet werden. Die genaue Syntax dieser Statements wird hier nicht erläutert.

7.2.2 Integration in die Testmodellierung

Bei der Testfallmodellierung wurden die Testszenarien in Form von hierarchischen Teilszenarien zusammengesetzt, wobei die Teilszenarien unterschiedliche Phasen des Tests modellieren. Entsprechend ihrer Rolle gibt es für Teilszenarien in der Regel auch spezifische Auswertungseigenschaften, die nur für die zugehörige(n) Phase(n) gelten. TPT bietet deshalb die Möglichkeit, Auswertungseigenschaften direkt einem Teilszenario zuzuordnen. Für den gesamten Testfall werden diese Eigenschaften dann automatisch zusammengefügt, wobei Eigenschaften, die nur bestimmte Phasen betreffen, in ein entsprechendes `during`-Statement eingebettet werden. Hierzu wird für die betreffenden Zustände während der Testdurchführung mit Hilfe eines automatisch erzeugten, eindeutigen Hilfskanals `state_`*id* protokolliert, wann der Zustand aktiv bzw. inaktiv war. Wird dieser Kanal während der Testauswertung importiert, wird er automatisch genau in den Intervallen definiert, in denen der Zustand aktiv war. Die Anweisung `during state_`*id* {...} berechnet die eingebetteten Auswertungseigenschaften demnach genau in den Kontextintervallen, in denen der Zustand aktiv war.

Aus methodischer Sicht sind diese technischen Details jedoch irrelevant. Bei der Testfallmodellierung können Auswertungseigenschaften problemlos den Teilszenarien zugeordnet werden, für die sie relevant sind. Dadurch ist die Möglichkeit gegeben, die einen Testfall und die zugehörigen Auswertungseigenschaften parallel zu spezifizieren.

7.3 Online- vs. Offline-Auswertung

Die oben definierte Auswertungssprache für TPT wird *offline* – also im Anschluss an die Testdurchführung – durchgeführt. Bestimmte Informationen, wie beispielsweise die Berechnung einer Signaldifferenz oder die Einhaltung einer Abfolge von Ereignissen, lassen sich grundsätzlich auch *online* – also parallel zur Testdurchführung – in Form von lokalen Hilfskanälen berechnen. Die Frage, welcher Weg der Analyse angemessener ist, hängt von der konkreten Anwendung ab.

Die Offline-Auswertung hat den Vorteil, dass Auswertungseigenschaften nicht mit der Modellierung des Testfalls vermischt werden, was die Übersicht bei der Testfallmodellierung bewahrt. Ein weiterer Vorteil: Die Offline-Auswertung benötigt keine Ressourcen auf der Testplattform.

Dennoch gibt es Fälle, in denen bereits während der Testdurchführung bestimmte Auswertungseigenschaften berechnet und protokolliert werden müssen. Sind z.B. interne Signale des Systems nur für die Auswertung relevant, müssen diese Signale trotzdem im Testfall mit angegeben werden, damit die Testdurchführung die Signale protokolliert.

Die Testauswertung von TPT erfolgt offline. Ist die Berechnung bestimmter Informationen während der Testdurchführung notwendig, müssen diese Aspekte mit in die Testfallmodellierung integriert werden.

7.4 Zusammenfassung

In diesem Kapitel wurde die Idee der automatisierten Testauswertung von TPT kurz vorgestellt. Sie stellt einen wichtigen Baustein für die Durchgängigkeit der TPT-Methode dar.

Die Testauswertung basiert auf einer spezialisierten Sprache, die die Zuordnung von Informationen zu Zeitintervallen in Form sogenannter Assessments ermöglicht. Bei der Berechnung von Assessments kann auf andere Assessments Bezug genommen werden, so dass die ursprünglichen Daten der Testdurchführung nach und nach immer weiter abstrahiert werden. Hierfür stehen umfangreiche Sprachkonzepte zur Verfügung, von denen die wichtigsten Elemente kurz erläutert wurden.[1]

Es wurde gezeigt, dass sich das Konzept der Testauswertung von TPT in das bisherige Modellierungskonzept einbettet, indem Auswertungseigenschaften unmittelbar mit der Modellierung von Szenarien verknüpft werden können. Dadurch können die Testfallmodellierung und die Spezifikation der Auswertungseigenschaften parallel erfolgen.

Eine detaillierte Beschreibung der Konzepte zur Testauswertung würde den Rahmen dieser Arbeit sprengen und ist deshalb leider nicht möglich.

[1] Weitere Konzepte wie z.B. reguläre Ausdrücke, Funktionsbausteine und Dokumentare können nicht erläutert werden, da dies den Rahmen der Arbeit deutlich sprengen würde.

Kapitel 8

Das Testwerkzeug TPT

Für die praktische Anwendbarkeit und Akzeptanz einer Methodik ist eine geeignete Werkzeugunterstützung in jedem Fall wichtig. Im Zusammenhang mit TPT ist die Existenz eines Werkzeugs außerdem erforderlich, um die Automatisierung einzelner Schritte des Tests und die damit verbundene Effizienzsteigerung realisieren zu können.

Die Werkzeugumgebung „TPT“ wurde im Kontext der vorliegenden Arbeit entwickelt, um die praktische Anwendbarkeit der theoretischen Konzepte verifizieren zu können. Die Entwicklung der Werkzeugumgebung wurde von der DaimlerChrylser AG finanziert und ihr praktischer Einsatz im Rahmen von konkreten Entwicklungsprojekten unterstützt.

Durch die parallele Entwicklung der Konzepte und der Werkzeugumgebung konnten konzeptionelle Ideen schnell verwirklicht und praktisch evaluiert werden, was die konkrete Gestaltung der beschriebenen TPT-Methode erheblich beeinflusst hat. Im folgenden Abschnitt soll die TPT-Werkzeugumgebung kurz vorgestellt werden, um einen Überblick über die momentan bereits umgesetzten Funktionalitäten zu geben.

8.1 Testfallmodellierung

In Abbildung 8.1 ist der Haupteditor von TPT dargestellt, mit dem Testfälle modelliert werden können. Im *Testlet-Browser* (1) auf der linken Seite im oberen Bereich ist die Hierarchie der Testlets abgebildet, aus denen sich die Testfälle zusammensetzen. Für jedes Testlet ist anhand des Icons erkennbar, ob es durch die direkte Definition oder durch ein TP-Diagramm modelliert wurde. Durch Auswahl eines Testlets im Browser werden die zugehörigen Szenarien im darunter liegenden Szenario-Browser (2) und auf der rechten Seite des Editors (3) bzw. (4) sichtbar.
Der *Szenario-Browser* (2) enthält zu dem aktuell ausgewählten Testlet die

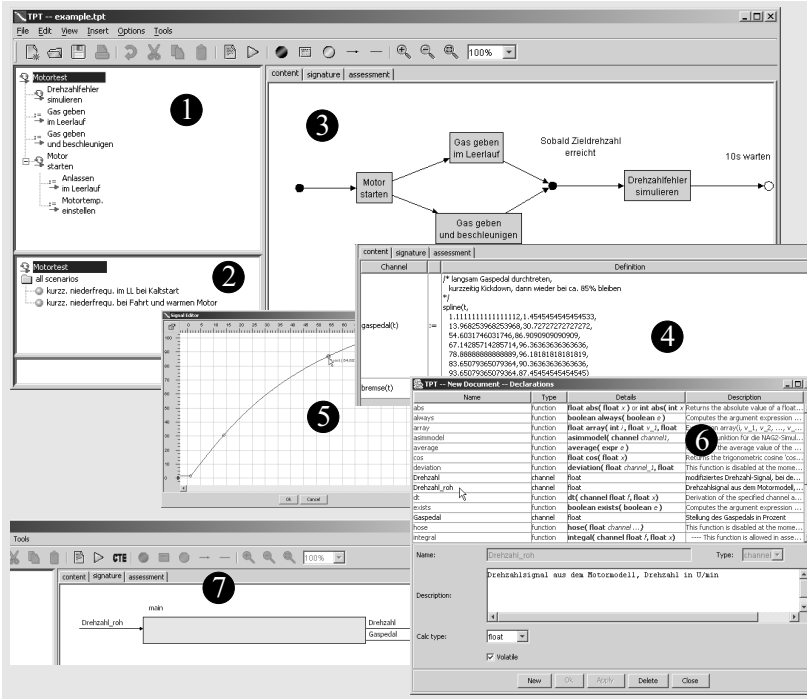

Abbildung 8.1: TPT-Editoren zur Testfallmodellierung

Hierarchie der zugehörigen Szenarien und Szenariogruppen. In der Abbildung gibt es nur zwei Szenarien, die unmittelbar auf oberster Ebene (unter der Hauptgruppe *all scenarios*) angeordnet sind. Außerdem enthält der Browser als oberstes Element immer das Testlet selbst. Je nach Auswahl in diesem Browser ändern sich die Details der auf der rechten Seite (3) bzw. (4) dargestellten Informationen.

Der *TP-Editor* (3) hat zwei Aufgaben und entsprechend auch zwei unterschiedliche „Look-and-Feels“. Ist im Szenario-Browser das Testlet selektiert, kann im TP-Editor das zugehörige TP-Diagramm editiert werden, d.h., es können Zustände, Transitionen und Junctions hinzugefügt, gelöscht, verschoben oder verändert werden. Ist im Szenario-Browser hingegen ein Szenario selektiert, zeigt der TP-Editor die Zuordnung von Subszenarien zu den einzelnen Zuständen und Transitionen für dieses Szenario an (vgl. Abb. 8.2). Zu

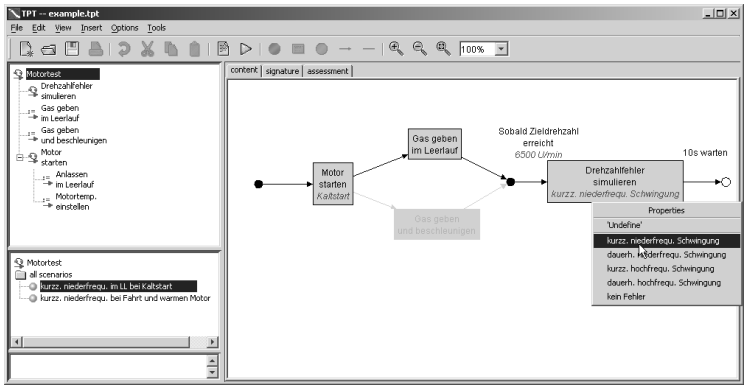

Abbildung 8.2: TP-Editor mit ausgewähltem Szenario

jedem Zustand und jeder Transition kann hier per Kontextmenü festgelegt werden, welches Subszenario bzw. welche Transitionsspezifikation für das aktuell fokussierte Szenario ausgewählt werden soll. Im abgebildeten Beispiel ist das Kontextmenü für den Zustand *Drehzahlfühler simulieren* geöffnet. An den Elementen *Motor starten*, *Sobald Zieldrehzahl erreicht* und *Drehzahlfühler simulieren* sind die aktuell ausgewählten Varianten im Diagramm eingeblendet. Alle anderen Elemente haben keine Variationen.

Der *Gleichungseditor* (4) für die direkte Definition besteht aus einer Tabelle, in der für jedes Szenario, das im Szenario-Browser ausgewählt ist, die Definitionen der Ausgangs- und lokalen Kanäle geschrieben werden. Diese Kanalverläufe lassen sich auch mit Hilfe des *Signaleditors* (5) grafisch editieren, wobei nur unabhängige Kanäle in Form von Linienzügen und Splines editierbar sind.

Mit dem *Deklarationseditor* (6) können Kanäle und Konstanten deklariert werden. Der *Signatureditor* (7) dient zur Festlegung der Schnittstelle für ein Testlet. Im abgebildeten Beispiel hat das Testlet einen Eingang und zwei Ausgänge.

8.2 Systematische Auswahl

Für die systematische Auswahl kann die Kombinatorik auch mit dem CTE XL durchgeführt werden. Ist im Testlet-Browser ein Testlet ausgewählt, das mit dem Time Partitioning modelliert wurde, kann über einen Menüpunkt ein Klassifikationsbaum generiert werden, der im CTE XL geöffnet wird und aktiv mit dem TPT-Datenmodell gekoppelt ist (vgl. Abb. 8.3). Wird in der Kombinationstabelle eine Markierung geändert, ändert sich dabei automatisch auch die Auswahl in TPT.

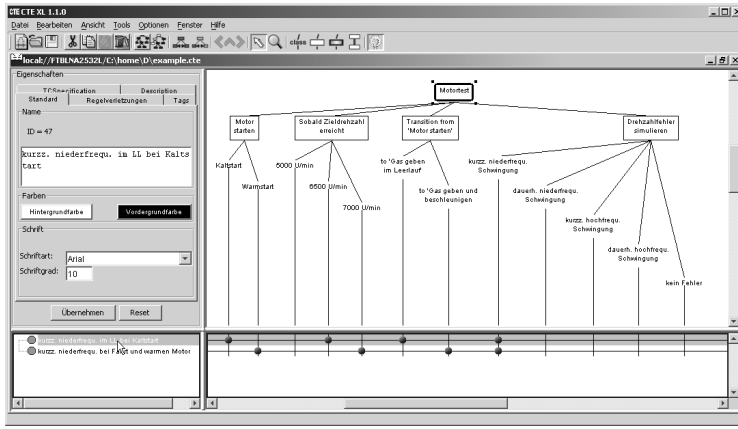

Abbildung 8.3: Systematische Auswahl mit dem CTE XL

8.3 Testdurchführung

Um die modellierten Testfälle ausführen zu können, muss im Testlet-Browser das Testlet der obersten Ebene selektiert werden. Im Szenario-Browser kann dann ein Szenario oder auch eine Szenariogruppe ausgewählt werden. In letzterem Fall werden alle in der Gruppe enthaltenen Szenarien ausgeführt.

