

Vorwort

Software muss funktionieren, um von Kunden akzeptiert zu werden. Doch wie stellt man sicher, dass die Steuerung eines Raumschiffs, eines Herzschrittmachers oder einer Aktienverwaltung korrekt funktioniert? Neben den klassischen Testansätzen spielen für den Korrektheitsnachweis zunehmend Formale Modelle eine zentrale Rolle. Ein Modell erlaubt es, ein beliebiges sequenzielles oder verteiltes System zu analysieren und zu simulieren. Dadurch, dass Korrektheitsanforderungen präzise definiert werden, kann man sogar beweisen, dass die Anforderungen erfüllt sind. Dieses Buch stellt unterschiedliche Formale Modelle mit ihren Einsatzmöglichkeiten und Werkzeugen vor. Dabei steht bewusst die Anwendung der Modelle und nicht die Theorie dahinter im Vordergrund. Formale Modelle ermöglichen es dabei, Aussagen so zu präzisieren, dass sie von jedem, der das Modell versteht, eindeutig interpretiert werden können. Weiterhin kann für Formale Modelle nachgewiesen werden, dass sie bestimmte Eigenschaften haben. Abhängig von der Art des Formalen Modells erfolgt dieser Nachweis automatisch durch ein Programm oder durch den Entwickler, der Beweisregeln anwendet. Dieses Buch zeigt u. a., wie man Aussagen über Modelle und Anforderungen an Modelle präzise formulieren und ihre Gültigkeit nachweisen kann.

Formale Methoden nutzen Formale Modelle und sichern die Korrektheit von Software. Sie werden kontinuierlich seit den 1970er Jahren entwickelt. Die Erkenntnisse führten unter anderem zu den Turing Awards von C.A.R. Hoare (1980), A. Pnueli (1996) und E.M. Clark, E. A. Emerson und J. Sifakis (2007), deren Ansätze zur Programmverifikation und zum Model Checking in diesem Buch vorgestellt werden. Kritisch bleibt anzumerken, dass die formalen Ansätze nur schleppend auch in den sicherheitskritischen Bereichen (Raumfahrt, Luftfahrt, Automobilsysteme) Einzug in die Entwicklung finden. Ein zentraler Grund für dieses Defizit ist, dass die Ansätze bisher immer zusammen mit ihrem sehr komplexen theoretischen Hintergrund präsentiert wurden und es keine Einführungen für „Endanwender“ dieser Ansätze, also Entwickler, gibt. Diese Lücke wird durch dieses Buch geschlossen.

Nach einer kurzen Motivation des Themengebiets wird zunächst die Spezifikationsprache PROMELA zusammen mit dem Werkzeug SPIN von G. Holzmann vorgestellt, das 2001 den renommierten ACM Software System Award gewonnen hat. SPIN eignet sich zur Spezifikation, zur interaktiven und automatischen Simulation und zur Verifikation verteilter Systeme. Es wird auch gezeigt, wie man klassische sequenzielle Softwaresysteme als Spezialfall mit SPIN behandeln kann.

Timed Automata erlauben die Spezifikation von verteilten Systemen mit Zeit, die dann im Werkzeug Uppaal simuliert und verifiziert werden können. Mit Uppaal

V

steht dabei ein graphisches Werkzeug zur Verfügung, das verteilte Abläufe sehr gut visualisieren kann.

Petrinetze sind der klassische Einstieg in die systematische Analyse verteilter Systeme und in vielen Varianten theoretisch sehr gut untersucht. Neben der Anschauung steht in diesem Buch die Erkenntnis im Mittelpunkt, dass Ansätze aus verwandten wissenschaftlichen Disziplinen, hier der linearen Algebra, wesentliche Erkenntnisse über Software liefern können. Weiterhin wird gezeigt, wie schwer es ist, eine Komponente in einem Netzwerk verteilter kommunizierender Systeme auszutauschen.

Formalere Überlegungen zur Programmverifikation runden dieses Buch ab und zeigen die Fundierung der Probleme und Lösungsansätze für die Korrektheit von Programmen. Der praktische Nutzen wird mit einer Verknüpfung zur Welt des Testens aufgezeigt.

Die Kapitel zwei bis fünf sind so geschrieben, dass man sie von vorne nach hinten alleine oder in einer Lehrveranstaltung durcharbeiten kann. Es besteht aber auch die Möglichkeit, die Kapitel einzeln zu betrachten oder Kapitel auszulassen. Mit wenigen Querverweisen wird auf relevante Informationen anderer Kapitel verwiesen. Wer z. B. mehr am Ausprobieren und der Veranschaulichung interessiert ist, kann den Fokus auf die Kapitel zwei und drei legen, wer anfänglich eine theoretische Fundierung der Korrektheitsproblematik wünscht, sollte mit Kapitel fünf starten.

Alle in diesem Buch genauer betrachteten Werkzeuge können zum Lernen ohne Kosten aus dem Internet bezogen werden. Gerade diese Möglichkeit, nicht nur über die Werkzeuge zu lesen, sondern mit ihnen zu experimentieren, macht die vorgestellten Ansätze für Praktiker interessant. Statt nach theoretischen Antworten auf Fragen, wie ein Werkzeug in bestimmten Situationen reagiert, zu suchen, ist der Nutzer selbst zum Experimentieren aufgerufen. Die Übungsaufgaben dieses Buches bieten einen guten Startpunkt dazu.

Jedes Kapitel schließt mit zwei Arten von Aufgaben ab. Im ersten Aufgabenteil werden Wiederholungsfragen gestellt, die man nach intensiver Lektüre des vorangegangenen Kapitels beantworten können sollte. Die Lösungen zu diesen Aufgaben kann man selbst im Buch nachschlagen. Der zweite Aufgabenteil umfasst Übungsaufgaben, in denen man gezielt das angelesene Wissen anwenden soll. Diese Übungsaufgaben sind in verschiedenen Lehrveranstaltungen erfolgreich eingesetzt worden.

Die Bilder, Spezifikationen, Folien zum Buch und Lösungen zu Aufgaben dieses Buches sowie weitere Informationen können von der folgenden Web-Seite

<http://www.edvsz.fh-osnabrueck.de/kleuker/fmse/index.html>

oder von den Web-Seiten des Verlages zum Buch herunter geladen und unter Berücksichtigung des Copyrights genutzt werden.

In diesem Buch benutze ich verkürzend ohne Hintergedanken beim Singular wie Leser oder Entwickler die männliche Form. Natürlich möchte ich mit diesem Buch auch die weiblichen ~~Leser~~ Leserinnen ansprechen.

Zum Abschluss wünsche ich Ihnen viel Spaß beim Lesen. Konstruktive Kritik wird immer angenommen. Bedenken Sie, dass das Lesen nur ein Teil des Lernens ist. Ähnlich wie in diesem Buch kleine Beispiele eingestreut sind, um einzelne Details zu klären, sollten Sie sich mit den hier vorgestellten Ideen hinsetzen und meine, aber vor allem selbst konstruierte Beispiele durchspielen. Sie runden das Verständnis des Themas wesentlich ab.

Osnabrück, Oktober 2009

Stephan Kleuker

Danksagung

Ein Buch kann nicht von einer Person alleine verwirklicht werden. Zu einer gelungenen Entstehung tragen viele Personen in unterschiedlichen Rollen bei, denen ich hier danken möchte.

Mein erster Dank geht an meine Ehefrau Dr. Cheryl Kleuker, die nicht nur die erste Kontrolle der Inhalte und Texte vorgenommen hat, sondern mir erlaubte, einen Teil der ohnehin zu geringen Zeit für die Familie in dieses Buchprojekt zu stecken.

Besonderer Dank gilt Henning Dierks, Jutta Göers, Elke Pulvermüller, Wolfgang Runte und Frank Thiesing, die unterschiedliche Teile einer Vorversion dieses Buches kritisch durchgelesen haben und interessante Anregungen lieferten. Viele Studierende, die Veranstaltungen zum Thema Formale Methoden bei mir gehört haben, trugen durch ihre Fragen und Probleme wesentlich zu der Herangehensweise an die Themen des Buches bei.

Abschließend seien Sybille Thelen, Andrea Broßler, Albrecht Weis und den weiteren Mitarbeitern des Verlags Vieweg+Teubner für die konstruktive Mitarbeit gedankt, die dieses Buchprojekt erst ermöglichten.

für Cheryl und Lisa

Inhaltsverzeichnis

Vorwort	V
1 Motivation von Formalen Modellen	1
1.1 Modellbegriff.....	1
1.2 Software-Fehler	3
1.3 Software-Engineering	7
1.4 Fragen.....	11
2 Modelchecking mit PROMELA und SPIN	13
2.1 Modelchecking im Entwicklungskontext.....	14
2.2 Die Spezifikationssprache PROMELA	16
2.3 Simulation von PROMELA-Spezifikationen	36
2.4 Einfache Verifikationsmöglichkeiten.....	49
2.4.1 Grundideen des Modelcheckings.....	49
2.4.2 Modelchecking sehr großer Systeme.....	56
2.4.3 Lebendigkeit und Sicherheit.....	59
2.4.4 Zusicherungen	60
2.4.5 Prozessterminierung	63
2.4.6 Ausführung einfacher Verifikationen	63
2.4.7 Nachweis von Lebendigkeitseigenschaften	67
2.4.8 Trace-Zusicherungen	70
2.4.9 Never-Claims.....	72
2.5 Verifikation von in LTL formulierten Anforderungen.....	77
2.6 Beispiele	84
2.7 PROMELA und SDL	98
2.8 Aufgaben	106
3 Modelchecking mit Timed Automata und Uppaal.....	117
3.1 Synchron kommunizierende Automaten.....	118
3.2 Spezifikationen mit Zeit.....	125
3.3 Nutzung von Uppaal	131
3.4 Timed Computation Tree Logic und Verifikation	138
3.5 Beispiele	146

3.6 Aufgaben.....	157
4 Petrinetze	163
4.1 Funktionsweise von Petrinetzen.....	163
4.2 Erreichbarkeitsgraphen und Überdeckungsgraphen	169
4.3 S- und T-Invarianten	174
4.4 Werkzeuggestützte Analyse von Petrinetzen.....	180
4.5 Beispiele	187
4.6 Äquivalenzen von Petri-Netzen	191
4.7 Aufgaben.....	197
5 Programmverifikation.....	201
5.1 Eine einfache Programmiersprache	202
5.2 Operationelle Semantik	204
5.3 Zusicherungen	211
5.4 Partielle und totale Korrektheit	214
5.5 Beweissysteme für partielle und totale Korrektheit	219
5.6 Programmverifikation in der Praxis	233
5.7 Syntax und Semantik paralleler Programme	242
5.8 Beweissysteme für parallele Programme	250
5.9 Parallele Programmierung	259
5.10 Aufgaben.....	266
A Installationsbeschreibungen	275
A.1 Installation von SPIN	275
A.2 Installation von Uppaal	277
A.3 Installation von NetLab	279
B Kurzeinführung in Java	281
B.1 Grundlagen	281
B.2 Threads	288
Literaturverzeichnis.....	293
Sachwortverzeichnis.....	297

X

einfach“, dass die zugehörigen Daten schnell erhoben und einfach bearbeitbar sind. Maßgeblich kann dazu beitragen, dass nur relativ wenige Daten benötigt werden.