Bei der Testdurchführung werden die Testszenarien zunächst in die XML-Zwischensprache für Szenarien übersetzt und kontextgeprüft. In einem spezi-

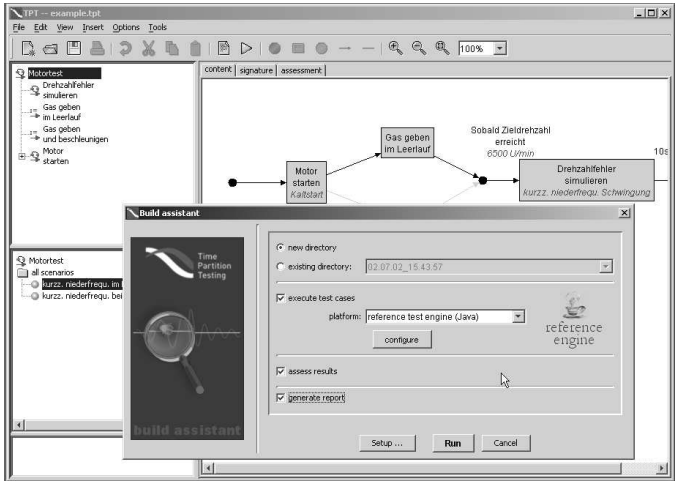

Abbildung 8.4: Starten der Testdurchführung

ellen Dialog (vgl. Abb. 8.4) wird unter anderem festgelegt, mit welcher Testengine die Testfälle ausgeführt werden sollen. Weiterhin kann entschieden werden, ob die Testauswertung im Anschluss an die Durchführung automatisch gestartet und ob ein Testreport zu den ermittelten Daten generiert werden soll.

8.4 Testauswertung

Für die Spezifikation der Auswertungseigenschaften können jedem Szenario jedes Testlets der Hierarchie individuelle Eigenschaften zugeordnet werden. Hierfür gibt es einen speziellen *Auswertungseditor* (vgl. Abb. 8.5), der zu dem im Szenario-Browser ausgewählten Szenario die Definition der Auswertungseigenschaften enthält. Dieser Editor ist zweigeteilt: In der oberen Hälfte des Editors sind die spezifischen Auswertungseigenschaften enthalten, die für das ausgewählte Szenario spezifiziert wurden. Im unteren Teil sind alle implizit mitgeltenden Eigenschaften für dieses Szenario aufgelistet. Dies sind zum einen alle Eigenschaften, die für Subzustände des Szenarios spezifiziert wurden und zum anderen Eigenschaften, die für Szenariogruppen definiert wurden, in denen das aktuell ausgewählte Szenario enthalten ist. Dadurch können Eigenschaften, die für alle Szenarien einer Gruppe gleichermaßen berechnet werden sollen, direkt der Gruppe zugeordnet werden.

In dem Beispiel in Abb. 8.5 wurden Auswertungseigenschaften der Haupt-

gruppe des Zustandes *Motor starten* zugeordnet. Damit gelten diese Eigenschaften implizit für alle Szenarien zu dem Zustand *Motor starten*. Für das in der Abbildung selektierte Szenario ist *Motor starten* ein Subzustand, so dass die Eigenschaften für das dort ausgewählte Szenario implizit mit berücksichtigt werden.

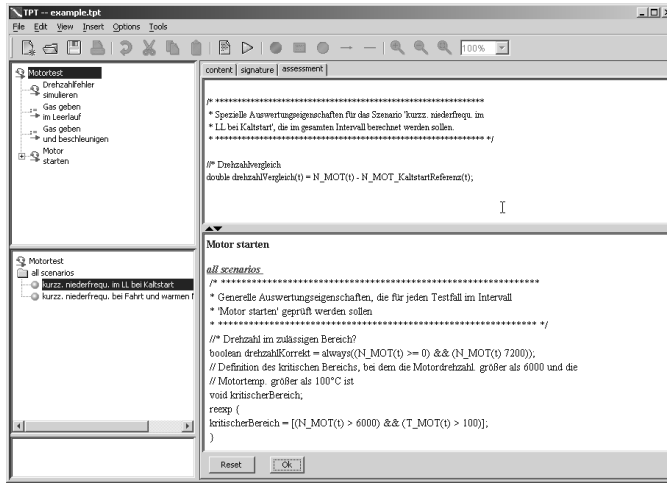

Abbildung 8.5: Spezifikation der Auswertungseigenschaften

Das Zusammensetzen der einzelnen Auswertungseigenschaften zu einem Gesamtskript sowie die Berechnung der Eigenschaften erfolgt automatisch im Anschluss an die Testdurchführung, falls im Dialog aus Abb. 8.4 die entsprechende Option ausgewählt wurde.

Kapitel 9

Zusammenfassung und Ausblick

Das Testen im Allgemeinen – und insbesondere auch im Kontext eingebetteter Systeme – ist die in der Praxis wichtigste Aktivität der analytischen Qualitätssicherung. Auf der anderen Seite ist das Testen jedoch auch eine im Vergleich zu anderen Disziplinen des Software-Engineering meist pragmatisch und wenig systematisch durchgeführte Aktivität. Um diese Diskrepanz zu beheben, wurde ein neuartiges Testverfahren – das Time Partition Testing – entwickelt, das das Testen eingebetteter Systeme auf eine systematische und transparente Art und Weise gestattet.

Ein wesentlicher Schwerpunkt bei der Entwicklung des Time Partition Testing lag auf der Durchgängigkeit des Ansatzes, d.h., das Verfahren legt die Vorgehensweise sowohl für die Testfallmodellierung als auch für die nachfolgenden Schritte der Testdurchführung, Testauswertung und Testdokumentation[1] fest und gibt dadurch einen klaren Rahmen für alle zentralen Testaktivitäten vor. Dies trägt wesentlich zu einer Verbesserung der Transparenz und der Steuerbarkeit des Testprozesses bei.

Um den Anforderungen aus der Praxis Rechnung zu tragen, wurde bei der Entwicklung des Time Partition Testing und der zugehörigen Modellierungssprache für Testfälle auf eine ausführbare Semantik Wert gelegt. Dadurch lassen sich vollständig modellierte Testfälle automatisch ausführen, was die Effizienz der Testdurchführung erheblich verbessert und dadurch den Schwerpunkt der kreativen Auseinandersetzung mit dem Testen auf die für die Testqualität wesentliche Aufgabe der systematischen Testfallermittlung verschieben hilft.

Die Modellierungssprache für Testfälle ermöglicht die Definition von Test-

[1] siehe unten im Abschnitt „Geplante Arbeiten“

fällen für kontinuierliches, reaktives Verhalten. Die Modellierung basiert auf der abstrakten Beschreibung von Testfällen in Form von hierarchischen, parallelen Automaten, die den generellen Ablauf eines Testfalls widerspiegeln. Diese Beschreibung kann mit TPT bis auf die Ebene konkreter Daten verfeinert werden, so dass alle Details präzise beschrieben sind und der Testfall automatisch ablaufen kann. Die abstrakte Betrachtung eines Testfalls bleibt davon unberührt, was insbesondere für das Verständnis der getesteten Situation für Domain-Experten relevant ist, die mit der Modellierungssprache von TPT im Einzelnen nicht vertraut sind.

Ein fundamentales Konzept von TPT ist die gemeinsame Modellierung einer Menge von Testfällen. Hierbei werden Testfälle nicht unabhängig voneinander betrachtet. Stattdessen werden sie stets durch Bildung von Varianten zu einem generelleren Ablaufschema modelliert. Dadurch sind Testfälle sehr einfach und schematisch vergleichbar, was die Übersichtlichkeit und Transparenz über die erzielte Gesamtabdeckung vereinfacht.

Die Definition von Testfällen wird durch die Einführung von Varianten zu einer Kombinatorikaufgabe. Aus einer meist sehr großen Menge möglicher Kombinationen muss eine Teilmenge relevanter Kombinationen ausgewählt werden muss. Für diese Aufgabe ist die Klassifikationsbaum-Methode in TPT integriert. Die K-Baum-Methode ist für die beschriebene Kombinatorikaufgabe sehr gut geeignet. Spezifische Erweiterungen der K-Baum-Methode zur Beschreibung von logischen Abhängigkeiten zwischen den Varianten sowie von Generierungsregeln, die im Kontext der Entwicklung von TPT entstanden sind, unterstützen die Kombinatorikaufgabe zusätzlich.

Neben der Möglichkeit, Testfälle vollautomatisch durchführen zu können, trägt auch die automatisierte Testauswertung von TPT dazu bei, dass der erforderliche Testaufwand auf ein angemessenes Maß beschränkt wird. Läuft eine Systementwicklung iterativ ab und sind dadurch viele Wiederholungstests notwendig, trägt diese Automatisierung zu einer erheblichen Arbeitserleichterung und einer damit verbundenen Akzeptanzverbesserung des Testansatzes bei.

TPT in der Praxis

Das Time Partition Testing ist kein bloßes theoretisches Konzept. Es wurde in unmittelbarem Bezug zur praktischen Anwendung entwickelt. Bereits die ursprünglichen Ideen basierten auf diversen praktischen Erfahrungen in Bezug auf den Test im Kontext der Steuergeräteentwicklung bei Daimler-Chrysler.

Im Laufe der Entwicklung des Testverfahrens und der Implementierung einer ersten prototypischen Werkzeugumgebung entstanden eine Reihe von Kooperationen mit Entwicklungsprojekten, die die Richtung der Weiterent-

wicklung des Ansatzes entscheidend mitbestimmt haben. Der Anspruch der Praxistauglichkeit war von Anfang an ein entscheidender Faktor der Entwicklung, der sich auch positiv auf die Weiterentwicklung des Verfahrens ausgewirkt hat.

TPT befindet sich heute bereits im praktischen Einsatz bei der Serienentwicklung der Steuergerätesoftware für ein neues Automatikgetriebe und hat in diesem Kontext seine Praxistauglichkeit bereits unter Beweis gestellt. Dies hat dazu geführt, dass TPT für diese Serienentwicklung bereits als zentrale Komponente für den funktionalen Test verwendet wird; Kooperationen mit weiteren Entwicklungsprojekten aus dem Umfeld der Innenraumsteuerung sind bereits gestartet, Kooperationen mit weiteren Bereichen sind in Planung.

Aktuelle und geplante Arbeiten zu TPT

Neben kleineren Verbesserungen und Erweiterungen der Werkzeugumgebung, die für die praktische Handhabbarkeit des Verfahrens relevant sind, liegt das Hauptaugenmerk der gegenwärtigen Weiterentwicklung von TPT auf der Integration der Testdokumentation in das TPT-Framework. Auf Grund der hohen Anzahl anfallender Daten ist die intuitive Darstellung und Aufbereitung dieser Informationen für den Anwender eine nicht triviale Aufgabe. Der momentan verfolgte Ansatz greift hierbei die Idee der hierarchischen Abstraktion aus Abschnitt 7.1 auf. Ein Tester soll die Möglichkeit haben, zu abstrakten Informationen die zu Grunde liegenden detaillierten Ursprungsdaten begutachten zu können. Das Ergebnis jede Auswertungseigenschaft ist hierzu mit den Daten, aus denen die Eigenschaft berechnet wurde, verknüpft. Wegen der Datenvielfalt wird hierbei eine interaktive Form der Verfeinerung angestrebt, bei der die Daten von einem Web-Server on the fly generiert und im Web-Browser dargestellt werden.

Zusätzlich zur Dokumentation der Testergebnisse soll auch eine Dokumentation zu den modellierten Testfällen erstellt werden können. Da die Struktur einer Menge mit TPT modellierter Testfälle wegen der parallelen Hierarchien von Testlets und Szenarien nicht direkt in einem linearen Dokument dargestellt werden kann, ist hierfür zunächst eine geeignete Darstellungsform zu definieren.

Weitere Themen im Kontext von TPT

In der bisherigen Modellierungssprache von TPT gibt es zwei alternative Techniken, mit denen Testlets und die zugehörigen Szenarien modelliert werden

können: die direkte Definition und das Time Partitioning. Es ist zu untersuchen, inwiefern die Definition weiterer Techniken zur Modellierung von Testlets und Szenarien sinnvoll ist und in welchem Kontext ihr Einsatz ggf. von Vorteil ist. Durch die klar definierte Struktur von Testlets, die sich durch ihre Signatur und die Existenz einer Menge von zugehörigen Szenarien auszeichnen, ist diese Integration problemlos möglich. So könnte beispielsweise im Umfeld von Regelungssystemen die Unterstützung spezieller Techniken mit Sprachparadigmen der Regelungstechnik (Simulink, Differentialgleichungen) unter Umständen sinnvoll sein, damit die Modellierung von Testfällen noch weiter vereinfacht wird; für Beschreibung rein sequentieller Abläufe könnten mit Message Sequence Charts verwandte Sprachkonzepte zum Einsatz kommen.

Insbesondere wäre hierbei zu untersuchen, inwiefern die Idee der Variationen bei solchen Techniken sinnvoll einsetzbar ist und welche Elemente sich für diese Variationen eignen.

Bislang ist die Notwendigkeit solcher Erweiterungen noch nicht absehbar. Bei der Untersuchung muss insbesondere berücksichtigt werden, ob der Nutzen von Erweiterungen die steigende Sprachkomplexität rechtfertigt. Durch eine komplexere Sprache sinkt die Akzeptanz in der Praxis, sofern der notwendige Einarbeitungsaufwand zunimmt.

Funktionale Testfälle resultieren immer aus einer oder mehreren funktionalen Anforderungen, die durch den Testfall geprüft werden sollen. Bei der Testfallmodellierung mit TPT kommt eine weitere Beziehung hinzu: Auch die strukturellen Abläufe von Testfällen sowie die gebildeten Variationen werden letzten Endes eingeführt, um bestimmte Anforderungen zu prüfen. Diese Abhängigkeiten zwischen Anforderungen, Testfällen sowie den strukturellen Elementen von Testfällen (Zuständen, Transitionen, Szenarien, Transitionsdefinitionen etc.) sind bei komplexen Systemen schwer im Überblick zu behalten. So ist es unter anderem nur schwer nachvollziehbar, ob bzw. in welchem Umfang eine Anforderung durch eine Menge von Testfällen abgedeckt ist. Ändern sich Anforderungen im Laufe der Systementwicklung, müssen Testfälle ggf. überarbeitet, ergänzt oder entfernt werden. Für das Tracing dieser Abhängigkeiten ist es notwendig, die Beziehungen zwischen den Anforderungen einerseits und den Testfällen und strukturellen Elementen andererseits explizit zu machen. Die Identifikation der zu Grunde liegenden Relationen, ihrer Darstellung sowie die Verfolgung von Änderungen der Anforderungen sollen in zukünftigen Arbeiten untersucht werden.

Ein weiteres interessantes Themenfeld ist die generelle Untersuchung der Potentiale und Risiken, die reaktive Tests darstellen. Reaktive Tests definieren kein konstantes Verhalten, d.h., sie können in Abhängigkeit ihrer Umgebung während des Testablauf reagieren. Diese Mächtigkeit und Ausdrucksstärke der Testmodellierung wirft jedoch die Frage auf, welche Auswirkungen die

„Veränderlichkeit“ von Testfällen in Bezug auf die Testqualität hat. In diesem Zusammenhang ergeben sich vorrangig zwei Probleme: Erstens kann ein Testfall ggf. anders ablaufen als ursprünglich vom Tester intendiert, so dass die eigentlich gewünschte Situation nicht geprüft wird. Zweitens kann sich ein Testfall von einem Durchlauf zum nächsten anders verhalten, obwohl sich das Systemverhalten nur leicht ändert. Somit ist die Vergleichbarkeit von Testergebnissen in solchen Fällen problematisch. Ein möglicher Ansatz ist die Einführung von Monitoring-Instrumenten für den Test, mit deren Hilfe protokolliert wird, wie Testfälle während der Testdurchführung tatsächlich abgelaufen sind. Durch Angabe bestimmter Constraints ließen sich dadurch Schwachstellen aufdecken.

Insbesondere im Anwendungsfeld eingebetteter Systeme ist die Plattformunabhängigkeit von TPT von zentraler Bedeutung, wenn ein und dieselbe Funktion im Laufe der Entwicklung auf mehreren Testplattformen getestet werden soll (z.B. Entwicklungsrechner, Echtzeitrechner, Rechner mit Zielprozessor, vollständiger Targetcomputer und reale Einsatzumgebung inklusive der realen Umgebung). Die Wiederverwendbarkeit von Testfällen ist hierbei ein großer Fortschritt. Sie ist aber nur dann möglich, wenn auf den unterschiedlichen Plattformen auch die entsprechenden Schnittstellen zur Verfügung stehen, die in der Testfallmodellierung verwendet wurden und wenn diese Schnittstellen auch automatisierbar sind.

Wird beispielsweise ein eingebettetes Kfz-Steuerungssystem in der realen Einsatzumgebung im Fahrzeug getestet, ist der automatisierte Eingriff in Gas- oder Bremspedal entweder technisch sehr aufwändig oder gar nicht erwünscht. In diesen Fällen müssen die Testfälle zwangsläufig manuell durchgeführt werden. Um dennoch sicherzustellen, dass die richtigen Testfälle durchgeführt wurden, wäre es in diesen Fällen interessant, offline (also nach Aufzeichnung der Testrecords) zu ermitteln, welcher der modellierten Testfälle zu dem manuell durchgeführten Testfall passt. Entsprechende Analysealgorithmen für diese Erkennung sind zu erarbeiten.

TPT wurde ursprünglich für den Test des kontinuierlichen Verhaltens entwickelt. Durch die Verwendung von hybriden Automaten kann das Verfahren grundsätzlich aber auch für diskretes Verhalten – z.B. für Protokolltests – verwendet werden. Hierbei spielen die kontinuierlichen Anteile innerhalb von Zuständen keine Rolle. Stattdessen werden alle Verhaltensbeschreibungen innerhalb der Transitionen (Bedingungen und Aktionen) modelliert. Die Möglichkeit der Bildung von Variationen ist hiervon unbenommen. Für den Einsatz von TPT für derartige Verhaltenstests ist zu untersuchen, inwiefern die Sprachmittel von TPT für die Modellierung ausreichen bzw. welche Besonderheiten in diesem Umfeld berücksichtigt werden müssen.

Die im Kapitel 7 beschriebene Sprache zur Modellierung von Auswer-

tungseigenschaften von Testfällen stellt bislang nur ein Framework mit einer vergleichsweise einfachen Auswertungssprache zur Verfügung. Interessant ist deshalb die Untersuchung, welche Arten von Eigenschaften über kontinuierliches Verhalten sich mit welcher Art von Beschreibungstechnik formulieren lassen und wie diese Beschreibungstechniken in das TPT-Framework integriert werden können. Interessante und vielversprechende Ansätze sind hierbei MWatch [Lep01], temporale reguläre Ausdrücke sowie Mustererkennungsalgorithmen für den Signalvergleich.

Um mit Hilfe reaktiver Tests ein bestimmtes Extremverhalten des Systems zu provozieren, ist unter Umständen eine präzise Einstellung bestimmter Parameter notwendig, um das System in diesen Zustand zu versetzen. Die Identifikation solcher Parameter ist eine meist schwierige und aufwändige Aufgabe. Untersuchenswert ist deshalb, inwieweit sich evolutionäre Algorithmen [Weg01, WSJ+97] zur Optimierung der Parameter eignen. Voraussetzung ist hierfür, dass sich ein Maß für den Abstand zum gewünschten Verhalten definieren lässt, das optimiert werden soll. Offen ist hierbei, welche Faktoren eines modellierten Testfalls durch evolutionäre Strategien verändert werden müssen, um möglichst effektiv eine Lösung in Form von Testfällen zu ermitteln, die das Extremverhalten des Systems provozieren.

Ein übliches Verfahren, um den Gesamtumfang des Tests zu bewerten, ist die Überdeckungsmessung. Hierbei gibt es unterschiedliche Maße von der Anweisungsüberdeckung, bei der jede Anweisung mindestens einmal durchlaufen werden muss, bis zur Pfadüberdeckung, bei der sämtliche möglichen Pfade durch den Programmcode betrachtet werden müssen. Durch solche Maße ist zumindest sichergestellt, dass alle Codefragmente in einem gewissen Mindestumfang beim Test berücksichtigt wurden. Da kontinuierliches Verhalten in der Regel durch zyklische Systeme implementiert wird, ist zu untersuchen, ob der Einsatz derartiger Überdeckungsmaße auch für die Bewertung kontinuierlicher Testfälle sinnvoll ist bzw. welche spezifischen Maße für den Test des kontinuierlichen Verhaltens definiert werden können.

Abschließend lässt sich feststellen, dass Testtechniken für kontinuierliches Verhalten bereits heute ein vielversprechendes Potential haben und in der Zukunft verstärkt an Bedeutung gewinnen werden. Die Entwicklung des Time Partition Testing hat gezeigt, dass eine systematische Herangehensweise und die Automatisierung des Tests einander nicht widersprechen. Der wachsende Einsatz von TPT in der Serienentwicklung bei DaimlerChrysler bestätigt diese Behauptung.

Anhang A

Beweise

In den folgenden Beweisen werden kompakte Verweise verwendet. Eine referenzierte Stelle wird durch eine hochgestellte[1] Nummer in eckigen Klammern markiert, die an einer anderen Stelle durch dieselbe Nummer [1] im Fließtext referenziert werden kann. Die Beweise der Theoreme 3.1 und 3.2 sind vertauscht, da der Beweis des ersten Theorems als Spezialfall auf dem des zweiten aufbaut.

Beweis zu Theorem 3.2 auf Seite 36

Gegeben seien Belegungen $i_1, i_1' \in \vec{I_1}$, $i_2, i_2' \in \vec{I_2}$ sowie $o, o' \in \vec{O}$ mit $o =_{<0} o'$[2], $F(i_1, i_2, o)$ und $F(i_1', i_2', o')$ sowie $\delta > 0$ eine Konstante, die die charakteristische Bedingung der bedingten Verzögerung erfüllt.

Es wird per Induktion für $n = 0, 1, \ldots$ gezeigt, dass für jeden Zeitpunkt $t \in [\delta{\cdot}n, \delta{\cdot}(n{+}1))$ gilt[1]

$$i_1 =_{\leq t} i_1' \wedge i_2 =_{\leq t-\delta} i_2' \implies o =_{\leq t} o' \text{[3]}$$

- I.A. ($n = 0$, d.h. $0 \leq t < \delta$): trivial wegen der bedingten Verzögerung, da für $t < \delta$ nach [2] auch $o =_{<t-\delta} o'$ gilt.

[1] Anmerkung zum Beweis: Der Beweis zerlegt die nichtnegative Zeit in Intervalle der Länge δ. Diese Beweisidee könnte abgeschwächt werden, indem eine Folge von Intervallen mit einer Länge kleiner oder gleich δ betrachtet wird. Hierbei müsste jedoch zusätzlich zugesichert sein, dass jeder Zeitpunkt in einem dieser Intervalle enthalten ist, d.h., es muss eine Folge gewählt werden, die die Non-Zeno-Eigenschaft garantiert.

Die Induktion über $\mathbb{N}$ ist trotz der zu Grunde liegenden reellen Zeitachse sinnvoll und notwendig, da die Zeitkausalität (als wichtigste Grundlage für den hier geführten Beweis) die verzögerte Wirksamkeit der Ausgänge für ein nichtleeres Intervall der Länge größer oder gleich δ fordert: In Bezug auf die Kausalität ist eine stromverarbeitende Funktion deshalb nicht dicht, sondern diskret. Dies ist insbesondere auch eine notwendige Voraussetzung für die schrittweise Berechenbarkeit des approximierten Verhaltens, wie schon im Abschnitt 3.5 beschrieben wurde.

- I.S. ($n \rightsquigarrow n+1$): Angenommen [3] gelte für alle $t \in [\delta{\cdot}n, \delta{\cdot}(n{+}1))$. Sei nun $t' \in [\delta{\cdot}(n{+}1), \delta{\cdot}(n{+}2))$ ein beliebiger Zeitpunkt, und es gelte außerdem $i_1 =_{\leq t'} i_1'$ **[4]** sowie $i_2 =_{\leq t'-\delta} i_2'$ **[5]**. Damit gilt auch trivialerweise $i_1 =_{\leq t'-\delta} i_1'$ sowie $i_2 =_{\leq t-2\cdot\delta} i_2'$ und damit nach Ind.voraussetzung $o =_{\leq t'-\delta} o'$. Hiermit ist zusammen mit [4] und [5] die Voraussetzung für die Bedingung der bedingten Verzögerung gegeben, so dass schließlich $o =_{\leq t'} o'$ folgt. □

Beweis zu Theorem 3.1 auf Seite 34

Seien $i \in \vec{I}$ und $o, o' \in \vec{O}$ beliebige Belegungen mit $F(i,o)$, $F(i,o')$ sowie $o =_{<0} o'$. Zu zeigen ist, dass dann auch $o = o'$ gilt.

Für jede Komponente ist die Menge $I_2 = \emptyset$ trivialerweise verzögert wirksam. Somit gilt für jede Komponente als Spezialfall von Theorem 3.2 die Eigenschaft $i =_{\leq t} i \wedge o =_{<0} o' \implies o =_{\leq t} o'$.