Die Suche nach einem passenden Modell ist dabei ein sehr kreativer Prozess, der durch die Kenntnisse unterschiedlicher Modelle fundiert werden kann. Ein Beispiel für eine innovative Wahl eines Modells stellt die Lösung zur Fragestellung nach der Geschwindigkeit einer Ausbreitung einer Epidemie in den USA dar. Zur Analyse werden wieder typischerweise sehr zeitaufwändig und teuer zu erhebende Bewegungsdaten benötigt. Der Modellansatz der Forscher war es allerdings, auf die Daten einer netten Internet-Spielerei zuzugreifen, an der sich relativ viele Amerikaner beteiligen. Auf der Internet-Seite „Where is George“ [WiG], kann man die Seriennummern seiner Ein-Dollar-Scheine (mit einem Bild von George Washington) und seinen Standort eintragen. Dadurch, dass eindeutige Seriennummern mehrfach auftreten, kann man sehen, von wo nach wo sich Dollarscheine und damit ihre zwischenzeitlichen Besitzer bewegt haben. Ein zentrales Ergebnis der Untersuchung ist, dass sich Epidemien wesentlich schneller von der Ost-Küste auf die West-Küste ausbreiten können, als vorher berechnet.

Auch bei formalen Modellen werden wir feststellen, dass sie verschiedene Aspekte, z. B. die Nutzung von Zeit betonen und vernachlässigen können, so dass man über die Nutzung mehrerer Modelle für unterschiedliche Anforderungen nachdenken kann.

1.2 Software-Fehler

Die relativ kurze Geschichte der Software-Entwicklung, die in den 1930er-Jahren begann [Zus07], ist geprägt durch zwei markante Eigenheiten. Auf der einen Seite steht der enorme Fortschritt in der Hardware-Entwicklung im Zusammenhang mit der stark steigenden Komplexität von Software, die bereits Einzug in einfachste Haushaltsgeräte gefunden hat. Auf der anderen Seite steht die lange Liste von markanten Software-Fehlern oder Fehlern im Zusammenspiel von Software und Hardware, die Menschenleben und viele Milliarden Euro gekostet haben. Seit Mitte der 1960er Jahre wird die kreative Energie der relativ wenigen Personen, die sich bis dahin mit der Software-Entwicklung beschäftigt haben, immer mehr systematisiert, so dass daraus die ingenieur-wissenschaftliche Disziplin des Software-Engineering wurde. Bevor wir uns mit den Errungenschaften des Software-Engineering im Hinblick auf die Korrektheit von Programmen beschäftigen, sollen hier zur Mahnung einige der markantesten Software-Fehler zusammengefasst werden. Interessierte Leser seien dazu auch auf die vielen interessanten und teilweise dramatischen Berichten hingewiesen, die man bei der Suche im Internet nach Begriffen wie „Software-Fehler“, „Software-Problem“ oder deren englischen Varianten findet.

1 Motivation von Formalen Modellen

Nr.	Jahr	Vorfall
1.	1962	Mariner 1, Cape Canaveral, Trägerrakete sprengt sich nach Kursabweichung selbst, Grund für die Abweichung war ein Komma statt eines Punktes im Quellcode
2.	1971	Eole1, französischer Satellit zur Koordination von 141 Wetterballons; 72 Ballons erhalten Befehl vom Satelliten, Daten zu senden, interpretieren dies aber als Befehl zur Selbstzerstörung
3.	1978	Amerikanisches Jagdflugzeug F16, im Simulationsbetrieb wird entdeckt, dass sich das Flugzeug beim Überfliegen des Äquators wegen eines Vorzeichenfehlers auf den Kopf dreht
4.	1979	Programm zur Berechnung der Erdbebenfestigkeit amerikanischer Kernkraftwerke berechnet kritische Werte, was zu Umbaumaßnahmen führt; Ursache der kritischen Werte war das Berechnen der arithmetischen Summe statt der Wurzel aus der Quadratsumme
5.	1982	Falkland Krieg, britische Fregatte Sheffield identifiziert anfliegenden argentinischen Flugkörper als freundlich, da er von einer befreundeten Nation hergestellt wurde; die Fregatte wurde zerstört
6.	1987	Therac 25, Gerät zur medizinischen Strahlentherapie; durch die zu schnelle Eingabe wird die Bestrahlung ausgelöst, bevor die Positionierung abgeschlossen ist
7.	1988	Airbus A 320, Absturz bei einer Flugshow, da die besondere Flugsituation mit niedriger Höhe und steilem Winkel nicht von der Software unterstützt wurde
8.	1990	AT & T-Netzausfall, 60000 Menschen können mehrere Stunden nicht telefonieren, nachdem eine neue Software eingespielt wurde und sich die verbundenen Vermittlungsrechner gegenseitig im Domino-Effekt abschalteten
9.	1990	Geldautomat, Person will Geld an einem Geldautomaten in Honolulu abheben, der zentrale Rechner zur Überprüfung steht in New Jersey, durch die Zeitverzögerung des Satellitensignals und einen Protokollfehler wird Geld nicht ausgezahlt und trotzdem abgebucht
10.	1991	Flugabwehrsystem Patriot verfehlt im Golfkrieg anfliegende irakische Scud-Rakete; Problem war, dass die Zeit nur in einer einfach genauen Gleitkommazahl gespeichert wurde, was eine vergessene Korrektur der Zeit nach maximal 100 Stunden Laufzeit benötigte
11.	1993	Pentium-Chip; durch einen Berechnungsfehler ist die Division bei diesem Chip ungenau, was sich erst mehrere Stellen hinter dem Komma auswirkt, allerdings zu einem sehr teuren Prozessoraustausch führt
12.	1993	Programm zur Steuerberechnung in den USA berechnet wegen zwei Software-Fehlern Steuern falsch, das Programm war unter Steuerberatern weit verbreitet
13.	1994	Mondsonde Clementine; nach dem erfolgreichen Fotografieren der Mondoberfläche sollte die Sonde einen Asteroiden ansteuern, durch einen Software-Fehler wurden die Triebwerke nach der Kurskorrektur nicht mehr abgestellt

Nr.	Jahr	Vorfall
14.	1995	Telefonnetz Singapur; 65% der Leitungen sind fünf Stunden nicht nutzbar, die restlichen stark überlastet, da sich Software-Fehler eines Vermittlungsknotens auf weitere Systeme fortpflanzte
15.	1995	Automatischer Gepäcktransport am Flughafen Denver; der Start des neuen Flughafens verzögert sich um mehrere Monate, da das Zusammenspiel zwischen Soft- und Hardware (zu viele Steuerungsinformationen) zum vollautomatischen Gepäcktransport zum Verlust und zur Beschädigung des Gepäcks führt
16.	1995	Bahnstellwerk Hamburg-Altona; durch einen Speicherüberlauf bei hohem Verkehrsaufkommen stürzt der Rechner ab; es werden drei Tage zur Reparatur benötigt
17.	1996	Abrechnungssystem der Telecom erkennt 1. Januar nicht als Feiertag und rechnet falsch ab
18.	1996	Ariane 5; Hauptrechner sendet unsinnige Befehle an Triebwerke, Rakete zerstört sich selbst, Problem war die ungeprüfte Übernahme von Software der Ariane 4, die für auftretende Beschleunigungen nicht konzipiert war
19.	1996	Passwort-System Kerebos; durch einen Fehler in diesem Protokoll können die genutzten Zufallszahlen vorausberechnet werden, so dass ein Eindringen in die Systeme möglich wird; dieser Fehler existierte acht Jahre
20.	1998	NEAR-Sonde; die Sonde soll zum Asteroiden Eros fliegen; die Zündung der Antriebsraketen für das Verlassen der Erdumlaufbahn wird von der Kontrollsoftware abgebrochen, um Fehlzündungen zu vermeiden; eine Korrektur der Software vom Boden aus gelang
21.	1999	Mars Climate Orbiter; Marssonde geht in 170 km zu tiefe Umlaufbahn, so dass die Sonde abstürzt; die Entwickler der Software gingen von einem metrischen System mit Metern aus, das Navigationssystem erwartete Angaben im englischen System in Yards
22.	1999	Mars Polar Lander; Landefahrzeug stürzt aus 40 Metern Höhe auf die Oberfläche; drei Sensoren hatten die Erschütterung beim Ausfahren der Landebeine als Bodenkontakt interpretiert
23.	1999	Software des Wetterdienstes interpretiert schnellen Druckabfall als Messfehler, ohne Meteorologen zu informieren; die notwendige Sturmwarnung für Süddeutschland bleibt aus
24.	2001	Fehlerhafte Krebsbestrahlung von Patienten; Bedienfehler waren von der Software nicht erkannt worden, wovon beim Konzept der Bedienoberfläche ausgegangen wurde
25.	2002	Bomben AGM 154-A schlagen bei amerikanischer Irak-Bombardierung wegen Berechnungsfehler 15 bis 250 Meter zu weit links ein
26.	2002	Berechnung der Delegiertenanzahl für Landesparteitag der Grünen in Baden-Württemberg scheitert durch Fehler in der Nutzung eines Tabellenkalkulationsprogramms; es werden 202 statt benötigter 200 Personen eingeladen
27.	2002	Teilweiser Absturz des britischen Luftraumüberwachungssystems beim Versuch, aktualisierte Software in das System einzuspielen; Resultat waren deutliche Verzögerungen im Flugbetrieb

1 Motivation von Formalen Modellen

Nr.	Jahr	Vorfall
28.	2003	Flugabwehrsystem Patriot identifiziert in Kuwait befreundetes britisches Kampfflugzeug als Feind und schießt es ab
29.	2003	Stromausfall in USA/Kanada; 50 Millionen Menschen ohne Strom, da die Software des Netzbetreibers auf eine große Lastschwankung des Netzes falsch reagierte
30.	2003	Autobahn-Maut-System; Fehler im Zusammenspiel mit der Hardware; Anzeige mautfreier Strecke auf zu bezahlenden Strecken, Verweigerung, Eingaben anzunehmen im laufenden Betrieb, verschiedene Beträge auf gleichen Strecken bei gleichen Konditionen
31.	2004	Rückruf von Mercedes-Transportern, da Software zur Dieselsteuerung die Kraftstoffzufuhr in unerwarteten Momenten drosselte, wodurch der Motor ausgehen konnte
32.	2004	Mars-Rover Spirit; Probleme in der Steuerung des Mars-Untersuchungsgerätes, da nach einiger Zeit das Dateiverwaltungssystem zu viel Speicher benötigt, was zu einem Reset des Systems führt
33.	2005	Fehler in der Software zur Berechnung des Arbeitslosengeldes 2 der Bundesagentur für Arbeit, da unter falschen Annahmen die Krankenkassenzahlungen storniert wurden
34.	2005	Fehler in der Software zur Berechnung des Arbeitslosengeldes 2 der Bundesagentur für Arbeit, Auszahlungen fanden nicht statt, da bei neunstelligen Kontonummern am Ende eine Null ergänzt wurde, um auf das gewünschte zehnstellige Format zu kommen
35.	2005	Londoner Tower-Bridge für einen halben Tag nicht passierbar; durch einen Software-Fehler ist die Brücke nach dem Öffnen nicht schließbar
36.	2005	Durch einen Software-Fehler bei der Weiterleitung von E-Mails eines Providers wird eine E-Mail eines Abgeordneten des sächsischen Landtages mehrere Tausend Mal an die Empfänger zugestellt
37.	2005	Cryosat-Trägerrakete, durch einen Software-Fehler der neuen russischen Trägerrakete, bei dem ein Signal zum Abschalten der Triebwerke der zweiten Stufe nicht weiter geleitet wurde, stürzt die Rakete ab
38.	2007	Die Tabellenkalkulation Excel 2007 zeigt bei Multiplikationen, die 65535 ergeben, z. B. $10.2 * 6425$, den Wert 100000 an
39.	2008	Die Eröffnung des fünften Terminals des Flughafens Heathrow verzögert sich und verzögert dann den Flugbetrieb wegen Problemen mit der Gepäckabfertigungssoftware
40.	2008	Musikplayer mit dem Freescale MC13783-Prozessor, z. B. Zune von Microsoft, versagt am 31.12.2008 vollständig den Dienst, da es einen Fehler in der Schaltjahresberechnung gibt
41.	2009	Bei der Umstellung der Bearbeitung der „Abwrackprämie“ werden durch Überlast verschiedenen Anträgen die gleichen Bearbeitungsnummern zugeteilt, so dass Betätigungsmails private Daten willkürlich verteilen
42.	2009	Durch Fehler in der neuen Version einer Software zur Verwaltung der Handy-Nummern der Telecom kann ein Großteil der Nutzer das Handy für vier Stunden nicht nutzen

Abb. 1: Beispiele für Software-Fehler

Abb. 1 ist zwar durchnummeriert, da die einzelnen Ereignisse noch genauer betrachtet werden sollen, aber natürlich hat sie keinen Anspruch auf Vollständigkeit, wobei sie aus einigen Quellen zusammengetragen wurde [Jau], [Spi], [Gie], [aeo], [gol] und ein Schwerpunkt auf aktuelleren Problemen liegt. Da für die genannten Problemfälle nur selten der Quellcode bekannt wurde, ist kritisch zu bemerken, dass Software-Probleme natürlich auch vorgeschoben werden können und dass echte Fehler meist ihre Ursachen in ganz anderen Dingen, wie mangelnden Zeit- und Geldressourcen für Software-Tests, haben können.