Für *alle* $t \geq 0$ gilt trivialerweise die linke Seite dieser Implikation. Damit gilt auch $o =_{\leq t} o'$ für alle $t \geq 0$ und somit $o = o'$. □

Beweis zu Theorem 3.3 auf Seite 37

Sei $\delta > 0$ ein beliebiger Wert, der die verzögerte Wirksamkeit von O_2 bzgl. F_1 und die Zeitkausalität von F_2 erfüllt. Seien weiterhin $i, i' \in \overrightarrow{I_1 \cup I_2}$ sowie $o, o' \in \overrightarrow{O_1 \cup O_2}$ mit $o =_{<0} o'$, $(F_1 \parallel F_2)(i,o)$ **[6]** sowie $(F_1 \parallel F_2)(i',o')$ **[7]**. Sei nun $t \geq 0$ ein beliebiger Zeitpunkt, für den sowohl $i =_{\leq t} i'$ **[8]** als auch $o =_{\leq t-\delta} o'$ **[9]** gilt. Zu zeigen ist, dass dann auch $o =_{\leq t} o'$ gilt.

Wegen [6] bzw. [7] gilt nach Def. $F_1(i|_{I_1}, o|_{O_2}, o|_{O_1})$ sowie entsprechend analog $F_1(i'|_{I_1}, o'|_{O_2}, o'|_{O_1})$. Damit gilt wegen [8] und [9] und der verzögerten Wirksamkeit von O_2 bzgl. F_1 sofort $o|_{O_1} =_{\leq t} o'|_{O_1}$.

Für F_2 gilt analog $F_2(i|_{I_2}, o|_{O_1}, o|_{O_2})$ und $F_2(i'|_{I_2}, o'|_{O_1}, o'|_{O_2})$. Wegen der Zeitkausalität von F_2 gilt $o|_{O_2} =_{\leq t} o'|_{O_2}$ und damit schließlich $o =_{\leq t} o'$ □

Beweis zu Theorem 3.4 auf Seite 37

zu 1. Die Kommutativität der Parallelisierung ist nach Definition trivial. Für die Assoziativität lässt sich leicht zeigen, dass $((F_1 \parallel F_2) \parallel F_3)(i,o)$ gdw. $(F_1 \parallel (F_2 \parallel F_3))(i,o)$ ist:

Sei $((F_1 \parallel F_2) \parallel F_3)(i,o)$. Dann gilt $(F_1 \parallel F_2)(i|_{I_1 \cup I_2}, o|_{O_3}, o|_{O_1 \cup O_2})$ sowie $F_3(i|_{I_3}, o|_{O_1 \cup O_2}, o|_{O_3})$ und damit auch sowohl $F_1(i|_{I_1}, o|_{O_2 \cup O_3}, o|_{O_1})$ als auch $F_2(i|_{I_2}, o|_{O_1 \cup O_3}, o|_{O_2})$ und damit nach Definition schließlich $(F_1 \parallel (F_2 \parallel F_3))(i,o)$. Diese Schlussfolgerung ist umkehrbar. □

zu 2. Sei $[i,o]_{\cong}$ eine Initialbelegung für ein beliebiges Paar (i,o). Zu zeigen ist, dass es ein z gibt mit $z =_{<0} o$, für das $(F_1 \parallel F_2)(i,z)$ gilt. Für den Beweis

wird zunächst eine Folge $\{z_n\}_{n\in\mathbb{N}}$ konstruiert mit

$$\begin{aligned} &z_0 = o \\ &z_{n+1} =_{<0} z_n \quad \textbf{[10]} \\ &F_1(i|_{O_1}, z_n|_{O_2}, z_{n+1}|_{O_1}) \quad \textbf{[11]} \\ &F_2(i|_{O_2}, z_n|_{O_1}, z_{n+1}|_{O_2}) \quad \textbf{[12]} \end{aligned}$$

Nun wird per Induktion gezeigt, dass z_n immer eindeutig existiert.

- I.A. ($n = 0$): Die Eindeutigkeit ist trivial.
- I.S. ($n \rightsquigarrow n+1$): Sei z_n mit obigen Eigenschaften eindeutig. Da F_1 total ist, gibt es für die Initialbelegung $[i|_{O_1} \cup z_n|_{O_2}, z_n|_{O_1}]_{\cong}$ genau eine Belegung, die F_1 erfüllt. Wegen [10] liegt $(i|_{O_1} \cup z_n|_{O_2}, z_{n+1}|_{O_1})$ in dieser Restklasse und erfüllt gleichzeitig wegen [11] F_1. Somit ist $z_{n+1}|_{O_1}$ eindeutig definiert. Analoges gilt für $z_{n+1}|_{O_2}$ wegen [12], so dass schließlich folgt, dass auch z_{n+1} eindeutig definiert ist.

Nun wird wiederum induktiv für alle $n \in \mathbb{N}$ gezeigt, dass die Eigenschaft $z_n =_{<\delta n} z_{n+1}$ gilt.

- I.A. ($n = 0$) trivial nach [10]
- I.S. ($n \rightsquigarrow n+1$): Sei $z_n =_{<\delta n} z_{n+1}$. Nach [11] gilt offenbar sowohl $F_1(i|_{O_1}, z_n|_{O_2}, z_{n+1}|_{O_1})$ als auch $F_1(i|_{O_1}, z_{n+1}|_{O_2}, z_{n+2}|_{O_1})$. Es gilt nun $i|_{O_1} =_{<\delta(n+1)} i|_{O_1}$, $z_n|_{O_2} =_{<\delta n} z_{n+1}|_{O_2}$ sowie nach [10] $z_{n+1}|_{O_1} =_{<0} z_{n+2}|_{O_1}$.

  Somit gilt nach Theorem 3.2 wegen der verzögerten Wirksamkeit von O_2 bzgl. F_1 auch $z_{n+1}|_{O_1} =_{<\delta(n+1)} z_{n+2}|_{O_1}$.

  Damit folgt analog nach Theorem 3.2 wegen der verzögerten Wirksamkeit von $\emptyset$ bzgl. F_2 aus [12] auch $z_{n+1}|_{O_2} =_{<\delta(n+1)} z_{n+2}|_{O_2}$ und somit schließlich $z_{n+1} =_{<\delta(n+1)} z_{n+2}$.

Es gilt also $z_n =_{<\delta n} z_{n+1}$ und $z_{n+1} =_{<\delta(n+1)} z_{n+2}$ und damit auch $z_n =_{<\delta n} z_{n+2}$ usw., so dass man erneut induktiv folgern kann, dass $z_n =_{<\delta n} z_{n+i}$ für alle $i \in \mathbb{N}$ gilt. Das heißt, dass $\{z_n\}$ für $n \to \infty$ gegen eine eindeutig definierte Belegung z konvergiert mit $z =_{<\delta n} z_n$ (für alle $n \in \mathbb{N}$).

Für dieses z gilt $z =_{<\delta\cdot 0} z_0 = o$. Es bleibt zu zeigen, dass auch $(F_1 \parallel F_2)(i, z)$ für diesen Grenzwert erfüllt ist.

Da F_1 total ist, gibt es für die Initialbelegung $[i|_{o_1}, z|_{O_2}, z|_{O_1}]_{\cong}$ genau eine Belegung $\gamma =_{<0} z|_{O_1}$ mit $F_1(i|_{O_1}, z|_{O_2}, \gamma)$. Für *jedes* $n \in \mathbb{N}$ gilt außerdem [11]. Damit folgt für *alle* $n \in \mathbb{N}$ wegen $z|_{O_2} =_{<\delta n} z_n|_{O_2}$ und $\gamma =_{<0} z_{n+1}|_{O_1}$ nach Theorem 3.2 auch $\gamma =_{<\delta n} z_{n+1}|_{O_1} =_{<\delta n} z|_{O_1}$ und somit $\gamma =_{<\delta n} z|_{O_1}$

und damit $\gamma = z|_{O_1}$ (da n beliebig ist) sowie $F_1(i|_{o_1}, z|_{O_2}, z|_{O_1})$. Analog gilt für F_2, dass auch $F_2(i|_{O_2}, z|_{O_1}, z|_{O_2})$, so dass schließlich nach Definition auch $(F_1 \parallel F_2)(i, z)$ gilt. Damit liegt das so konstruierte z zusammen mit i in der Restklasse $[i, o]_{\cong}$ und erfüllt die Parallelisierung, womit die Totalität bewiesen ist. □

Beweis zu Theorem 3.5 auf Seite 39

Zeitkausalität. Sei $\delta > 0$ eine Konstante, die die verzögerte Wirksamkeit von $\{c\}$ bzgl. e_c erfüllt. Um die Zeitkausalität von G_c zu zeigen, werden beliebige Belegungen $i, i' \in \vec{I}$ sowie $o, o' \in \{\vec{c}\}$ sowie Zeitpunkte $t \geq 0$ betrachtet mit $o =_{<0} o'$**[13]**, $G_c(i, o)$**[14]**, $G_c(i', o')$**[15]**, $i =_{\leq t} i'$**[16]** und $o =_{\leq t-\delta} o'$**[17]**. Zu zeigen ist, dass $o =_{\leq t} o'$ ist.

Nach [14] und [15] gilt per Definition $o_c(t') = e_c(i, o)(t')$ sowie $o'_c(t') = e_c(i', o')(t')$ für alle $t' \geq 0$. Aufgrund der verzögerten Wirksamkeit von $\{c\}$ bzgl. e_c und wegen [16] und [17] gilt auch $e_c(i, o)(t') = e_c(i', o')(t')$ für alle $t' \leq t$, so dass auch $o_c(t') = o'_c(t')$ für alle $t' \in [0, t]$ gilt. Zusammen mit [13] folgt damit sofort $o =_{\leq t} o'$. □

Totalität. Sei (i, o) beliebig. Zu zeigen ist, dass $G_c[i, o]$ definiert ist. Sei $\{z_n\}_{n\in\mathbb{N}}$ eine Folge von Belegungen $z_n \in \{\vec{c}\}$, die wie folgt definiert ist:

$$\begin{aligned} z_0 &= o \\ z_{n+1} &=_{<0} z_n \text{**[18]**} \\ z_{n+1} &=_{\geq 0} (c \mapsto e_c(i, z_n))\text{**[19]**} \end{aligned}$$

Es wird nun induktiv gezeigt, dass $z_n =_{<\delta n} z_{n+1}$ für alle $n \in \mathbb{N}$ gilt.

- I.A. ($n = 0$) trivial nach [18]
- I.S. ($n \rightsquigarrow n + 1$): Sei $z_n =_{<\delta n} z_{n+1}$. Dann gilt $z_n =_{\leq t'-\delta} z_{n+1}$ sowie trivialerweise $i =_{\leq t'} i$ für alle $t' < \delta \cdot (n + 1)$. Damit gilt für all diese t' wegen der verzögerten Wirksamkeit von $\{c\}$ bzgl. e_c auch $e_c(i, z_n)(t') = e_c(i, z_{n+1})(t')$ und folglich nach [19] auch $z_{n+1} =_{[0,\delta\cdot(n+1))} z_{n+2}$. Wegen [18] folgt sofort $z_{n+1} =_{<\delta\cdot(n+1)} z_{n+2}$.

Dies bedeutet analog zum Beweis von Theorem 3.4 auf Seite 149, dass $\{z_n\}$ gegen eine Belegung z konvergiert, für die gilt, dass $z =_{<\delta n} z_n$.

Für jedes beliebige $t \geq 0$ gibt es ein $n \in \mathbb{N}$ mit $\delta n \geq t$, so dass $z =_{\leq t} z_n =_{\leq t} z_{n+1}$ und damit wegen der verzögerten Wirksamkeit von $\{c\}$ bzgl. e_c auch $e_c(i, z_n)(t) = e_c(i, z)(t)$ und damit nach [19] auch $z_{n+1_c}(t) = e_c(i, z)(t)$ sowie schließlich $z_c(t) = e_c(i, z)(t)$. Da diese Gleichheit für beliebige $t \geq 0$ gilt, folgt nach Definition sofort $G_c(i, z)$. Damit liegt das so konstruierte z

zusammen mit i in der Restklasse $[i,o]_{\cong}$ und erfüllt G_c, womit die Totalität bewiesen ist. □

Beweis zu Theorem 3.6 auf Seite 40

Sei $\delta > 0$ eine Konstante, die die Zeitkausalität von F_1 und F_2 sowie die bedingte Verzögerung von O bzgl. P erfüllt. Seien weiterhin $i, i' \in \vec{I}$, $o, o' \in \vec{O}$ und $t \geq 0$ mit $o =_{<0} o'$**[20]**, $(F_1 \overset{P}{\rightsquigarrow} F_2)(i,o)$, $(F_1 \overset{P}{\rightsquigarrow} F_2)(i',o')$, $i =_{\leq t} i'$**[21]** sowie $o =_{\leq t-\delta} o'$**[22]**. Zu zeigen ist, dass dann auch $o =_{\leq t} o'$ gilt.

Seien $\tau := P[i \cup o]$, $\tau' := P[i' \cup o']$. Wegen der verzögerten Wirksamkeit von O bzgl. P folgt damit nach [21] und [22], dass $P(i \cup o)(t') = P(i' \cup o')(t')$**[23]** für alle $t' \leq t$ gilt. Es werden nun die folgenden beiden Fälle unterschieden:

1. Fall ($\tau \leq t$): Nach Definition von τ gilt $P(i \cup o)(t') = 0$ für alle $t' \in [0, \tau)$ sowie $P(i \cup o)(\tau) = 1$. Nach [23] gilt insbesondere auch $P(i \cup o)(t') = P(i' \cup o')(t')$ für alle $t' \leq \tau$ und damit auch $P(i' \cup o')(t') = 0$ sowie $P(i' \cup o')(\tau) = 1$. Nach Definition von τ' gilt damit $\tau' = \tau$.

Nach Definition gilt $o =_{<\tau} \bar{o}$**[24]** und $o' =_{<\tau} \bar{o}'$**[25]** und damit nach [20] auch $\bar{o} =_{<0} \bar{o}'$, da $\tau \geq 0$ ist. Weiterhin gilt nach [21] und [22] $i =_{\leq \tau} i'$ und $o =_{\leq \tau - \delta} o'$ (da $\tau \leq t$ gilt). Zusammen mit [24] und [25] folgt damit $\bar{o} =_{<\tau-\delta} \bar{o}'$. Damit kann wegen der Zeitkausalität von F_1 gefolgert werden, dass $\bar{o} =_{\leq \tau} \bar{o}'$ gilt und damit letztendlich $o =_{<\tau} o'$**[26]** wiederum wegen [24] und [25].

Nach [21] und [22] gilt $(i \oplus \tau) =_{\leq t-\tau} (i' \oplus \tau)$ und $(o \oplus \tau) =_{\leq t-\tau-\delta} (o' \oplus \tau)$. Weiterhin gilt nach [26] $(o \oplus \tau) =_{<0} (o' \oplus \tau)$. Damit sind alle Voraussetzungen für die Zeitkausalität von F_2 gegeben, so dass schließlich auch $(o \oplus \tau) =_{\leq t-\tau} (o' \oplus \tau)$ und damit $o =_{\leq t} o'$ gilt.

2. Fall ($\tau > t$): In diesem Fall gilt nach Definition von τ für alle $t' \in [0, \tau)$ die Eigenschaft $P(i \cup o)(t') = 0$ und damit auch insbesondere für alle $t' \in [0, t]$. Wegen [23] gilt damit auch $P(i' \cup o')(t') = 0$ für alle $t' \in [0, t]$ und somit nach Definition $\tau' > t$.

Nach Definition gilt $o =_{<\tau} \bar{o}$**[27]** und $o' =_{<\tau'} \bar{o}'$**[28]** (und damit nach [20] auch $\bar{o} =_{<0} \bar{o}$, da $\tau \geq 0$ und $\tau' \geq 0$) sowie $F_1(i, \bar{o})$ und $F_1(i', \bar{o}')$. Nach [22] gilt außerdem $o =_{\leq t-\delta} o'$ und damit wegen $t - \delta < \tau$ und $t - \delta < \tau'$ auch $\bar{o} =_{<t-\delta} \bar{o}'$. Außerdem gilt [21] und damit wegen der Zeitkausalität von F_1 sofort $\bar{o} =_{<t} \bar{o}'$. Wegen $t < \tau$ gilt damit nach [27], [28] auch $o =_{<t} o'$. □

Beweis zu Theorem 3.7 auf Seite 40

Sei (i, o) ein beliebiges Paar von Belegungen. Zu zeigen ist, dass es ein $o' =_{<0} o$ gibt, so dass $(F_1 \overset{P}{\rightsquigarrow} F_2)(i, o')$ gilt.

Wegen der Totalität von F_1 gibt es ein eindeutiges $\bar{o} =_{<0} o$**[29]**, für das $F_1(i, \bar{o})$**[30]** gilt. Sei weiterhin $\tau' := P[i \cup \bar{o}]$. Wegen der Totalität von F_2 gibt es ein eindeutiges $\bar{o}' =_{<0} (\bar{o} \oplus \tau)$, für das $F_2(i \oplus \tau, \bar{o}')$ gilt.

Sei mit diesen Begriffen nun $o' := \bar{o}' \oplus -\tau$. Dann gelten für dieses o' folgende Eigenschaften:

- $F_2(i \oplus \tau, o' \oplus \tau)$**[31]**
- $(o' \oplus \tau) =_{<0} (\bar{o} \oplus \tau)$ und damit $o' =_{<\tau} \bar{o}$**[32]**
- Damit wegen der verzögerten Wirksamkeit von O bzgl. P auch $\tau = P[i \cup o']$**[33]**

Mit den Eigenschaften [30], [31], [32] und [33] sind alle Definitionseigenschaften für die Sequenzialisierung erfüllt, so dass schließlich $(F_1 \overset{P}{\rightsquigarrow} F_2)(i, o')$ gefolgert werden kann. Außerdem gilt wegen [29] und [32] $o' =_{<0} o$, so dass das so konstruierte o' die gewünschten Eigenschaften der Totalität erfüllt. □

Beweis zu Korollar 3.8 auf Seite 40

zu 1. Für alle (i, o) gilt $T[i \cup o] = \tau = 0$. Gilt also für ein solches Paar $(F_1 \overset{P}{\rightsquigarrow} F_2)(i, o)$, dann gilt per Definition trivialerweise auch $F_2(i \oplus 0, o \oplus 0)$ und damit $F_2(i, o)$. □

zu 2. Für alle (i, o) gilt $F[i \cup o] = \tau = \infty$. Gilt also für ein solches Paar $(F_1 \overset{P}{\rightsquigarrow} F_2)(i, o)$, dann gilt per Definition auch $F_1(i, \bar{o})$ mit $\bar{o} =_{<\infty} o$. D.h., es gilt $\bar{o} = o$ und damit $F_1(i, o)$. □

Beweis zu Theorem 3.9 auf Seite 42

Für den Beweis wird zunächst folgendes Lemma bewiesen:

Lemma: Seien $n \geq 1$ und eine Verkettung $S = F_1 \overset{P_1}{\rightsquigarrow} F_2 \overset{P_2}{\rightsquigarrow} \ldots \overset{P_{n-1}}{\rightsquigarrow} S'$ gegeben, wobei S' die endliche oder unendliche Restfolge von S ab der Komponente F_n ist[2]. Gegeben seien weiterhin Belegungen (i, o) mit $S(i, o)$ sowie $\langle \tau_0, \tau_1, \tau_2, \ldots \tau_{n-1}, \ldots \rangle$ die Schaltfolge von S bzgl. (i, o), wobei $\tau_{n-1} \neq \infty$ gelte. Dann gilt $S'(i \oplus \tau_{n-1}, o \oplus \tau_{n-1})$.

Beweis des Lemmas: Beweis durch Induktion über $n = 1, 2, \ldots$.

- I.A. $(n = 1)$. Es gilt per Definition $\tau_0 = 0$ und außerdem $S = S'$, so dass trivialerweise auch $S'(i \oplus 0, o \oplus 0)$ gilt.

[2] S' ist also selbst eine endliche oder unendliche Verkettung der Form $S' = F_n$, $S' = F_n \overset{P_n}{\rightsquigarrow} F_{n+1}$ oder $S' = F_n \overset{P_n}{\rightsquigarrow} F_{n+1} \overset{P_{n+1}}{\rightsquigarrow} \ldots$

- I.S. ($n \rightsquigarrow n+1$): Sei $S = F_1 \overset{P_1}{\rightsquigarrow} F_2 \overset{P_2}{\rightsquigarrow} \ldots \overset{P_{n-1}}{\rightsquigarrow} F_n \overset{P_n}{\rightsquigarrow} S''$ mit $\tau_n \neq \infty$ gegeben. Damit gilt nach Def. der Schaltfolge auch $\tau_{n-1} \neq \infty$, so dass nach Ind.vorauss. gilt $(F_n \overset{P_n}{\rightsquigarrow} S'')(i \oplus \tau_{n-1}, o \oplus \tau_{n-1})$. Sei $t := P_n[(i \oplus \tau_{n-1}) \cup (o \oplus \tau_{n-1})]$ und damit $t = P_n[(i \cup o) \oplus \tau_{n-1}]$. Nach Definition von τ_n gilt folglich $\tau_n = \tau_{n-1} + t$, so dass wegen $\tau_n \neq \infty$ auch $t \neq \infty$ gelten muss. Nach Definition der Sequenzialisierung gilt damit sofort, dass $S''((i \oplus \tau_{n-1}) \oplus t, (o \oplus \tau_{n-1}) \oplus t)$ und damit auch $S''(i \oplus (\tau_{n-1} + t), o \oplus (\tau_{n-1} + t))$ bzw. $S''(i \oplus \tau_n, o \oplus \tau_n)$. □

Mit Hilfe des obigen Lemmas ist der Beweis nun folgendermaßen zu führen: Gegeben seien ein beliebiges $n \geq 1$ und eine Verkettung S der Form $S = F_1 \overset{P_1}{\rightsquigarrow} \ldots \overset{P_{n-1}}{\rightsquigarrow} F_n$ oder der Form $S = F_1 \overset{P_1}{\rightsquigarrow} \ldots \overset{P_{n-1}}{\rightsquigarrow} F_n \overset{P_n}{\rightsquigarrow} S'$, wobei S' die endliche oder unendliche Restfolge von S ist. Seien weiterhin (i, o) Belegungen mit $S(i, o)$ sowie $\langle \tau_0, \tau_1, \ldots \tau_{n-1}, \tau_n, \ldots \rangle$ die Schaltfolge von S bzgl. (i, o) mit $\tau_{n-1} \neq \infty$. Zu zeigen ist nun, dass es ein $\bar{o} =_{<\tau_n} o$ gibt, mit $F_n(i \oplus \tau_{n-1}, \bar{o} \oplus \tau_{n-1})$.

Nach obigem Lemma gilt auf Grund der Voraussetzungen, dass $(F_n \overset{P_n}{\rightsquigarrow} S')(i \oplus \tau_{n-1}, o \oplus \tau_{n-1})$. Sei $t := P_n[(i \oplus \tau_{n-1}) \cup (o \oplus \tau_{n-1})]$ und damit $t = P_n[(i \cup o) \oplus \tau_{n-1}]$. Nach Definition von τ_n gilt folglich $\tau_n = \tau_{n-1} + t$. Nach Definition der Sequenzialisierung gibt es ein $o' =_{<t} o \oplus \tau_{n-1}$ mit $F_n(i \oplus \tau_{n-1}, o')$. Sei nun $\bar{o} := o' \oplus (-\tau_{n-1})$. Dann gilt sofort $\bar{o} \oplus \tau_{n-1} =_{<t} o \oplus \tau_{n-1}$ und damit $\bar{o} =_{<t+\tau_{n-1}} o$ bzw. $\bar{o} =_{<\tau_n} o$. Andererseits gilt $F_n(i \oplus \tau_{n-1}, \bar{o} \oplus \tau_{n-1})$, so dass dieses so konstruierte τ_n die gewünschten Eigenschaften erfüllt. □

Anhang B

Sprachbeschreibung

B.1 Ausdrücke

Jeder Ausdruck in TPT (vgl. 4.4.3) ist ein Wort zum Non-Terminal *expression*. Im Folgenden werden Syntax, Kontextbedingungen und die Bedeutung der einzelnen Sprachelemente kurz umrissen.

Jeder Ausdruck hat einen eindeutig definierten Typ, der entweder `boolean`, `int` oder `float` ist. Die Analyse der Kontexteigenschaften eines Ausdrucks und die Ermittlung des Typs erfolgt immer im Kontext des globalen *Deklarations-Environments*, in dem alle referenzierten Kanäle, Konstanten und Funktionen deklariert sein müssen. Das Deklarations-Environment enthält auch die jeweiligen Typen der Kanäle, Konstanten und Funktionen.

Für die Auswertung/Berechnung eines Ausdrucks ist ebenfalls ein entsprechendes *Auswertungs-Environment* notwendig, das folgende Informationen enthält:

- Wert des aktuellen Zeitpunkts '`t`', zu dem der Ausdruck ausgewertet werden soll,
- Wert der Berechnungsschrittweite δ (vgl. Abschn. 4.3.5),
- Wert des „`exited`"-Flags für die Information, ob ein Zustand verlassen wurde (Bedeutung des Flags wird weiter unten erläutert),
- zugehöriger Wert für jede im Ausdruck referenzierte Konstante c,
- zugehöriger Wert für jeden im Ausdruck referenzierten Kanal k, (Dies ist ein Strom, der zu jedem Zeitpunkt den Wert des Kanals enthält. Der Strom eines Kanals ist in der Regel partiell, d.h., es existieren Zeitpunkte, für die der Wert des Kanals nicht oder noch nicht bekannt ist.)
- Auswertungs-Semantik aller referenzierten Funktionen (siehe unten)

Grammatik der Ausdrücke

Die Definition der Ausdrucks-Sprache ist stark an die Syntax und Semantik von C angelehnt. Bei der Erläuterung der Sprache wird vorrangig auf die Sprachelemente eingegangen, die von C abweichen. Die C-verwandten Elemente werden nur kurz kommentiert.