Neben den genannten markanten Beispielen sind schleichende Fehler in der IT-Branche ein großes Problem. Dies sind Fehler, die nicht sofort zu einem Desaster führen, sondern dazu, dass eine sich langsam weiterentwickelnde Software immer schwieriger zu erweitern wird, bis letztendlich keine neuen Software-Versionen mehr ausgeliefert werden können.

Beispielhaft sei aus eigener Erfahrung die Geschichte eines kleinen Software-Unternehmens genannt, das in der Nahrungsmittelbranche erfolgreich Lösungen zur Verwaltung des Einkaufs von Schlachtvieh über die Steuerung von Produktionsanlagen bis zur Auslieferung erstellte. Die erste Software wurde von wenigen Entwicklern mit großem Know-how in der Lebensmittelbranche entwickelt. Die Software wurde an wenige Kunden verkauft und vor Ort auf die individuellen Bedürfnisse angepasst. Dieser Weg war anfänglich sehr erfolgreich und führte zu hoher Kundenzufriedenheit. Mit steigender Kundenzahl wurden neue Module entwickelt, die zur existierenden Software passen mussten. Da es nicht nur eine Ausgangssoftware für Änderungen gab – die kundenindividuellen Programme mussten ja weiterlaufen – und da zentrale Informationen teilweise mehrfach im System gehalten wurden, stieg der Aufwand für jede neue Software-Entwicklung enorm. Letztendlich wurde die Software unwartbar, so dass Verträge nicht eingehalten wurden und das Unternehmen nach einem Verkauf geschlossen wurde.

Aus modelltechnischer Sicht fehlte die Erkenntnis, dass man die Daten so strukturieren muss, dass es für jede Information nur eine Quelle geben darf und dass nur lokal benötigte Informationen für andere nicht sichtbar werden sollten. Mit einem Datenmodell, aus dem Abhängigkeiten von Daten ableitbar gewesen wären, hätte die Erweiterungsfähigkeit langfristig erhalten bleiben können.

1.3 Software-Engineering

Um der steigenden Komplexität von Software-Systemen Herr zu werden, wurden verschiedene prozessorientierte Ansätze entwickelt, die Fehlerquellen minimieren, beziehungsweise zur frühzeitigen Entdeckung von Fehlern führen sollen. Dabei spielt die Erkenntnis, dass komplexe Abläufe systematisch als nachvollziehbare Prozesse aufbereitet werden müssen, eine zentrale Rolle. Die Erkenntnis, dass für jeden Arbeitsablauf klar sein muss,

2 Modelchecking mit PROMELA und SPIN

Generell besteht der Wunsch, dass die Korrektheit eines Programms automatisch geprüft werden kann. Dabei muss zunächst geklärt werden, was mit Korrektheit gemeint ist. Dies wird in diesem Kapitel durch Anforderungen präzisiert, die ein Programm erfüllen soll. Aus theoretischer Sicht weiß man, dass ein allgemeines Verfahren, die Korrektheit von Programmen bezüglich gegebener Anforderungen zu zeigen, durch die Nichtentscheidbarkeit vieler Probleme generell nicht möglich ist. Auch aus praktischer Sicht stellt sich die Frage, wie man dies bei all den möglichen Zuständen einer komplexen Software in den Griff bekommen kann. In diesem Kapitel sehen wir, dass man einiges in den Griff bekommt, wenn man es schafft, ein einfaches passendes Modell zu einer Aufgabenstellung zu finden. Auch das Problem mit den eventuell unendlich vielen möglichen Zuständen kann man lösen, wenn man sicher ist, dass nur endlich viele dieser Zustände interessant sind.

Modelchecking erlaubt die Analyse großer Zustandsmengen und nutzt den folgenden Ansatz: Gegeben sind ein Modell und eine Anforderung, dann überprüft der Modelchecking-Algorithmus, ob das Modell die Anforderung erfüllt oder nicht. Falls das Modell die Anforderung nicht erfüllt, sollte der Algorithmus ein Gegenbeispiel liefern.

Die Umsetzung erster Modelchecking-Ideen [Pnu77] [WVS83], [CES86], [BCM90], die sich auf Ende der 1970er Jahre datieren lassen, fand Mitte der 1980er Jahre statt, da es ab dann neben der Theorie für diesen Ansatz auch verfügbare Computer gab, mit denen große Zustandsräume durchsucht werden konnten. Die grundlegenden Arbeiten von Clarke, Emerson und Sifakis wurden u. a. in 2007 mit der renommiertesten Auszeichnung im Informatik-Umfeld, dem Turing Award, ausgezeichnet.

Für die praktische Nutzbarkeit spielt neben der wesentlichen Effizienz des Modelchecking-Verfahrens die möglichst einfache Modellierungssprache, d. h. eine möglichst einfache Sprache zur Formulierung der Modelle und Anforderungen sowie ihre Integration in einem bedienbaren Werkzeug, eine wichtige Rolle.

Als erstes Werkzeug wird hier SPIN (Simple PROMELA Interpreter) [Hol04] vorgestellt, das von Gerard Holzmann 1991 veröffentlicht [Hol91] und seitdem kontinuierlich verbessert wurde. Das Werkzeug steht auf der zugehörigen Internetseite [SPI] mit Installationsanleitungen und vielen weiteren Informationen frei zur Verfügung. Einige Installationsdetails sind im Anhang A.1 zusammengefasst. Holzmann hat mit diesem Werkzeug 2001 den renommierten ACM Software System Award gewonnen, der z. B. 1983 von UNIX, 1987 von Smalltalk, 1991 von TCP/IP, 1995 vom World-Wide Web und 2002 von Java gewonnen wurde [ACM].

SPIN ist auch aus einem anderen Gesichtspunkt interessant, da es neben der Verifikationskomponente auch die Möglichkeit zur Simulation der erstellten Modelle gibt.

Mit PROMELA steht eine mächtige Spezifikationssprache zur Verfügung, mit der nichtdeterministische, kommunizierende verteilte Systeme spezifiziert werden können. Wichtig ist dabei, dass eine Spezifikationssprache verschiedene Sprach-elemente anbietet. Der Nutzer muss entscheiden, ob die angebotenen Ansätze für seine praktische Aufgabenstellung nutzbar sind oder nicht. Kann man z. B. in der Zielimplementierung keine gemeinsamen Variablen für verteilte Prozesse nutzen, sollte man intensiv darüber nachdenken, ob man diese Möglichkeit in der Spezifikation nutzt.

Weiterhin werden wir sehen, dass man auch klassische deterministische Programme als Spezialfall in PROMELA beschreiben kann.

Im folgenden Abschnitt wird zunächst beschrieben, welche Bedeutung Modelchecking im Rahmen der Software-Entwicklung haben kann. Danach wird die zu SPIN gehörige Spezifikationssprache PROMELA (Process Meta Language) vorgestellt und gezeigt, wie sie zunächst in Simulationen genutzt werden kann. Die Formalisierung von Anforderungen findet in einer temporalen Logik statt, die dann eingeführt wird. Abschließend werden mehrere Fallstudien diskutiert und die Verwandtschaft zur Spezifikationssprache SDL gezeigt.

2.1 Modelchecking im Entwicklungskontext

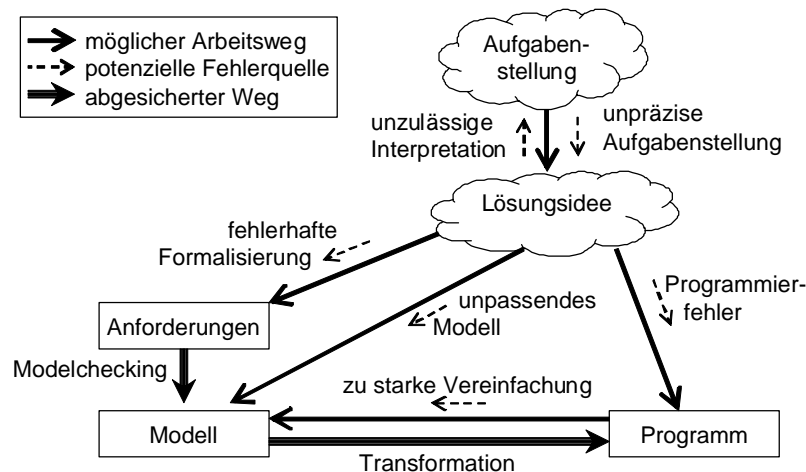


Abb. 3: Fehlerquellen der Programmentwicklung

Um zu verstehen, welche Möglichkeiten das Modelchecking bietet und welche Probleme trotzdem bleiben, ist es hilfreich, Abb. 3 zu analysieren. Die einfachen durchgezogenen Pfeile zeigen mögliche Arbeitswege in der Software-Entwicklung. Ausgehend von der Aufgabenstellung wird eine Lösungsidee entwickelt, die dann z. B. direkt in einem Programm ausprogrammiert wird. In einem formaleren Weg werden, ausgehend von den Ideen, Anforderungen formalisiert, die zur Überprüfung des Programms genutzt werden. Da es oft eine sehr große Herausforderung ist, nicht-triviale Programme zu verifizieren, wie später noch gezeigt wird, wird jetzt zusätzlich ein Modell für die erdachte Lösung erstellt. Dieses Modell wird entweder aus der Lösungsidee entwickelt oder aus dem existierenden Programm abgeleitet. In einem Spezialfall kann man das Programm selbst als das Modell auffassen. Dies ist nur bei einfachen Programmen möglich, da ein Modell handhabbar sein soll und so auf zu untersuchende kritische Elemente reduziert wird.

Durch Modelchecking ist es möglich, automatisch nachzuprüfen, ob ein Modell die formulierten Anforderungen erfüllt. Wie ein solches Modell und die Anforderungen beschrieben werden, ist Schwerpunkt der folgenden Unterkapitel. Die Verknüpfung zwischen Modell und Anforderungen ist in Abb. 3 fett eingezeichnet. Ein zweiter dicker Pfeil zeigt die häufig gegebene Möglichkeit, ein Modell automatisch in ein Programm zu übersetzen, genauer: zu transformieren. Dieser Transformationsschritt, der für verschiedene Modellierungssprachen möglich ist, wird in diesem Buch nicht weiter betrachtet, da er stark davon abhängt, welche Ziel-Programmiersprache und Ziel-Hardware das Projekt hat.

Durch die gestrichelten Pfeile in Abb. 3 sind mögliche Fehlerquellen angedeutet. Dabei wird sichtbar, dass Modelchecking kein Allheilmittel sein kann, da auch als korrekt bewiesene Modelle im Sinn der Aufgabenstellung fehlerhaft sein können. Die erste zentrale Fehlerquelle ist die Umsetzung der Aufgabenstellung in die Lösungsidee, die in der Praxis meist den kritischsten Teil eines Projekts ausmacht. Projekte scheitern häufig, da dem Kunden unklar ist, was im Detail seine Forderungen sind. Diese Fehlerquelle wird verstärkt, wenn auf Entwicklerseite nicht im Detail nachgefragt wird und Annahmen über das zu entwickelnde System gemacht werden, die unzutreffend sind. Eine weitere gravierende Fehlerquelle sind klassische Programmierfehler, die z. B. auf vergessenen Ablaufalternativen beruhen.

Auch bei der Formulierung von Anforderungen und Modellen können Fehler auftreten, so dass jeweils Experten für den Arbeitsschritt benötigt werden. Ist z. B. jemand nicht in der Lage, die Anforderungen korrekt in der für den Modelchecker genutzten Logik auszudrücken, haben die nachgewiesenen Anforderungen wenig Aussagekraft. Im schlimmsten Fall werden Anforderungen falsch in Logik umgesetzt, diese falschen Formeln als erfüllt nachgewiesen und dann, basierend auf der vermeintlich nachgewiesenen Anforderung, kritische Entscheidungen getroffen. Auch bei der Modellierung selbst können Fehler auftreten, so dass man sich bei jedem gescheiterten Modelchecking auch fragen muss, ob das Modell und die An-

forderungen korrekt sind. Leider ist auch der andere Weg möglich, dass eine Anforderung falsch umgesetzt und dann als korrekt bewiesen wird.

Eine letzte in der Abbildung nicht eingezeichnete Fehlerquelle stellt natürlich die Möglichkeit eines fehlerhaften Modelcheckers oder fehlerhafter Transformationen dar. Die typische Antwort auf die Frage „wer prüft den Prüfer“ stellt dabei der pragmatische Ansatz dar, dass ein über lange Zeit funktionierendes Prüfverfahren „wohl“ wenig Fehler aufweisen wird. Der theoretisch für andere Systeme verfolgte Ansatz, dass ein Werkzeug zur Prüfung der eigenen Algorithmen genutzt wird, wird für SPIN nicht verfolgt.

2.2 Die Spezifikationssprache PROMELA

Für eine Spezifikationssprache, die zum Modelchecking genutzt werden soll, kann man zwei zentrale Forderungen formulieren:

1. Die Modelle sollten möglichst einfach beschreibbar sein.
2. Die Modelle sollten möglichst klein sein.

Um die Gefahr falscher Modelle zu reduzieren, ist die Spezifikationssprache PROMELA (PROcess MEta LAnguage) an einfache Programmiersprachen angelehnt. Einzelne Programmkonstrukte sind aus der Programmiersprache C entnommen.

Damit die Modelle bei der Untersuchung klein bleiben, muss für Variablen der Datentyp möglichst genau festgelegt werden, dabei spielt jedes eventuell überflüssige Bit eine Rolle.

PROMELA wird u. a. zur Spezifikation von Kommunikationsprotokollen eingesetzt, weiterhin eignet sich die Sprache sehr gut zur Modellierung verteilter Systeme oder paralleler Algorithmen, die gemeinsame Variablen nutzen. Sie wurde um spezielle Möglichkeiten zur Kommunikation zwischen den Prozessen durch verschiedene Varianten von Kommunikationskanälen erweitert.

In diesem Buch werden zentrale Sprachelemente von PROMELA vorgestellt; weitere Möglichkeiten und noch weiterführende Diskussionen können [Hol04] entnommen werden. Eine weitere Variante einer Einführung befindet sich in [Ben08].

Zur Einführung werden jetzt kleine Spezifikationen betrachtet, die vielleicht nicht typisch für das Einsatzgebiet von PROMELA sind, aber wesentlich zum Verständnis der Sprache beitragen.

Die Spezifikation in Abb. 4 beinhaltet drei Prozesse, deren Aufgabe es ist, die Summe der ersten N ganzen Zahlen zu berechnen. Im Beispiel gibt es einen Start-Prozess, der mit dem Schlüsselwort `init` gekennzeichnet ist. Durch das Schlüsselwort `run` ist es möglich, Prozesse zu starten. Im Beispiel gibt es einen Prozess, der die ungeraden Zahlen, und einen zweiten Prozess, der die geraden Zahlen aufsummieren soll.

```

#define N 20
int erg;

proctype Ungerade(){
  int x=1;
  do
    :: x<=N -> erg=erg+x;
               x=x+2
    :: x>N -> break
  od
}

proctype Gerade(){
  int x=0;
  do
    :: x<=N -> erg=erg+x;
               x=x+2
    :: x>N -> break
  od
}

init{
  erg=0;
  run Ungerade();
  run Gerade()
}

```

Abb. 4: Erste Beispielspezifikation in PROMELA

Oberhalb der einzelnen Prozessdefinitionen werden gemeinsame Eigenschaften für alle Prozesse festgehalten. Konstanten können wie in C mit dem Makro-Befehl `#define` definiert werden. Bei der späteren Ausführung wird der Text, hier `N`, durch den folgenden Text, hier `20`, ersetzt.

Datentyp	Wertebereich	Anmerkung
bit	0..1	
bool	0..1	auch Werte <code>true (==1)</code> und <code>false (==0)</code> möglich
byte	0..255	
chan	1..255	Kommunikationskanäle (später)
mtype	1..255	Nachrichtenwerte (später)
pid	0...255	für Prozessidentifikatoren (später)
short	$-2^{15}..2^{15}-1$	
int	$-2^{31}..2^{31}-1$	
unsigned	$0..2^{\text{Wert}}-1$	Anzahl Bits <Wert> wird angegeben, z.B. <code>unsigned x:5;</code> Variable hat dann Maximalwert 31

Abb. 5: Basisdatentypen in PROMELA

2.8 Aufgaben

Wiederholungsfragen

Versuchen Sie zur Wiederholung folgende Fragen aus dem Kopf, d. h. ohne nochmaliges Blättern und Lesen, zu beantworten.

1. Erklären Sie aus Sicht der Logik den Begriff Modelchecking.
2. Welche Fehlerarten kann man mit Modelchecking vermeiden, welche nicht?
3. Erklären Sie anschaulich die Semantik von `if` und `do` in PROMELA.
4. Erklären Sie die besondere Bedeutung von Booleschen Bedingungen wie `(fertig < 4)` in PROMELA.
5. Wie kann man neue Datentypen in PROMELA definieren?
6. Welche Möglichkeiten gibt es in PROMELA, Prozesse zu definieren und zu starten?
7. Wozu gibt es atomare Bereiche, wie werden sie eingesetzt, was passiert, wenn sie nicht weiter ausgeführt werden können?
8. Wie kann man die Kommunikation von Prozessen in PROMELA spezifizieren?
9. Was versteht man unter synchroner und asynchroner Kommunikation, welches Verhalten steckt bei der Ausführung dahinter?
10. Welche zusätzlichen Möglichkeiten gibt es in PROMELA bei asynchroner Kommunikation außer dem reinen Empfangen?
11. Was sind Message Sequence Charts, wozu können sie eingesetzt werden?
12. Was ist der Zustandsraum einer Spezifikation, welche Bedeutung hat er für das Modelchecking, wie kann er verwaltet werden?
13. Wie kann man beim Modelchecking mit sehr großen Spezifikationen umgehen?
14. Was versteht man unter Sicherheits- und Lebendigkeitseigenschaften?
15. Was unterscheidet unfair, schwach fair und stark fair Systeme?
16. Wie kann man mit Hilfe von `assert` mit SPIN verifizieren?
17. Wozu dienen `end`-Markierungen in PROMELA?
18. Wozu dienen `progress`-Markierungen in PROMELA?
19. Was sind Trace-Zusicherungen, wie werden sie überprüft?
20. Was sind Never-Claims, wie werden sie überprüft?
21. Wie ist die Syntax und Semantik von Formeln der Linearen Temporalen Logik definiert?

22. Nennen Sie typische Arten von Anforderungen an PROMELA-Spezifikationen und ihre Syntax in LTL.
23. Wie können LTL-Formeln im Zusammenhang mit SPIN genutzt werden?
24. Erklären Sie, wie man mit SDL Prozesse spezifizieren kann.
25. Wie werden Nachrichten in SDL verarbeitet?
26. Welche Zusammenhänge kann man zwischen PROMELA und SDL aufbauen?

Übungsaufgaben

1)

- a) Geben Sie folgende Spezifikation in XSPIN ein.

```
byte x=0;

active proctype P1(){
    x=x+1;
    x=x+1
}

active proctype P2(){
    x=2
}
```

Überprüfen Sie die Syntax, stellen Sie als Simulationsart „Interactive“ ein und starten Sie die Simulation. Versuchen Sie dann durch verschiedene von Hand gesteuerte Simulationen unterschiedliche Ergebnisse für x zu erreichen. Welche Ergebnisse sind möglich? (Um eine Simulation zu beenden, drücken Sie „Cancel“ im Fenster „Simulation Output“.)

- b) Schreiben Sie eine PROMELA-Spezifikation, mit der Sie feststellen können, wie sich der Datentyp byte bei einem Überlauf und bei einem Unterschreiten des Wertebereichs verhält.
- c) Nutzen Sie den Nichtdeterminismus, um in einer globalen Variable x eine zufällige ganze Zahl zwischen 0 und 100 zu generieren; dabei soll die Zahl nicht durch 10 teilbar sein. (Modulorechnung $7 \% 3$ ist 1, mit 1 als Rest bei der Division, funktioniert auch in PROMELA.) Sie sollten jede erlaubte Zahl durch eine interaktive Simulation erreichen können.

2) Gegeben sei die folgende Spezifikation:

```
byte x;
byte y;
```

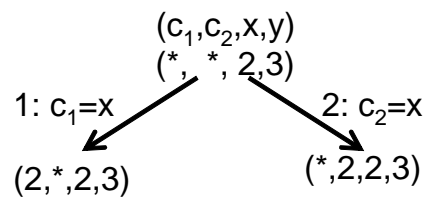
```

proctype P(){
  byte c;
  c=x;
  x=y;
  y=c
}

init{
  x=2;
  y=3;
  run P();
  run P()
}

```

Überlegen Sie sich mit einer Skizze, welche Werte x und y am Ende haben können. Der Anfang der Skizze könnte wie folgt aussehen; bedenken Sie, dass c jeweils eine lokale Variable ist.



Überprüfen Sie die möglichen Endergebnisse, indem Sie diese durch Simulation mit XSPIN erzeugen.

3) Zu entwickeln ist eine Spezifikation mit N Prozessen, wobei jeder Prozess eine Boolesche Variable `drucken` enthält, diese sind in einem Array `bool drucken[N]`; global definiert. Jeder der Prozesse soll zum Start eine eindeutige Identifikationsnummer erhalten, weiterhin werden alle Werte des Arrays `drucken` auf `false` gesetzt. Jeder Prozess P_i möchte jetzt immer wieder drucken. Wenn er druckt, setzt er seine Variable `drucken[i]` auf `true`, nach dem Drucken auf `false`. Überlegen Sie sich ein Verfahren, dass immer nur ein Prozess druckt, es also maximal ein i gibt, für das `drucken[i]==true` gilt.