In der folgenden Darstellung der Grammatik werden *Non-Terminale* kursiv dargestellt. '`Terminale`' werden nicht-proportional geschrieben und durch einfache Quotes umschlossen. Alle anderen Zeichen sind Zeichen der Grammatikregeln. (Dies sind die Zeichen $\leftarrow$, $|$, $*$, $+$, ε sowie die beiden runden Klammern.) ε ist das leere Wort. Zur Erläuterung wird ein Non-Terminal z.T. benannt, indem ein Bezeichner vorangestellt ist: Eine Expression *expression* mit dem Bezeichner *name* wird dann als *name : expression* notiert. Die Kontextbedingungen werden informell beschrieben. Ihnen wird ein „©"-Zeichen vorangestellt.

Es gibt die unten stehenden Klassen von Ausdrücken.

expression $\leftarrow$ *basicExpr*
| *castExpr*
| *conditionalExpr*
| *variableRefExpr*
| *functionRefExpr*
| *unaryExpr*
| *binaryExpr*

Einfache Ausdrücke

basicExpr $\leftarrow$ *floatNumber* | *intNumber*
| *constantIdent*
| '`t`' | '`exited`' | '@' | '`true`' | '`false`'
| '(' *expression* ')'

constantIdent $\leftarrow$ *ident*

- *floatNumber*: numerische Konstante vom Typ `float` ohne Vorzeichen (z.B. `12.34`, `0.01` oder 12.0).
- *intNumber*: numerische Konstante vom Typ `int` ohne Vorzeichen (z.B. `0` oder `12`).
- *constantIdent*: Name einer benannten Konstante, die wie eine Variable mit TPT global im Deklarations-Environment deklariert sein muss. Der Typ des Ausdrucks entspricht dem bei der Deklaration angegebenen Typ der Konstante.

  © Es muss eine Konstante mit dem angegeben Namen im Deklarations-Environment existieren.

- '`t`': repräsentiert den „aktuellen Zeitpunkt", der im Auswertungs-Environment festgelegt ist. Die Zeit wird entsprechend der Berechnungssemantik von TPT schrittweise weitergezählt. Durch die sich so verändernden Zeitwerte und die wiederholte Berechnung des Ausdrucks entstehen Zeit-Werte-Paare entsprechend der Abbildungsvorschrift $t \mapsto f(t)$.

  Der Ausdruck '`t`' hat den Typ `float`.

- '`exited`': Flag, das in Conditions einer Transition verwendet werden kann, um festzustellen, ob der Quellzustand der Transition soeben verlassen wurde, d.h., ob dieser Zustand „von innen" terminiert ist. Das Flag wird benötigt, wenn eine default-Transition definiert werden soll, die beim Verlassen eines Zustandes das Folgeverhalten definiert. Das Flag ist im Auswertungs-Environment festgelegt. Der Typ dieses Ausdrucks ist `boolean`.

  © Der Ausdruck '`exited`' darf nur innerhalb der Condition einer Transition verwendet werden.

- '@': Dieser Ausdruck entspricht der Schrittweite δ aus dem diskretisierten Berechnungsmodell von TPT (vgl. Abschn. 4.3.5). Normalerweise ist die Schrittweite der Diskretisierung für die Modellierung von Testfällen irrelevant und frei wählbar. Dadurch bewirkt eine feinere Diskretisierungsdichte lediglich eine genauere Berechnung mit geringeren Werte- und Zeitfehlern, ohne das grundsätzliche Verhalten zu verändern. In seltenen Fällen ist der Zugriff auf die verwendete Diskretisierungsdichte jedoch sinnvoll. Ein typisches Beispiel ist die Gleichung $\texttt{ch}(\texttt{t}) = \texttt{ch}(-@)$, bei der die Variable *ch* für alle $t \geq 0$ mit dem Wert der Variablen zum Zeitpunkt $-@$ übereinstimmt. Dieser Punkt ist bei Automaten der letzte diskret berechnete Zeitpunkt vor dem Betreten des aktuellen Zustandes, so dass dadurch die „konstante und stetige Fortsetzung des bisherigen Verlaufs" für die Variable *ch* modelliert wird.

  Der Ausdruck '@' hat den Typ `float`.

- '`true`': Diese Konstante entspricht dem Wahrheitswert *true* und hat den Typ `boolean`.

- '`false`': Diese Konstante entspricht dem Wahrheitswert *false* und hat den Typ `boolean`.

- '(' *expression* ')': Durch die Klammerung einer Expression kann in üblicher Weise die Bindungsstärke der Operatoren des Ausdrucks beeinflusst werden. Der Typ des Ausdrucks entspricht dem Typ der eingebetteten Expression.

Typ-Konvertierungen

castExpr ← '(' *typeName* ')' *expression*

typeName ← '`boolean`' | '`int`' | '`float`'

Im Gegensatz zu C hat TPT eine strenge Typisierung, die einen expliziten Typ-Cast erfordert. Die Syntax dieses Casts entspricht der Syntax von C. Es kann grundsätzlich jeder Typ in einen anderen Typ konvertiert werden. Die Konvertierungsregeln sind dabei wie folgt:

	nach `boolean`	nach `int`	nach `float`
von `boolean`		`true` $\rightsquigarrow 1$ `false` $\rightsquigarrow 0$	`true` $\rightsquigarrow 1.0$ `false` $\rightsquigarrow 0.0$
von `int`	$(i \neq 0)$		Fließkommawert von i
von `float`	$(f \neq 0.0)$	$\lvert f \rvert$ (ganzz. Anteil)	

Bedingter Ausdruck

conditionalExpr ← '(' *c : expression* ')' '?'
a$_1$ *: expression* ':'
a$_2$ *: expression*

Mit Hilfe eines bedingten Ausdrucks können wie in C alternative Definitionen in Abhängigkeit einer Bedingung formuliert werden. Die Auswertungssemantik ist lazy, d.h., es wird immer nur einer der beiden alternativen Ausdrücke a_1, a_2 berechnet. Der Typ des Gesamtausdrucks entspricht dem Typ der beiden alternativen Ausdrücke a_1/a_2.

© Der Ausdruck c muss den Typ `boolean` haben.

© Die Typen der beiden Ausdrücke a_1 und a_2 müssen identisch sein.

Referenz auf Variablen

variableRefExpr ← *variableIdent* '(' $tVal$ *: expression* ')'
| *variableIdent*

variableIdent ← *ident*

Innerhalb eines Ausdrucks kann auf den Wert von Variablen Bezug genommen werden. Hierfür gibt es zwei Formen. Bei der ersten Form wird neben dem Variablennamen ein Ausdruck $tVal$ angegeben, der den Zeitpunkt beschreibt, zu dem der Wert der Variablen referenziert werden soll. Existiert zu

diesem Zeitpunkt kein definierter Wert für die Variable oder ist dieser Wert zum aktuellen Zeitpunkt der Auswertung (noch) nicht bekannt, weil er „in der Zukunft“ liegt, erfolgt ein Laufzeitfehler. Der Typ des Ausdrucks entspricht dem Typ der Variablen.

Die zweite Form erfordert lediglich die Angabe eines Variablennamens (ohne weiteres Argument). Der Wert dieses Ausdrucks entspricht dem aktuellen Wert der Variablen zum Zeitpunkt der Auswertung des Ausdrucks. Diese Art der Referenzierung entspricht der klassischen Referenzierung von Variablen aus Programmiersprachen. Beim Zugriff auf eine nicht initialisierte Variable erfolgt ein Laufzeitfehler. Der Typ des Ausdrucks entspricht dem Typ der Variablen.

© Es muss eine Variable mit dem angegebenen Namen existieren.

© Der Ausdruck $tVal$ muss den Typ `float` haben.

Funktionsanwendung

functionRefExpr	←	*functionIdent* ‘(’ *parameters* ‘)’
functionIdent	←	*ident*
parameters	←	ε | *paras*
paras	←	*parameter*
	|	*parameter* ‘,’ *paras*
parameter	←	*expression* | *quotedText*

Funktionen können mit TPT dazu verwendet werden, komplexere Berechnungen und Analysen durchzuführen, die mit der relativ einfachen Sprache von TPT nicht realisierbar sind. Beispielsweise könnte es sinnvoll sein, den Verlauf einer Variablen (also den Strom) aus einer Datei auszulesen. Auch mathematische Funktionen wie $\sin(x)$ u.a. sind in TPT nicht vordefiniert, können aber über das Funktions-Framework eingebunden werden.

Zu diesem Zweck können Funktionen dynamisch in die Analyse und Auswertung von Ausdrücken eingebunden werden. Jede Funktion wird durch eine Komponente repräsentiert, die den Namen der Funktion, die Bedingungen der Kontextanalyse und die Auswertungssemantik der Funktion festlegt (vgl. Abschn. 6.3).

Parameter einer Funktion können beliebige Expressions sein. Zusätzlich sind als Parameter auch Strings zulässig. Diese können beispielsweise für zusätzliche Optionen, Dateinamen o.ä. verwendet werden.

© Es muss eine Funktion mit dem angegebenen Namen im Deklarations-Environment existieren.

© Die Kontextanalyse der Funktion muss die angegebene Parameterliste akzeptieren. (Hierbei werden implizit auch die Parameterexpressions selbst auf ihre Kontexteigenschaften hin geprüft.)

Einstellige Operatoren

unaryExpr	←	(‘+’ | ‘−’) *a : expression*
	|	‘!’ *b : expression*

Die arithmetischen Vorzeichen können für jeden Ausdruck a vom Typ `int` oder `float` verwendet werden. Die logische Negation ist für alle boolschen Ausdrücke b zulässig.

© Der Typ von a muss entweder `int` oder `float` sein.

© Der Typ von b muss `boolean` sein.

Zweistellige Operatoren

binaryExpr	←	*expression binOp expression*
binOp	←	‘+’ | ‘−’ | ‘∗’ | ‘/’
	|	‘&&’ | ‘||’
	|	‘<’ | ‘>’ | ‘<=’ | ‘>=’ | ‘==’ | ‘! =’

Zweistellige Operatoren werden grundsätzlich infix notiert. Die Semantik der Operatoren entspricht der Semantik in C. Mit Ausnahme der beiden Operatoren ‘&&’ und ‘||’ haben alle eine eager-Semantik. Diese beiden logischen Operatoren werden von links nach rechts ausgewertet.

Der Typ des Ausdrucks entspricht bei ‘+’, ‘−’, ‘∗’ und ‘/’ dem Typ der beiden Argument-Expressions. Der Typ der verbleibenden Ausdrucksformen ist immer `boolean`.

© Bei Verwendung der Operatoren ‘+’, ‘−’, ‘∗’, ‘/’, ‘<’, ‘>’, ‘<=’ oder ‘>=’ muss der Typ für beide Teilausdrücke entweder `int` oder für beide `float` sein.

© Bei Verwendung den Operatoren ‘&&’ und ‘||’ muss der Typ für beide Teilausdrücke `boolean` sein. Bei den Operatoren ‘==’ und ‘! =’ muss der Typ für beide Teilausdrücke gleich sein.

Alle binären Operatoren sind links-assoziativ. Die Bindungsstärke aller Operatoren ist in der Tabelle B.1 zusammengefasst.

Op.	Bindung	Bemerkung
!	7	log. Negation
$+, -$	6	arithm. Vorzeichen/Negation
$*, /$	5	Multiplikation, Division
$+, -$	4	Addition, Subtraktion
$\&\&, \|\|$	3	log. Konjunktion/Disjunktion
$<, >, <=, >=$	2	Ordnungsrelationen
$==, !=$	1	Gleichheit/Ungleichheit
$(\_)?\_ : \_$	0	bedingter Ausdruck

Abbildung B.1: Bindungsstärken der Operatoren

Lexikalik

Die Lexikalik der Ausdruckssprache von TPT ist trivial. Die beiden Zeichen **nl** und **lf** entsprechen dem Newline- und Linefeed-Zeichen.

ident	:=	*letter* (*letter* | *digit*)*
letter	:=	'a' | .. | 'z' | 'A' | .. | 'Z' | '_'
digit	:=	'0' | .. | '9'
intNumber	:=	(*digit*)+
floatNumber	:=	(*digit*)+ '.' (*digit*)+
quotedText	:=	'"' (*textContent*)* '"'
textContent	:=	'\"' | ~ ('\' | '"' | **'nl'** | **'lf'**)

B.2 Testfalldefinition

Jede XML-Szenariobeschreibung für TPT (vgl. Abschn. 6.2) besteht aus einem Tag *tscen*, das die Daten zu dem Szenario enthält. Das Attribut *origin* enthält eine URL zur Ursprungsdatei des generierten Testfalls. Weiterhin gibt es ein Header-Tag, das die Schnittstelle des Szenarios beschreibt sowie ein Body-Tag, das den Inhalt definiert.

```
<!ELEMENT   tscen (header, body) >
<!ATTLIST   tscen
            origin CDATA #IMPLIED >
```

Headerdefinition

Im Header der Szenariobeschreibung sind alle Eingänge, Ausgänge und lokalen Kanäle aufgelistet. Diese Mengen sind disjunkt, d.h., die Namen der Kanäle sind eindeutige XML-IDs. Zu jedem Kanal gehört ein Name, der Typ (float, boolean oder int) sowie die Information, ob es sich um einen flüchtigen Kanal handelt.