4) Zu entwickeln ist eine Spezifikation, die drei ganzzahlige globale Variablen x , y und z hat, die mit dem Wert Null starten. Weiterhin gibt es drei Prozesse, wobei jeder eine dieser globalen Variablen schrittweise bis einschließlich 10 hochzählt. Die Prozesse sollen sich so synchronisieren, dass jeweils immer nur ein Prozess fortschreiten kann, genauer erst P_1 , dann P_2 , dann P_3 und dann wieder von vorne. Bei einer interaktiven Simulation sollte damit die Spezifikation durchlaufen, ohne

3 Modelchecking mit Timed Automata und Uppaal

Um die Zusammenarbeit verteilter Systeme zu ermöglichen, gibt es verschiedene Varianten, die typischerweise Kommunikationsmöglichkeiten zum Informationsaustausch und gemeinsame Variablen beinhalten. Spezifikations Sprachen unterscheiden sich häufig in der Mächtigkeit der angebotenen Möglichkeiten. Ein weiteres zentrales Unterscheidungskriterium ist der Umgang mit Zeit. Gerade bei verteilten Systemen, aber auch bei sequenziellen Programmen, kann die Dauer von bestimmten Aktionen die Funktionalität und die Qualität eines Systems wesentlich beeinflussen. Reagieren Systeme zu langsam, werden Sie von Endnutzern nicht akzeptiert; technische Komponenten hingegen werden dann nicht warten wollen und schicken Timeout-Nachrichten zum Abbruch. Mit Timed Automata lernen wir in diesem Kapitel eine Spezifikations Sprache kennen, die einfache synchrone Kommunikation, globale Variablen und Uhren zur Kontrolle von Aktionen anbietet. Timed Automata können komfortabel mit dem Werkzeug Uppaal bearbeitet werden. Weiterhin bietet Uppaal Modelchecking für Anforderungen in der Timed Computation Tree Logic an. Es handelt sich dabei um eine eingeschränkte Variante der allgemeinen Timed Computation Tree Logic. Wie immer gilt, dass man die Sprachkonstrukte zur Spezifikation auswählen muss, die zum realen System passen.

Grundsätzlich kann dieses Kapitel ohne das Vorgänger-Kapitel gelesen werden. An einigen Stellen werden einige Vergleiche mit und Verknüpfungen zu PROMELA aufgezeigt, die man beim alleinigen Studium von Timed Automata einfach überlesen kann. Zur Einführung in das Thema Modelchecking sollte das Teilkapitel 2.1 gelesen werden.

Mit PROMELA ist es möglich, verteilte Systeme zu spezifizieren und ihre Eigenschaften nachzuprüfen. PROMELA ist aber für Anfänger relativ schwer zu handhaben, da man sich an die Syntax gewöhnen muss und die Unterstützung bei syntaktischen Fehlern relativ gering ist. Um den Zugang zu Spezifikationen zu erleichtern, kann man auf den Einsatz visueller Möglichkeiten zurückgreifen. Dies ist eine Motivation für den Einsatz von Timed Automata, da hier Spezifikationen auf der Basis graphisch dargestellter endlicher Automaten entwickelt werden.

Der Einsatz von PROMELA wird schwierig, wenn man versucht, Aussagen über die benötigte Ausführungszeit von Spezifikationen zu gewinnen. Man kann zwar versuchen, spezielle Zählvariablen für Zeitaspekte einzuführen, sehr viel bringt dieser Ansatz aber nicht, da z. B. das Voranschreiten von Zeit in mehreren Prozessen aufwändig nachmodelliert werden müsste. Timed Automata bilden da eine Abhilfe, da hier spezielle Zeitvariablen, so genannte Uhren, genutzt werden können. Es wird so möglich zu spezifizieren, dass gewisse Zeit in Zuständen verbracht wird und in bestimmten Zeitintervallen Aktionen ausgeführt werden können.

In diesem Kapitel lernen Sie zunächst, wie man verteilte Systeme mit synchroner Kommunikation graphisch spezifizieren kann. Danach wird gezeigt, wie man Spezifikationen um Zeitbedingungen ergänzt und welche Besonderheiten, wie neue auf Zeitrestriktionen beruhende Deadlocks, zu beachten sind. Die Nutzung des hierbei unterstützenden Werkzeugs Uppaal wird dann beschrieben. Eine Vielzahl von Anforderungen können in der dann eingeführten Timed Computation Tree Logic spezifiziert werden. Einige Fallstudien runden das Kapitel ab.

3.1 Synchron kommunizierende Automaten

Die Basis von Timed Automata-Spezifikationen sind endliche Automaten, wie sie aus der theoretischen Informatik [VW04] bekannt sind. Es werden nicht-deterministische Automaten genutzt, so dass zu einem konkreten Zeitpunkt Übergänge zu verschiedenen Zuständen mit der gleichen Aktion möglich sein können.

Zur Spezifikation verteilter Systeme werden mehrere Automaten eingesetzt, die für Timed Automata Templates genannt werden; der Template-Begriff wird später noch präzisiert. Die Automaten können mit synchronen Kommunikationen Informationen austauschen. Dabei beschreibt $c!$ wieder, dass über einen Kanal c gesendet, und $c?$, dass über einen Kanal c empfangen werden soll. Diese Kommunikation ist dabei nur möglich, wenn Sender und Empfänger zur Kommunikation bereit sind.

Anders als in PROMELA können die Kommunikationen nicht direkt mit einer Wertübertragung zusammen spezifiziert werden. Dies ist aber durch die Nutzung globaler Variablen modellierbar.

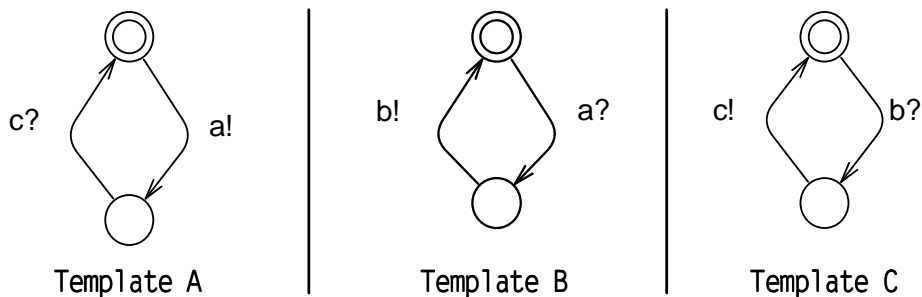


Abb. 91: Parallelkomposition von Automaten

Als erstes Beispiel wird ein kleines System mit drei Prozessen A, B, C spezifiziert, wobei zunächst A mit B über a , dann B mit C über b und schließlich C mit A über c kommuniziert. Danach beginnt die Kommunikation von vorne. Zunächst werden die Kanäle als globale Variablen spezifiziert. Dies erfolgt durch

```
//global
chan a;
chan b;
chan c;
```

Als Kommentare stehen die von C++ und Java bekannten Möglichkeiten mit `//` für einen Kommentar, der bis zum Zeilenende geht, und mit einem längeren Kommentar, der in `/*` und `*/` eingeschlossen ist, zur Verfügung.

Danach werden die Prozesse als endliche Automaten, ab jetzt Templates genannt, spezifiziert. Die Spezifikation steht in Abb. 91, dabei bezeichnet ein doppelt umrandeter Zustand bei Templates den jeweiligen Startzustand. Zur Spezifikation des Gesamtsystems muss angegeben werden, aus welchen Teilen das System zusammengesetzt wird, dies erfolgt durch

```
system A,B,C;
```

Durch die Abänderung dieser Zeile ist es z. B. möglich, spezifizierte Templates in der zu untersuchenden Spezifikation nicht zu berücksichtigen.

Neben Kommunikationskanälen können in den Spezifikationen auch lokale und globale Variablen mit den „üblichen“ Datentypen `int` und `bool` eingesetzt werden. Man kann für jede Transition dann festlegen, welche Veränderungen der Variablen stattfinden sollen.

Als Beispiel soll das Template A aus der vorherigen Spezifikation eine lokale Variable `x` erhalten, die mit dem Wert Null initialisiert, bei jeder `a`-Kommunikation hochgezählt und bei jeder `c`-Kommunikation runtergezählt wird. Es wird folgende lokale Variable genutzt.

```
// lokale Variablen
int x:=0;
```

Zuweisungen können bei Timed Automata mit `x:=0` oder `x=0` erfolgen, was semantisch die gleiche Bedeutung hat. Da die Zuweisung mit `:=` sich besser von dem Vergleich mit `==` auf Gleichheit unterscheidet, wird dieser Ansatz verfolgt.

Abb. 92 zeigt das veränderte Template für A. Man erkennt, dass es verschiedene Möglichkeiten gibt, die Veränderung lokaler Variablen zu beschreiben. Die offensichtlichste Variante ist es, die Änderung direkt an die Transition zu schreiben. Falls es mehrere Änderungen geben soll, sind diese mit einem Komma zu trennen. Die zweite Variante ist die Nutzung von Funktionen, die dann wiederholt im Template genutzt werden können. Diese Funktionen basieren auf der C-Syntax, aus der die Notationen für `if`, `while`- sowie `for`-Schleifen übernommen werden. Die Funktion `sub` sieht dann wie folgt aus.

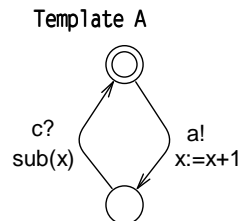


Abb. 92: Template mit lokaler Variable

```
void sub (int& v){
    v:=v-1;
}
```

Ähnlich wie bei den meisten Programmiersprachen ist bei der Übergabe von Variablen zu klären, wie diese übergeben werden. Es werden folgende Varianten unterstützt.

Call by Value (int x): Es wird eine vollständige Kopie des übergebenen Werts erstellt, Veränderungen der Variable x sind nur in der Funktion bekannt und haben keine Auswirkungen auf die beim Aufruf genutzte Variable.

Call by Reference (int& x): Es wird eine Referenz auf die übergebene Variable erstellt, x hat den gleichen Speicherplatz wie die übergebene Variable, Veränderungen der Variable x verändern auch die beim Aufruf genutzte Variable.

Im konkreten Fall wird die übergebene Variable um Eins verringert.

Variablen können auch genutzt werden, um Vorbedingungen anzugeben, die erfüllt sein müssen, damit eine Transition ausgeführt werden kann. Dies soll mit folgendem Beispiel in Abb. 93 verdeutlicht werden.

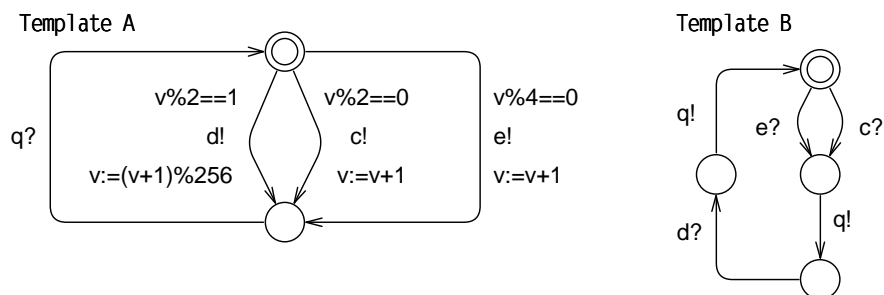


Abb. 93: Transitionen mit Bedingungen

Dabei steht % wieder für die Modulo-Rechnung, also $7\%3=1$. Die beiden Templates nutzen die folgenden Kommunikationen

```
//global
chan c,d,e,q;
```

Das System wird durch

```
system A,B;
```

zusammengebaut. Weiterhin hat A eine lokale Variable

```
int[0,255] v:=0;
```

Durch die Angabe des Intervalls [0,255] wird festgelegt, dass v nur Werte aus diesem Bereich, einschließlich der Grenzen, annehmen darf. Wird der Wertebereich verlassen, handelt es sich um einen Spezifikationsfehler.

Betrachtet man die Spezifikation als Parallelkomposition von A und B zunächst ohne Vorbedingungen, dann kann am Anfang nichtdeterministisch zwischen c und e gewählt werden, danach sind dann nur q , d und wieder q möglich. Dies entspricht dem regulären Ausdruck $((c+e).q.d.q)^*$. Durch die Vorbedingungen ist am Anfang weiterhin die Auswahl zwischen c und e offen. Danach hat v den Wert 1. Nach $q.d.q$ hat v den Wert 2. Jetzt ist $v\%2==0$ erfüllt, aber nicht $v\%4==0$, so dass jetzt nur noch c ausgeführt werden kann. Nach weiteren $q.d.q$ hat v den Wert 4 und c sowie e sind wieder möglich. Insgesamt gehört damit zur Spezifikation der reguläre Ausdruck $((c+e).q.d.q.c.q.d.q)^*$.

Zunächst irritiert es leicht, dass hier die Bezeichnung Template für Automaten genutzt wird. Der Name wird aber plausibel, wenn man die jetzt vorgestellte Möglichkeit kennt, dass man über Parameter steuern kann, dass es mehrere gleichartige Automaten gibt. Dies wird durch folgende Spezifikation veranschaulicht. Für alle Datentypen gibt es zunächst die Möglichkeit, Arrays dieser Datentypen in der von C bekannten Art zu bilden.

```
//global
const int N:=4;
chan c[N];
chan d[N];
bool gesendet[N];
```

Konstanten werden durch das Schlüsselwort `const` eingeleitet, es gibt vier c -Kanäle mit den Namen $c[0]$, $c[1]$, $c[2]$ und $c[3]$, vier d -Kanäle und vier Boolesche Variablen.

Die Spezifikation in Abb. 94 zeigt ein parametrisiertes Template `Emp`. Diesem Template wird ein Parameterwert im Parameter name zur Konstruktion eines neuen konkreten Template übergeben. Danach ist `Emp` bereit, auf einem $c[name]$ -Kanal zu empfangen und dann auf $d[name]$ zu senden. Die parametrisierten Templates müssen vor der Nutzung der Spezifikation wie folgt zur Erzeugung konkreter Templates genutzt werden, leider gibt es hier keine Schleifennotation.

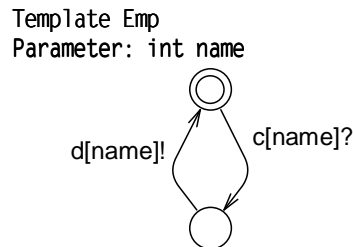


Abb. 94: Parametrisiertes Template

```

// template instantiations
E0 = Emp(0);
E1 = Emp(1);
E2 = Emp(2);
E3 = Emp(3);
  
```

Ergänzend soll jetzt ein Prozess V spezifiziert werden, der abwechselnd mit den vier Emp-Prozessen kommuniziert. Die gesamte Spezifikation hat dann die folgende Form.

```

// processes composed to system
system V, E0, E1, E2, E3;
  
```

Der Ablauf bei V soll so sein, dass mit den vier Emp-Prozessen in beliebiger Reihenfolge einmal die Kommunikationsfolge $c[i]$ und $d[i]$ durchlaufen wird. Erst wenn alle vier Prozesse kommuniziert haben, dann sollen die Prozesse erneut wieder in beliebiger Reihenfolge kommunizieren können. Formal bedeutet dies, dass für jede Sequenz w von Kommunikationen gilt, dass für alle i und j zwischen Eins und Vier die Anzahl der Vorkommen der $c[i]$ in w sich maximal um eins von den Vorkommen der $c[j]$ in w unterscheiden kann. Gleiches gilt auch für d -Kommunikationen.

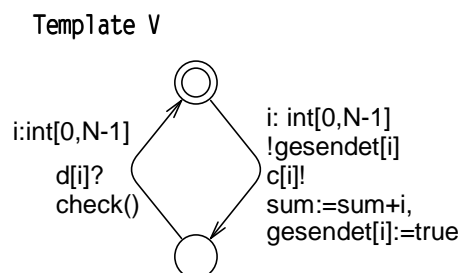


Abb. 95: Parametrisierte Transitionen

Formal benötigt V am Anfang vier Kommunikationsmöglichkeiten $c[0]$, $c[1]$, $c[2]$, $c[3]$. Dies kann, wie in Abb. 95 gezeigt, durch die Nutzung einer parametrisierten Transition vereinfacht werden. Der Ausdruck $i: \text{int}[0, N-1]$ bedeutet, dass es für

jedes i in dem genannten Intervall, hier mit den Werten 0, 1, 2 und 3, eine Transition geben soll. Auf den Index i kann bei der weiteren Beschreibung der Transition in Vorbedingung, Kommunikation und Nachbedingung direkt zugegriffen werden. Man beachte, dass das i an der mit $d[i]$ beschrifteten Kante nichts mit dem i an der mit $c[i]$ beschrifteten Kante zu tun hat; es handelt sich dabei um lokale Variablen der Transitionen.

Damit ein Prozess nicht zu häufig sendet, werden Boolesche Wächter `gesendet[i]` eingeführt, die am Anfang, insofern keine Startwerte angegeben sind, automatisch auf `false` gesetzt werden. Nur wenn `gesendet[i]` noch `false` ist, kann die zugehörige Kommunikation $c[i]$ ausgeführt werden. Danach wird als Spielerei der Wert von i auf `sum` aufaddiert. Relevant für den weiteren Ablauf ist, dass der Wert von `gesendet[i]` auf `true` gesetzt wird. Die Kommunikation auf $d[i]$ kann ohne Vorbedingung stattfinden, dabei wird die Funktion `check` ausgeführt. Die lokalen Informationen für V sehen wie folgt aus.

```
// local V
int sum:=0;

void check(){
    bool allegesendet:=true;
    for(i: int[0,N-1])
        allegesendet=allegesendet && gesendet[i];
    if(allegesendet)
        for(i: int[0,N-1])
            gesendet[i]:=false;
}
```

Die Funktion zeigt, dass es neben den aus C bekannten Schleifen auch eine auf der Intervallschreibweise basierende Schleifennotation gibt. Mit der ersten Schleife wird dabei geprüft, ob schon über alle vier Kanäle gesendet wurde, nur wenn dies der Fall ist, hat `allegesendet` nach der ersten Schleife den Wert `true`. In diesem Fall werden dann in der zweiten Schleife alle Booleschen Wächter wieder auf `false` gesetzt.

Die Funktionsschreibweise hat den Vorteil, dass man viele Informationen aus der Graphik heraushalten kann. Später wird es sich aber bei der Simulation und der Suche nach Fehlern zeigen, dass es schwierig ist, Fehler in diesen Funktionen zu finden, da sie nicht Schritt für Schritt ausgeführt werden können.

Es gibt immer die Alternative zur Nutzung von Funktionen, den Funktionsablauf im Automaten zu beschreiben, was diesen Automaten, wie in Abb. 96 ersichtlich, komplizierter macht, aber später die schrittweise Ausführung in der Simulation zur Fehlersuche erleichtert. Die neue Spezifikation von V , die die alte ersetzt, hat folgende lokale Variablen.

```
// local V
int sum:=0;
int[0,N] tmp;
```

```
bool allegesendet;
```

Die Variable `tmp` wird als Schleifenzähler genutzt, die Boolesche Variable `allegesendet` wird wie in der Funktion `check` genutzt.

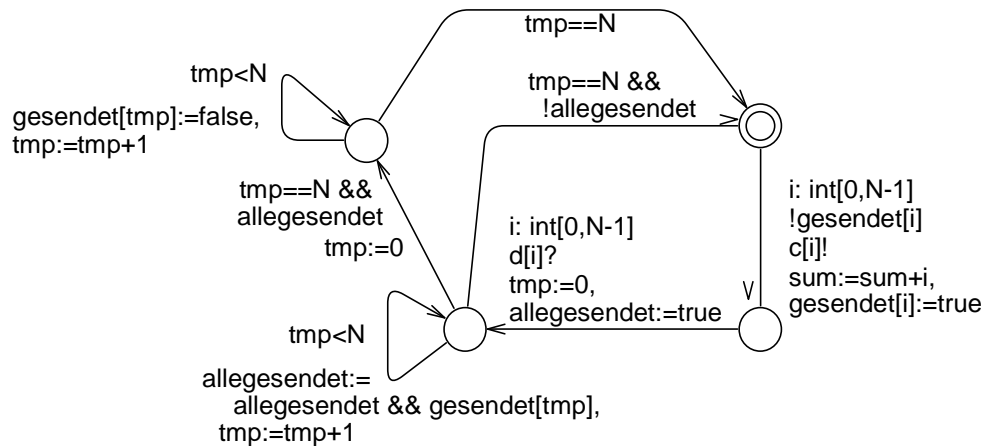


Abb. 96: Algorithmus als Template

Sollen bei einer Kommunikation Werte übertragen werden, so werden dazu globale Variablen genutzt, die vom Sender beschrieben und vom Empfänger gelesen werden. Dies soll mit einem kleinen Beispiel verdeutlicht werden, bei dem ein Sender die Werte von Eins bis Zehn zu einem Empfänger übertragen möchte.

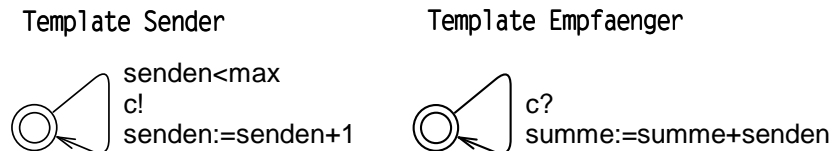


Abb. 97: Übertragung von Werten

Abb. 97 zeigt die Templates zu den Prozessen. Es werden folgende globale Variablen deklariert.

```
const int max:=10;
int[0,max] senden:=0;
chan c;
```

Der Empfänger hat noch folgende lokale Variable.

```
int summe:=0;
```

Das Gesamtsystem ist wie folgt spezifiziert.

system Sender,Empfaenger;

Als erstes wird der Wert 1 an den Empfänger übertragen. Daraus kann man auch die detaillierte Semantik einer gemeinsamen Transitionsausführung zweier Prozesse ableiten.

1. Es wird mit den aktuellen Werten geprüft, ob die Vorbedingungen erfüllt sind.
2. Es wird geprüft, ob Sende- und Empfangsoperation stattfinden können.
3. Wenn 1. und 2. positiv geprüft werden, kann die Kommunikation stattfinden und es werden zuerst die Zuweisungen des Senders und dann die Zuweisungen des Empfängers ausgeführt.

Als weitere Spezifikationsmöglichkeit bieten Timed Automata an, dass Zustände nur erreicht werden können, wenn zugehörige Zusicherungen erfüllt sind, außerdem können Zustände Namen bekommen. Zustandsnamen und Zusicherungen sind fettgedruckt. Das später vorgestellte Werkzeug Uppaal unterstützt die Lesbarkeit durch die Nutzung verschiedener Farben.

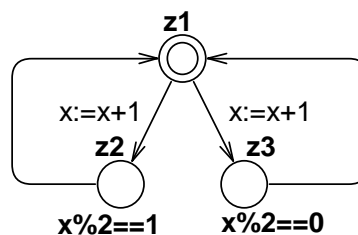


Abb. 98: Zustände mit Zusicherungen

Die Spezifikation in Abb. 98 nutzt folgende lokale Variable.

```
int x:=0;
```

Zunächst sieht es so aus, dass nichtdeterministisch immer eine der ausgehenden Kanten in z1 nutzbar ist. Da eine Transition aber nur ausgeführt werden kann, wenn die Zusicherung des erreichten Zustands erfüllt ist, werden immer abwechselnd die linke und dann die rechte ausgehende Transition von z1 gewählt.

3.2 Spezifikationen mit Zeit

Die Untersuchung zeitlichen Verhaltens spielt bei verteilten Systemen oft eine große Rolle, da Fragen nach der schnellstmöglichen Reaktion und nach maximalen Antwortzeiten beantwortet werden sollen. Weiterhin gibt es Kommunikationsprotokolle, die ohne Zeitbetrachtung funktionieren und erst durch die Berücksichti-

gung der Zeit Probleme haben. Genauso ist die andere Variante möglich, dass die Korrektheit eines Protokolls auf Zeitannahmen basiert.