Weiterhin sind im Header der Szenariobeschreibung alle verwendeten Konstanten aufgeführt, die in Ausdrücken des Szenarios verwendet werden. Konstanten haben einen eindeutigen Namen, einen Typ (float, boolean oder int) sowie einen definierten Wert.

`<!ELEMENT`	*header* (*inputs*, *outputs*, *locals*, *consts*) `>`
`<!ELEMENT`	*inputs* (*channel**) `>`
`<!ELEMENT`	*outputs* (*channel**) `>`
`<!ELEMENT`	*locals* (*channel**) `>`
`<!ELEMENT`	*channel* 'EMPTY' `>`
`<!ATTLIST`	*channel name* `ID #REQUIRED` *volatile* (`true` | `false`) `#REQUIRED` *%type;* `>`
`<!ELEMENT`	*consts* (*const**) `>`
`<!ELEMENT`	*const EMPTY* `>`
`<!ATTLIST`	*const name* `ID #REQUIRED` *%type;* *value* `CDATA #REQUIRED >`
`<!ENTITY`	*type* "*type* (`float` | `boolean` | `int`)" `#REQUIRED >`

Verhaltensdefinition eines Szenarios

Die Verhaltensdefinition besteht aus einer Hierarchie von *Blöcken*, die die (Sub-)Szenarien des gesamten Testszenarios beschreiben. Es wird zwischen Gleichungsblöcken (*block:eqn*), Parallelblöcken (*block:parallel*) und Diagrammblöcken (*block:diagram*) unterschieden.

Ein Gleichungsblock enthält zu einem gegebenen Kanal (siehe Attribut *defines*) einen Definitionsausdruck (*%expr*-Subtag; siehe unten). Optional kann auch der Quellstring des Definitionsausdrucks mit angegeben werden. Diese Information wird jedoch zur Auswertung nicht benötigt und dient nur zur leichteren Lesbarkeit.

Ein Parallelblock setzt sich aus einer Sequenz von Subblöcken zusammen, die zu jedem Zeitpunkt von oben nach unten ausgewertet werden.

Ein Diagrammblock definiert die zu dem TP-Diagramm gehörenden Zustände und Transitionen, wobei bei den Transitionen zwischen initialen, finalen und „normalen" Transitionen unterschieden wird (siehe unten).

`<!ELEMENT`	*body* (*%block;*) `>`
`<!ENTITY`	*block "block:eqn | block:parallel | block:diagram"* `>`
`<!ELEMENT`	*block:eqn* (*%expr;*) `>`
`<!ATTLIST`	*block:eqn defines* `IDREF #REQUIRED` *source* `CDATA #IMPLIED >`
`<!ELEMENT`	*block:parallel* ((*%block;*)*) `>`
`<!ELEMENT`	*block:diagram* (*state**, (*trans:init* | *trans* | *trans:final*)*) `>`

Jeder Zustand eines TP-Diagrammblocks hat eine eindeutige Id, sowie ggf. eine assoziierte „Zustandsvariable" (Attribut *statevar*). Diese Zustandsvariable entspricht einem lokalen Kanal, der für diesen Zustand protokollieren soll,

in welchen Zeitintervallen der Zustand aktiv war. Diese Information wird für die Testauswertung benötigt (vgl. Kapitel 7).

Transitionen haben eine Transitionsbedingung (Subtag *cond*) sowie eine Aktiondefinition. Die Bedingung einer Transition wird durch einen Ausdruck (*%expr*) beschrieben; die Aktionen bestehen aus einer Folge von Gleichungen (*block:eqn*), die die Aktionen zum Transitionszeitpunkt beschreiben. Diese Gleichungen werden jedoch nur genau zum Zeitpunkt des Schaltens der Transition berechnet. Sowohl die Bedingung als auch die Aktionen sind optional. Eine Transition ohne Bedingung ist äquivalent zu der Bedingung `exited` (vgl. Abschn. 4.5.1).

Für initiale Transitionen (*trans:init*) wird nur die Id des Zielzustandes der Transition angegeben. Die Quelle dieser Transition ist der initiale Knoten, der für die Ausführungssemantik irrelevant ist und deshalb in der XML-Repräsentation nicht enthalten ist. Analog wird für finale Transitionen (*trans:final*) nur die Id des Quellzustandes angegeben. Der finale Knoten ist nicht Bestandteil der XML-Struktur. „Normale“ Transitionen erfordern die Angabe der Id des Quell- und des Zielknotens.

`<!ELEMENT`	*state* (*%block;*) `>`
`<!ATTLIST`	*state id* `ID #REQUIRED` *statevar* `IDREF #IMPLIED >`
`<!ELEMENT`	*trans:init* (*cond*?, *actions*?) `>`
`<!ATTLIST`	*trans:init to* `IDREF #REQUIRED >`
`<!ELEMENT`	*trans:final* (*cond*?, *actions*?) `>`
`<!ATTLIST`	*trans:final from* `IDREF #REQUIRED >`
`<!ELEMENT`	*trans* (*cond*?, *actions*?) `>`
`<!ATTLIST`	*trans from* `IDREF #REQUIRED` *to* `IDREF #REQUIRED >`
`<!ELEMENT`	*cond* (*%expr;*) `>`
`<!ATTLIST`	*cond source* `CDATA #IMPLIED >`
`<!ELEMENT`	*actions* ((*%block;*)*) `>`

Expressions

Ausdrücke werden als Syntaxbaum in Form von XML-Tags repräsentiert. Die einzelnen Ausdrucksformen entsprechen denen, wie sie in der Sprachbeschreibung im Abschnitt B.1 bereits definiert wurden.

<!ENTITY	*expr* "*expr:neg* | *expr:not* | *expr:ifthenelse* | *expr:appl* | *expr:libfun* | *expr:t* | *expr:exited* | *expr:density* | *expr:true* | *expr:false* | *expr:num* | *expr:const* | *expr:current* | *expr:cast* | *expr:plus* | *expr:minus* | *expr:times* | *expr:div* | *expr:land* | *expr:lor* | *expr:greaterthan* | *expr:lessthan* | *expr:greaterorequal* | *expr:lessorequal* | *expr:equals* | *expr:notequals*" >
<!ELEMENT	*expr:neg* (%expr;) >
<!ELEMENT	*expr:not* (%expr;) >
<!ELEMENT	*expr:ifthenelse* ((%expr;), (%expr;), (%expr;)) >
<!ELEMENT	*expr:appl* (%expr;) >
<!ATTLIST	*expr:appl channel* IDREF #REQUIRED >
<!ELEMENT	*expr:libfun* ((%expr; | *text*)*) >
<!ATTLIST	*expr:libfun name* CDATA #REQUIRED >
<!ELEMENT	*expr:t* EMPTY >
<!ELEMENT	*expr:exited* EMPTY >
<!ELEMENT	*expr:density* EMPTY >
<!ELEMENT	*expr:true* EMPTY >
<!ELEMENT	*expr:false* EMPTY >
<!ELEMENT	*expr:num* EMPTY >
<!ATTLIST	*expr:num value* CDATA #REQUIRED >
<!ELEMENT	*expr:const* EMPTY >
<!ATTLIST	*expr:const const* IDREF #REQUIRED >
<!ELEMENT	*expr:current* EMPTY >
<!ATTLIST	*expr:current channel* IDREF #REQUIRED >
<!ELEMENT	*expr:cast* (%expr;) >
<!ATTLIST	*expr:cast to* (float | boolean | int) #REQUIRED >
<!ELEMENT	*expr:plus* ((%expr;), (%expr;)) >
<!ELEMENT	*expr:minus* ((%expr;), (%expr;)) >
<!ELEMENT	*expr:times* ((%expr;), (%expr;)) >
<!ELEMENT	*expr:div* ((%expr;), (%expr;)) >
<!ELEMENT	*expr:land* ((%expr;), (%expr;)) >
<!ELEMENT	*expr:lor* ((%expr;), (%expr;)) >
<!ELEMENT	*expr:greaterthan* ((%expr;), (%expr;)) >
<!ELEMENT	*expr:lessthan* ((%expr;), (%expr;)) >
<!ELEMENT	*expr:greaterorequal* ((%expr;), (%expr;)) >
<!ELEMENT	*expr:lessorequal* ((%expr;), (%expr;)) >
<!ELEMENT	*expr:equals* ((%expr;), (%expr;)) >
<!ELEMENT	*expr:notequals* ((%expr;), (%expr;)) >
<!ELEMENT	*text* EMPTY >
<!ATTLIST	*text value* CDATA #REQUIRED >

Index

Aktion, 50
Assessment, 140

Belegung, 35
    diskrete, 53
    Kompatibilität, 35
Black-Box-Test, 102

Definitionsvarianten, 103
Direkte Definition, 69
Direkte Definition, 67

Funktionstestmethode, 102

Gleichung, 42

Hybride Systeme, 33, 47

Initialbelegung, 38
Interleaving-Punkt, 31

Kanal, 35, 64, 140
    flüchtig, 91
    nichtflüchtig, 91
Klassifikationsbaum-Methode, 119
Kombinationsregeln, 127
Komponente, 36
    diskrete, 53
    partielle, 38
    totale, 38

Logische Anhängigkeiten, 127

Message, 34
Multiplizität, 31

Parallelisierung, 40

Schaltfolge, 45
Sequenzialisierung, 43
Stetige Anknüpfung, 45, 75
Strom, 34
    diskreter, 53
Stromverarb. Funktionen, 33
Strukturtestmethode, 102
Szenario, 63
    Signatur, 66
Szenariogruppen, 111

Temporale Logik, 32
Temporales Prädikat, 43
Termination, 50
Testengine, 26, 64, 131
    interpretierende, 132
    vorverarbeitende, 132
Testlet, 106, 111–113
Testobjekt, 13
Testorakel, 15
Testrecord, 131, 135
Testreferenz, *siehe* Testorakel
Time Partitioning, 67, 84
Time Transition Systems, 33
Timed Automata, 32
Transitionsbedingung, 43
Transitionspunkt, 44
Transitionsspezifikation, 114

Unabhängigkeit, 39
Unmittelbare Wirksamkeit, 39

Variable, 140

Variation, 107
    an Transitionen, 107
    an Zuständen, 107
    Pfadvariation, 107
Verhalten
    kontinuierliches, 23
    zeitdiskretes, 23
Verkettung, 45
Verzögerte Wirksamkeit, 38, 42

White-Box-Test, 15, 102

Zeit
    lineare, 30
    reale, 30
    universelle, 31
Zeitkausalität, 36

Literaturverzeichnis

[AAH83] Rita L. Atkinson, Richard C. Atkinson, and Ernest R. Hilgard. *Introduction to Psychology.* Harcourt Brace Jovanovich, Inc., New York, 1983.

[ACH+95] R. Alur, C. Coucoubetis, N. Halbwachs, T.A. Henzinger, P.-H. Ho, et al. The Algorithmic Analysis of Hybrid Systems. *Theoretical Computer Science*, 138:3–34, 1995.

[AH89] Rajeev Alur and Thomas A. Henzinger. A really temporal logic. In *Proceedings of the 30th IEEE Symposium on Foundations of Computer Science*, pages 164–169. IEEE Computer Society Press, 1989.

[AH92] Rajeev Alur and Thomas A. Henzinger. Logics and Models of Real Time: A Survey. In *Real Time: Theory in Practice*, volume 600 of *LNCS*, pages 74–106. Springer Verlag, 1992.

[AH97] Rajeev Alur and Thomas A. Henzinger. Modularity of Timed and Hybrid Systems. In *Eighth Conference on Concurrency Theory*, volume 1243 of *LNCS*, pages 74–88. Springer Verlag, 1997.

[AL92] M. Abadi and L. Lamport. An old-fashioned recipe for Real Time. In *Real Time: Theory in Practice*, volume 600 of *LNCS*. Springer Verlag, 1992.

[Alu99] Rajeev Alur. Timed Automata. In *11th International Conference On Computer-Aided Verification*, volume 1633 of *LNCS*, pages 8–22. Springer-Verlag, 1999.

[AMC97] E. Asarin, O. Maler, and P. Caspi. A Kleene Theorem for Timed Automata. In *Proceedings of the 12th IEEE Symposium on Logic in CS*, pages 160–171, 1997.

[Bei90] B. Beizer. *Software Testing Tequniques.* Van Nostrand Reinhold, New York, 2nd. edition, 1990.

[Bei95] B. Beizer. *Black-Box Testing: Tequniques for Functional Testing of Software and Systems.* John Wiley & Sons Inc., New York, 1995.

[BKP89] S. Brander, A. Kmpa, and Ulf Peltzer. *Denken und Problemlösen: Einführung in die kognitive Psychologie.* Westdeutscher Verlag, 1989.

[Blu92] Bruce Blum. *Software Engineering: A Holistic View.* Oxford University Press, 1992. ISBN 0-19-507159-X.

[Bou85] Luc Bougé. A Contribution to the Theory of Program Testing. *Theoretical Computer Science*, 37, 1985.

[Bri89] Ed Brinksma. A Theory for the Derivation of Tests. In S. Aggarwal and K. Sabnani, editors, *Protocol Specification, Testing, and Verification*, pages 235–247. Elsevier Science Publishers, 1989.

[Bro97] Manfred Broy. Refinement of Time. In *Transformation-Based Reactive System Development*, volume 1231 of *LNCS*, pages 44–63. ARTS97, Springer-Verlag, 1997.

[BS+95] I.N. Bronstein, K.A. Semendjaev, et al. *Taschenbuch der Mathematik*, chapter 19, page 786. Verlag Harri Deutsch, 1995.

[CD96a] H.D. Chu and J. Dobson. A Statistics-based Framework for Automated Software Testing. In *9th International Software Quality Week (QW'96)*, San Francisco, CA, 1996.

[CD+96b] D. Cohen, S. Dalal, et al. The Combinatorial Design Approach to Automatic Test Case Generation. *IEEE Software*, pages 83–87, September 1996.

[CY94] T.Y. Chen and Y.T. Yu. On the Relationship Between Partition and Random Testing. *IEEE Transactions on Software Engineering*, 20(12):977–980, 1994.

[CY96] T.Y. Chen and Y.T. Yu. On the Expected Number of Failures Detected by Subdomain Testing and Random Testing. *IEEE Transactions on Software Engineering*, 22(2):109–119, 1996.

[Dij72] E.W. Dijkstra. Notes on structured programming. In O.-J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, editors, *Structured Programming*, pages 1–81. Academic Press, New York, 1972.

[DKRT83] D. Dörner, H.W. Kreuzig, F. Reither, and T.Stäudel, editors. *Vom Umgang mit Komplexität.* Huber, Bern, Switzerland, 1983.

[DN84] J.W. Duran and S.C. Ntafos. An Evaluation of Random Testing. *IEEE Transactions on Software Engineering*, 10(4):438–444, 1984.

[Dör81] D. Dörner. Über die Schwierigkeiten menschlichen Umgangs mit Komplexität. *Psychologlische Rundschau*, XXXI(3):163–179, 1981.

[Dör89] D. Dörner. *Die Logik des Mißlingens.* Rowohlt, Reinbek, Germany, 1989.

[DR78] D. Dörner and F. Reither. Über das Problemlösen in sehr komplexen Realitätsbereichen. *Zeitschrift für experimentelle und angewandte Psychologie*, 25:527–551, 1978.

[DS96] Dietrich Dörner and Herbert Selg, editors. *Psychologie: Eine Einführung in ihre Grundlagen und Anwendungsfelder.* Verlag W. Kohlhammer GmbH, 2nd edition, 1996.

[Dun59] C.P. Duncan. Recent Research on Human Problem Solving. *Psychological Bulletin*, 56(6):397–429, 1959.

[Fer93] R.C. Ferguson. *Test Data Generation for Sequential and Distributed Programs.* PhD thesis, Wayne State University, 1993.

[FK96] R.C. Ferguson and B. Korel. The Chaining Approach for Software Test Data Generation. *ACM Transactions on Software Engineering and Methodology*, 5(1):63–86, 1996.

[Fri01] Jeffrey Friedl. *Reguläre Ausdücke.* O'Reilly, 1st edition, 2001.

[FW93] P.G. Frankl and E.J. Weyuker. A Formal Analysis of the Fault-Detecting Ability of Testing Methods. *IEEE Transactions on Software Engineering*, 19(3):202–213, 1993.

[Gel78] M. Geller. Test Data as an Aid in Proving Program Correctness. *Commun. Ass. Comput. Mach.*, 21, May 1978.

[GG75] John B. Goodenough and Susan L. Gerhart. Toward a Theory of Test Data Selection. *IEEE Transactions on Software Engineering*, 1(2):156–173, 1975.

[GG93] M. Grochtmann and K. Grimm. Classification Trees for Partition Testing. *Software Testing, Verification & Reliability*, 3(2):63–82, 1993.

[GH99] J. Grabowski and D. Hogrefe. An Introduction to TTCN-3. In *Testing of Communicating Systems – Methods and Applications.* Kluwer Academic Publishers, 1999.

[Gou83] John S. Gourlay. A Mathematical Framework for the Investigation of Testing. *IEEE Transactions on Software Engineering*, 9(6), 1983.

[Gri95] Dr. Klaus Grimm. *Systematisches Testen von Software – Eine neue Methode und eine effektive Teststrategie*. GMD-Bericht 251. R. Oldenburg Verlag, München/Wien, 1995.

[Gro94] Matthias Grochtmann. Test Case Design Using Classification Trees. In *Proceedings of STAR'94*, pages 93–117, Washington, DC, 1994.

[GSB98] Radu Grosu, Thomas Stauner, and Manfred Broy. A Modular Visual Model for Hybrid Systems. In *Formal Techniques in Real Time and Fault Tolerant Systems*. Springer-Verlag, 1998.

[GW95] M. Grochtmann and J. Wegener. Test Case Design Using Classification-Trees and the Classification-Tree Editor CTE. In *8. International Software Quality Week*, San Francisco, USA, 1995.

[Hal86] M. Hall. *Combinatorial Theory*. interscience series. John Wiley, 1986.

[Har87] D. Harel. A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[Hen96] Thomas A. Henzinger. The Theory of Hybrid Automata. In *11th IEEE Symposium on Logics of Computer Science*, pages 287–293, 1996.

[HMP92] Thomas A. Henzinger, Zohar Manna, and Amir Pnueli. Timed Transition Systems. In *Real Time: Theory in Practice*, volume 600 of *LNCS*, pages 226–252. Springer Verlag, 1992.

[HN96] D. Harel and A. Naamad. The Statemate Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.

[Hob95] Hermann Hobmair, editor. *Psychologie*. Stam Verlag, Köln, Germany, 1995.

[Hop81] Grace Murray Hopper. The first bug. *Annals of History of Computing*, 3:285–286, 1981.

[How78] W.E. Howden. Algebraic Program Testing. *Acta Informatica*, 10, January 1978.

[HRS97] T. Henzinger, J. Raskin, and P. Schobbens. The regular real-time languages. In *ICALP98: Automata, Languages, and Programming*, volume 1443 of *LNCS*, pages 580–593. Springer-Verlag, 1997.

[HT90] Dick Hamlet and Ross Taylor. Partition Testing Does Not Inspire Confidence. *IEEE Transactions on Software Engineering*, 16(12):1402–1411, December 1990.

[Inc87] D.C. Ince. The Automatic Generation of Test Data. *The Computer Journal*, 30(1):63–69, 1987.

[IPL99] Testing: Not Rocket Science, But When Will We Ever Learn? *Testing Times*, (14), 1999.

[Jos92] Matthew Josephson. *Edison: A Biography*, page 198. John Wiley & Sons, 1992.

[Kit95] E. Kit. *Software Testing in the Real World: improving the process.* Addison-Wesley Publishing Company, Workingham, UK, 1995.

[Knu86] Donald E. Knuth. *TEX: The Program.* Addison-Wesley, Menlo Park, California, 2nd edition, 1986. reprinted with corrections in 1988.

[Leh00] Eckard Lehmann. Time Partition Testing: A Method for Testing Dynamic Functional Behavior. In *Proceedings of The European Software Test Congress 2000*, London, 2000.

[Leh01] Eckard Lehmann. Time Partition Testing: Systematischer Test eingebetteter Systeme. *Softwaretechnik-Trends*, 21(3), November 2001. Bericht zur GI-Fachtagung TAV 17.

[Lep01] Markus Lepper. MWatch – Extended Tutorial. Interner Report der TU Berlin im Auftrag der DaimlerChrysler AG, Dezember 2001. http://www.cs.tu-berlin.de/∼lepper.

[Lig94] P. Liggesmeyer. Selecting Adequate Test Techniques in the Real World. In *11th International Conference and Exposition on Testing Computer Software*, pages 61–69, Washington DC, US, 1994.

[Lig96] P. Liggesmeyer. Selecting Test Methods, Techniques, Metrics and Tools Using Systematic Decision Support. In *9th International Software Quality Week (QW'96)*, San Francisco, CA, 1996.

[LW00] Eckard Lehmann and Joachim Wegener. Test Case Design by Means of the CTE XL. In *8th European International Conference on Software Testing, Analysis & Review (EuroSTAR 2000)*, Copenhagen, Denmark, 2000.

[Mag93] Stephen A. Maguire. *Writing Solid Code.* Mircosoft Press, Redmond, 1993.

[May79] Richard E. Mayer. *Denken und Problemlösen.* Springer-Verlag, Berlin, Heidelberg, New York, 1979.

[MB+33] James A.H. Murray, Henry Brandley, et al., editors. *The Oxford English Dictionary*, volume 1. Carendon Press, 1933.

[McK80] Robert H. McKim. *Thinking Visually – A Strategy Manual for Problem Solving.* Wadsworth, Inc., Belmont, California, 1980.

[MMP92] Oded Maler, Zohar Manna, and Amir Pnueli. From Timed to Hybrid Systems. In *Real Time: Theory in Practice*, volume 600 of *LNCS*, pages 447–484. Springer Verlag, 1992.

[Mor90] Larry J. Morell. A Theory of Fault-Based Testing. *IEEE Transactions on Software Engineering*, 16(8):844–857, 1990.

[MP91] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer Verlag, New York, 1991.

[MS97] Olaf Müller and Peter Scholz. Functional Specification of Real-Time and Hybrid Systems. In *Proc. Hybrid and Real-Time Systems*, LNCS. Springer-Verlag, 1997.

[Mye78] G.J. Myers. *The Art of Software Testing.* John Wiley & Sons, New York, 1978.

[Nol99] Dr. Justus Noll. TeXtelmechtel – Die Geschichte des Satzsystems TeX. *c't*, (22):224–226, 1999.

[NS72] A. Newell and H.A. Simon. *Human Problem Solving.* Prentice-Hall, Englewood Cliffs, NJ, 1972.

[NSY93] X. Nicollin, J. Sifakis, and S. Yovine. From ATP to Timed Graphs and Hybrid Systems. *Acta Informatica*, 30:181–202, 1993.

[OB88] T. Ostrand and M. Balcer. The Category-Partition Method for Specifiying and Generating Functional Tests. *Communications of the ACM*, 31(6):676–686, 1988.

[Ost89] J.S. Ostroff. *Temporal Logic for Real-Time Systems.* Advanced Software Development Series. Research Studues Press (John Wiley & Sons), Taunton, Englang, 1989.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE Computer Society Press, 1977.

[Pos89] Michael I. Posner, editor. *Foundations of cognitive science*. Bradford Book, The MIT Press, 1989.

[Rei65] W.R. Reitman. *Cognition and Thought: An Information-Processing Approach*. Wiley, New York, 1965.

[Roh88] Hubert Rohracher. *Einführung in die Psychologie*. Psychologie Verlags Union, München, Weinheim, 13th edition, 1988.

[Rop94] M. Roper. *Software Testing*. McGraw-Hill Book Company, London, 1994.

[Roy93] T.C. Royer. *Software Testing Management: Life on the Critical Path*. Prentice-Hall International Lt., London, 1993.

[Sea80] John R. Searle. Minds, Brains, and Programs. In *The Behavioral and Brain Sciences*, volume 3. Cambridge University Press, 1980.

[Sel89] Robert Sell. *Angewandtes Problemlösungsverhalten – Denken und Handlen in komplexen Zusammenhängen*. Springer Verlag, Berlin, 1989.

[Sha94] Fred R. Shapiro. The first bug. *Byte*, page 308, April 1994.

[TDN93] M.Z. Tsoukalas, J.W. Duran, and S.C. Ntafos. On Some Reliability Estimation Problems in Random and Partition Testing. *IEEE Transactions on Software Engineering*, 19(7):687–697, 1993.

[Tra96] Dr. Antonio A. Trani. NSF SUCCEED Engineering Visual Database. Web-Site at cesun1.ce.vt.edu/evd/default.html, 1996.

[TW91] P. Thevenod-Fosse and H. Waeselynck. An Investigation of Statistical Software Testing. *Journal of Software Testing, Verification and Reliability*, 1(2):5–25, 1991.

[Weg01] J. Wegener. *Evolutionärer Test des Zeitverhaltens von Realzeit-Systemen*. PhD thesis, Humboldt-Universität zu Berlin, 2001. Shaker Verlag.

[WJ91] E.J. Weyuker and B. Jeng. Analyzing Partition Testing Strategies. *IEEE Transactions on Software Engineering*, 17(7):703–711, 1991.

[WO80] E.J. Weyuker and T.J. Ostrand. Theories of Program Testing and the Application of Revealing Subdomains. *IEEE Transactions on Software Engineering*, 6, 1980.

[WSJ+97] J. Wegener, H. Sthamer, B. Jones, et al. Testing Real-Time Systems using Genetic Algorithms. *Software Quality Journal*, 6(2):12–135, 1997.

[WT94] J.A. Whittaker and M.G. Thomason. A Markov Chain Model for Statistical Software Testing. *IEEE Transactions on Software Engineering*, 20(10):812–824, 1994.