Grundsätzlich verbraucht jeder Rechenschritt, ausgehend von der einfachen Zuweisung bis zur ausgeführten Kommunikation, Zeit. Wenn man allen Schritten einen Zeitaspekt zuordnen will, werden Spezifikationen sehr komplex. Aus diesem Grund nutzen Timed Automata einen anderen Zeitanatz, der aber die Modellierung der anderen Zeitanätze erlaubt.

In Timed Automata gibt es den zusätzlichen Datentypen `clock`. Es ist festgelegt, dass Zeit nur in Zuständen verbraucht wird, also die Ausführung von Transitionen keine Zeit verbraucht. Da der Begriff Zustand im eigentlichen Sinn nicht mit dem Verbrauchen von Zeit in Verbindung gebracht wird, nutzen die Autoren der Timed Automata auch nicht den Begriff Zustand, sondern *location*, der ins Deutsche als Lokation oder besser Aufenthaltsort übersetzt werden kann. Da der Zustandsbegriff genügend anschaulich ist, wird hier auf eine weitere Begriffsbildung verzichtet.

Uhren messen Zeit und dazu wird eine Zeiteinheit benötigt. Bei Timed Automata werden abstrakte Zeiteinheiten genutzt, denen man in realen Spezifikationen z. B. die Semantik von Sekunden oder Mikrosekunden zuordnen kann. Da man durch die Einführung von Zeitvariablen leicht einen unentscheidbaren Modellierungsansatz erhalten kann, gibt es folgende Restriktionen bei der Verwendung von Zeitvariablen in Timed Automata.

- Uhren können nur mit einer anderen Uhr oder ganzzahligen Ausdrücken, die keine Uhren enthalten, verglichen werden.
- In Bedingungen dürfen mehrere Teilbedingungen, die sich auf Uhren beziehen, nicht mit „oder“ verknüpft werden.
- In Zuweisungen können Uhren nur auf den Wert Null zurückgesetzt werden.

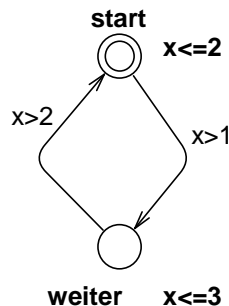


Abb. 99: Template mit Zeit

Es wird jetzt die Spezifikation aus Abb. 99 mit einer lokalen Uhr `clock x`; betrachtet. Die Zusicherung in `start` garantiert, dass dieser Zustand nach spätestens 2 Zeiteinheiten verlassen wird; die beiden Transitionen kann man sich auch mit der Bedingung `true` beschriftet vorstellen. Wichtig ist, dass Uhren nicht automatisch zurückgesetzt werden. Da es in der Spezifikation kein `x:=0` gibt, wodurch eine Uhr zurück gesetzt wird, läuft die Uhr immer weiter. Der Zustand `weiter` muss nach maximal drei Zeiteinheiten verlassen werden.

Da die Zusicherung in `start` garantiert, dass maximal zwei Zeiteinheiten vergangen sind, kann diese Transition nicht ausgeführt werden, wenn mehr als zwei Zeiteinheiten insgesamt in `weiter` verbracht wurden. Insgesamt wird also die ausgehende Transition von `start` im Zeitintervall $[0,2]$ ausgeführt. Ab drei Zeiteinheiten befindet sich die Spezifikation in einem inkonsistenten Zustand, da die Zusicherung des Zustands `weiter` nicht mehr erfüllt ist und keine ausgehende Transition genutzt werden kann. Da es keine Verpflichtung gibt, Zeit in einem Zustand zu verbringen, kann man beliebig oft beliebig schnell zwischen den beiden Zuständen wechseln. Formal ist eine Verweildauer von Null Zeiteinheiten erlaubt. Dies klingt zunächst irritierend, macht aber den Spezifikationsansatz, wie später gezeigt, flexibler.

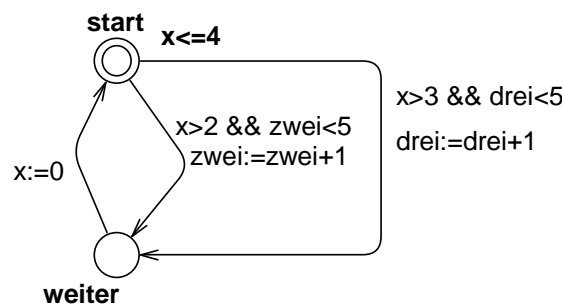


Abb. 100: Nichtdeterminismus mit Zeit

Die Spezifikation aus Abb. 100 wird um folgende lokale Variablen ergänzt.

```

clock x;
clock y;
int [0,5] zwei:=0;
int [0,5] drei:=0;
  
```

Der Zustand `start` muss nach maximal vier Zeiteinheiten verlassen werden, nach mehr als zwei Zeiteinheiten und erfüllter Nebenbedingung `zwei<5` kann die linke und nach mehr als drei Zeiteinheiten und erfüllter Nebenbedingung `drei<5` auch nichtdeterministisch die rechte Transition genutzt werden.

4 Petrinetze

Petrinetze wurden von Carl Adam Petri 1962 [Pet62] als eine der ersten formalen Beschreibungen verteilter Systeme entwickelt, die präzise Schlüsse über ihr Verhalten zuließen. Ausgehend von den klassischen endlichen Automaten, bei denen sich das System immer in genau einem Zustand befinden kann, wurde dieser Ansatz mit Petrinetzen verfeinert, bei dem es Möglichkeiten gibt, dass sich ein System in mehreren Teilzuständen befindet und gleichzeitig mehrere Aktionen ausgeführt werden können. Insgesamt ergibt sich so eine sehr einfache syntaktische Beschreibungsmöglichkeit, mit der man sehr gut verschiedene Problemquellen von verteilten Systemen vorstellen kann.

Zur Beschreibung werden nur drei Sprachelemente benötigt, dies sind zunächst Stellen, die so genannte Token, auch Marken genannt, aufnehmen können. Weiterhin gibt es Transitionen, mit denen Token von Stellen weggenommen und auf Stellen gelegt werden. Dieses so genannte Tokenspiel wird im Folgenden zunächst präzisiert.

Petrinetze sind neben ihrer Beschreibungsmöglichkeit für parallele Abläufe beliebt, da man sie unter bestimmten Randbedingungen genau analysieren kann, sie werden z. B. als semantische Grundlage von Aktivitätsdiagrammen der UML genutzt. Ähnlich wie bei Automaten kann man sich auch für ihre Sprache interessieren, sich fragen, ob und wie Token auf bestimmte Stellen gelangen können sowie welche generellen Eigenschaften man aus der Struktur eines Netzes ableiten kann. Diese mathematisch fundierten Analysemethoden werden in den dann folgenden Unterkapiteln beschrieben, und es wird gezeigt, dass sie werkzeuggestützt berechenbar sind. Einige Fallstudien runden diese Vorstellung ab.

Abschließend werden Petrinetze genutzt, um die generell sehr spannende Frage zu analysieren, wann man in einem verteilten System eine einzelne Komponente durch eine andere ersetzen kann. Die Antwort zeigt, dass eine reine Betrachtung von Schnittstellen bei Weitem nicht ausreicht, um einzelne Komponenten gegeneinander austauschen zu können.

4.1 Funktionsweise von Petrinetzen

Petrinetze bestehen aus den folgenden drei Bestandteilen:

- Stellen, meist als Kreise dargestellt, die Token enthalten können.
- Token, die nur in Stellen existieren und bei geringer Anzahl als schwarze Punkte, sonst als Zahl, dargestellt werden.
- Transitionen, meist als Kästen oder sehr dicke Striche dargestellt, die Token von Stellen entfernen beziehungsweise auf Stellen legen können.

Stellen und Transitionen sind miteinander durch gerichtete Kanten verbunden, dabei können diese Kanten nur Stellen mit Transitionen und Transitionen mit Stellen verbinden.

In Abb. 134 sind auf der linken Seite drei Petrinetze dargestellt, wobei die Stellen auf der linken Seite jeweils mit einem Token markiert sind. Durch die schraffierten Pfeile wird das so genannte Schaltverhalten von Petrinetzen angedeutet. Allgemein formuliert beschreibt diese Regel, wann eine Transition schalten kann. Dazu muss auf jeder Stelle, die durch eine in die Transition eingehende Kante verbunden ist, mindestens ein Token liegen. Schaltet dann die Transition, wird auf jede Stelle, die über eine von der Transition ausgehende Kante verbunden ist, ein Token gelegt und es wird von jeder Stelle, die über eine zur Transition hinführende Kante verbunden ist, ein Token weggenommen. Die Beispiele zeigen, dass die Anzahl der Token im Netz dabei nicht konstant bleiben muss.

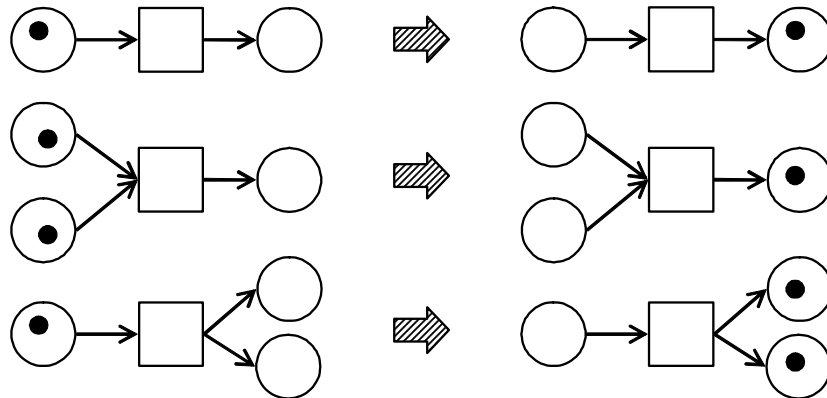


Abb. 134: Schaltende Transitionen

Das Aussehen der drei Netze nach dem Schalten ist auf der rechten Seite der Abbildung dargestellt. Man erkennt hier bereits, dass es Transitionen geben kann, mit denen getrennte Abläufe vereinigt werden, was durch das Netz in der Mitte spezifiziert wird. Weiterhin ist ersichtlich, dass man durch Transitionen verteilte Abläufe starten kann, was durch das untere Netz beschrieben wird. Die Möglichkeit, dass sich mehr als ein Token auf einer Stelle befindet, wird hier noch nicht benutzt.

Nun soll als einleitendes Beispiel ein Prozess spezifiziert werden, der für zwei getrennt laufende Prozesse den Zugriff auf eine gemeinsame Ressource, z. B. einen Drucker, regelt, der nur von einem Prozess allein genutzt werden kann.

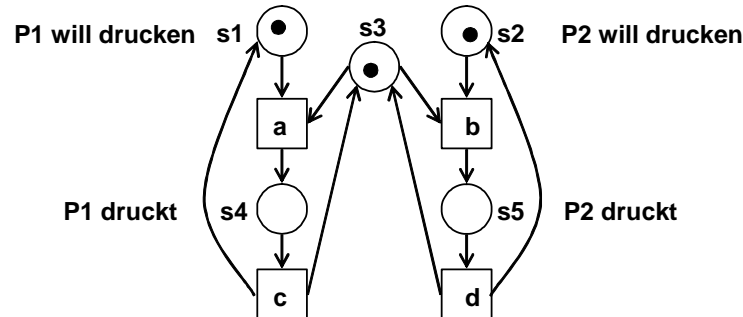


Abb. 135: Wechselseitiger Ausschluss als Petrinetz

Das zugehörige Netz ist in Abb. 135 abgebildet. Man erkennt, dass man zur besseren Argumentation den Stellen und Transitionen Namen geben kann, die beiden Prozesse P1 und P2 sind links und rechts dargestellt, wobei P1 gerne a und dann c und P2 gerne b und dann d ausführen möchte. Dabei können c und d nicht gleichzeitig stattfinden, da nur ein Prozess auf den Drucker zugreifen soll. Im Beispiel ist s3 die entscheidende Synchronisationsstelle. Nach der Schaltregel können jetzt entweder a oder b schalten, was einer Zuordnung der Ressource an einen Prozess entspricht. Wenn der Prozess die Ressource genutzt hat, wird durch c bzw. d dieses so genannte Synchronisationstoken wieder auf s3 erzeugt, so dass die Synchronisation von Neuem starten kann.

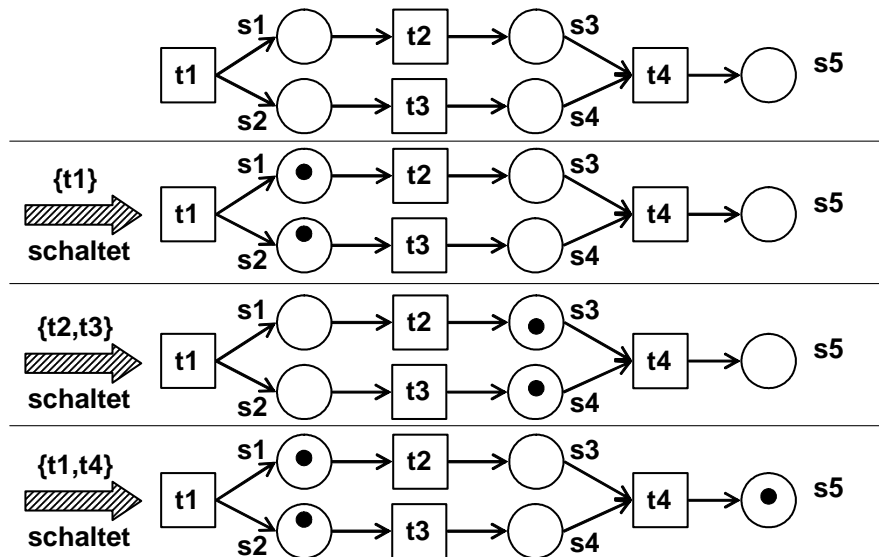


Abb. 136: Beispiel für Schaltfolge

Sachwortverzeichnis

\perp	216	für totale Korrektheit	228
ω	170	BigInt	241
\equiv	213	Bitstate-Hashing	57
\rightarrow^*	209	Boolesche Bedingung	
$[\cdot >]$	167	Ausführbarkeit	18
$_pid$	46	Breitendurchlauf	54
$\{p\}$ Prog $\{q\}$	215, 217	Call by Reference	120
abstrakte Klasse	287	Call by Value	20, 120
accept-Markierung	73	clock	126
Aktivitätsdiagramm	163	Deadlock	32, 130, 169, 190
all pathes	139	Deadlockvermeidung	191
Alternative	202	Default-Konstruktor	284
Alternativenregel	221	Design	9
always	77, 139	dinierende Philosophen	190
Ampelsystem	89	disjunkte Programme	257
Anforderungsanalyse	8	Divergenz	214
Anweisungsüberdeckung	235	in parallelen Programmen	247
Äquivalenzklassen zur Testfallermittlung		Domäne	205
.....	234	Einführung neuer Programmkonstrukte	
assert	233	209
Assertion	60	einschwingendes System	147
asynchrone Kommunikation	28	erfüllen	212
atomare Aktion	31, 249	erreichbare Konfiguration	209
Atomaritätsregel	252	Erreichbarkeit	
atomic	249	unentscheidbar	174
Attribut	283	Erreichbarkeitsgraph	169
Ausführung		Erweiterung von Klassen	286
paralleler Programme	245	eventually	77
Automat	118	Exception	235
Bedingung		Exemplarvariable	<i>Siehe</i> Attribut
Semantik	206	exists a path	139
Syntax	203	Fairness	68, 142, 168, 248
Befehlssequenz	203	Fallgraph	<i>Siehe</i> Erreichbarkeitsgraph
Begrenzungsfunktion	232	Fehlertrace	64
Beweisskizze	231	finally	139
Beweissystem	219	Folgemarkierung	167
für parallele Programme	253	formales Modell	10
für partielle Korrektheit	220	Formulierung von Korrektheit	219

Sachwortverzeichnis

Fortschritt.....	80	total korrekt	217
gemeinsame Variablen	245	leads to.....	139
Genauigkeit der Zahlendarstellung.....	240	Lebendigkeit	60, 67, 141, 169
gepufferte Kommunikation	28	Lineare Temporale Logik	77
gerichtete Kommunikation.....	29	lineares Gleichungssystem	
Geschäftsprozess.....	7, 187	homogenes	176
globally	139	Lösungsraum.....	177
Grenzwertanalyse	235	Livelock	67
Guarded Command Language.....	18	Liveness	60
Halteproblem.....	219	Lock.....	264
Hash-Funktion.....	51	logische Äquivalenz.....	212
Hoare-Kalkül	219	logische Schlussfolgerung	212
imperative Programmierung.....	201	Lokation.....	126
Implikation.....	212	LTL	77
inkrementelle Entwicklung.....	9	als Never-Claim	83
Instanzvariable	<i>Siehe</i> Attribut	Semantik.....	79
Interface.....	287	Syntax	77
Interferenzfreiheit		typische Anforderungen.....	80
für Terminierungsfunktionen	254	markiertes Petrinetz	192
von Beweisskizzen.....	252	Markierung	166
von Zusicherungen.....	252	erreichbar	169
Interleaving-Semantik	245	reproduzierbar	176
Invariante	61	überdeckt	170
Java.....	281	Matrix eines Petrinetzes	175
Zusicherung	233	Matrixmultiplikation	175
join.....	261	Message Sequence Chart	43
JSPIN.....	36	Methode.....	284
Kanal		Modelchecker.....	49
unidirektional.....	25	Modelchecking	13
Klassenmethode	284	Fehlerquellen	15
Klassenvariable	288	Laufzeit.....	157
Knotenüberdeckung	235	Obermengenbetrachtung	58
Kommunikation		Teilmengenbetrachtung	58
synchron	26, 28	Umgang mit Grenzen	56, 144
Kommunikationskanal.....	25, 118	Modell.....	1
Kommunikationsprotokoll	16	Nachbedingung	10, 215, 217
kommunizierende Prozesse.....	25	Namensraum	283
Kompositionsregel	221	NaN.....	241
Konfiguration	208	Netlab.....	180
Konstruktor.....	284	Never-Claim.....	72
Korrektheit	219	Nichtdeterminismus	18, 192
Korrektheitsformel.....	215	Non-Progress-Cycles	68
partiell korrekt	215	null	285

Oberklasse	286	wechselseitiger Ausschluss	165
Paket		Potenzmenge	169
in Java	283	Präfixabschluss	193
parallele Programme	243	Prioritätenregelung	92
Partial Order Reduction	57	Programm	
partielle Korrektheit	215	partiell korrekt	215
partielle Semantik	214	partielle Semantik	214
partieller Livelock	68	total korrekt	217
passendes Modell	2	totale Semantik	216
Petrinetz	163	Programmkonfiguration	208
ϵ -Übergang	192	Programmzähler	256
aktivierte Transition	166	Programmzustand	205
Deadlock	169	PROMELA	14
Definition	166	_pid	46
erreichbare Markierung	169	accept-Markierung	73
Erreichbarkeitsgraph	169	active	21
Fairness	168	Alternative	21
Folgemarkierung	167	Array	24
formale Sprache	193	asynchrone Empfangsmöglichkeiten	32
lebendig	169	asynchrone Kommunikation	28
markiert	192	atomic	31
Markierung	166	bewachte asynchrone Kommunikation	
Matrix	175	34
Nachbereich	166	chan	25
Parallelkomposition	192	Datentypen	18
positive S-Invariante	180	deterministisches Programm	84
pref(.)	193	dubious else	35
Ready-Menge	193	Empfangsvarianten	27
reproduzierbare Markierung	176	end-Markierung	63
Schaltverhalten	164	Funktion	87
Simulation	184	goto	21
S-Invariante	179	Kanalanalyse	29
Sprachäquivalenz	195	lokale Variable	18
Stelle	163	Makro	17
T-Invariante	176	mtype	24
Token	163	Never-Claim	72
Trace	193	Nichtdeterminismus	18
Trace-Readiness-Semantik	195	Operatoren	19
Transition	163	printf	45
transponierte Matrix	179	progress-Markierung	68
überdeckte Markierung	170	Prozess	16
Überdeckungsgraph	172	Prozessmarkierung	61
Vorbereich	166	Record	24

Sachwortverzeichnis

run	21	Software-Fehler	3
Schleife	18	sometimes.....	139
Simulation.....	38	Sortiervverfahren.....	84, 148
synchrone Kommunikation.....	26	Spezifikationssprache	14
Trace-Zusicherung.....	71	Anforderungen daran	16
XSPIN	36	SPIN	13, 36
Queries	141	Verifikationsoptionen.....	53
Ready-Menge.....	193	Verifikationsprozess	65
Rendezvous.....	26	Sprache eines Petrinetzes	193
run.....	260	Stabilität.....	80
Runnable	288	starke Fairness	69
Safety	60	strukturierte operationelle Semantik ..	208
schleichende Fehler.....	7	Substitution	213
Schleife.....	202	Suchtiefe	56
Schleifeninvariante	221	super	286
Schleifenregel		Supertrace.....	57
für partielle Korrektheit	221	synchrone Kommunikation	26, 118
für totale Korrektheit.....	228	Synchronisation	165
schwache Fairness.....	69	synchronized.....	261
SDL.....	98	Syntax	
nach PROMELA.....	103	LTL.....	77
Prozess	100	Petrinetz	166
Signal.....	100	Programmiersprache	202
Signalabarbeitung.....	101	PROMELA	16
Semantik		SDL	100
der Programmausführung.....	208	TCTL.....	138
einer Zusicherung.....	212	Timed Automata	118
LTL	79	TCTL	138
paralleler Programmausführung.....	245	deadlock	139
partielle eines Programms	214	Teilprogramm	243
totale eines Programms.....	216	Telefonsystem	151
von Bedingungen.....	206	Template.....	119
von TCTL	140	Terminierung	63, 214
semantisch äquivalent	213	Terminierungsfunktion	228
Sequenz	202	Testen mit Programmverifikation	
Sicherheit.....	60, 141	verknüpft	241
Sichtbarkeit		Testfall	
von Attributen.....	283	Äquivalenzklassen.....	234
Simulation	38, 136, 184	Grenzwertanalyse	235
S-Invariante.....	179	this	284
positiv	180	Thread.....	260, 288
Software Engineering	8	Tiefendurchlauf	54
Software-Entwicklung.....	8	Timed Automata	117

Call by Reference.....	120	Mehrdeutigkeit.....	173
Call by Value	120	Übergangsrelation	208
chan.....	118	Uhr.....	126
clock	126	UML.....	163
Committed	130	unendlicher Ablaufpfad.....	68
Datentypen.....	119	unidirektional.....	25
Deadlock.....	130	Unit-Test	235
deterministisches Programm.....	148	unlock	264
Fairness.....	142	until.....	77
Funktion	119	Uppaal.....	117, 131
globale Variablen	118	Diagnostic Trace	144
Kommentar	119	Queries.....	141
lokale Variablen.....	119	Simulation	136
Lokation.....	126	Variablen eines Programms	204
parametrisierte Transition.....	122	Vererbung	240, 286
parametrisiertes Template	121	Verifikation deterministischer	
Semantik einer Transitionsausführung		Programme.....	84, 148
.....	125	Verkaufsautomat	192
Semantik von Uhren.....	126	verteiltes System	16
synchrone Kommunikation	118	Optimierung.....	195
Template.....	119	vollständiges Beweissystem	222
Urgent.....	130	Vorbedingung	10, 215, 217
Vorbedingung.....	120	Wasserfallmodell	9
Zustandszusicherungen	125	wechselseitiger Ausschluss	81, 165
Zuweisung	119	Wertebereich.....	205
Timed Computation Tree Logic	138	XSPIN	36
Timer	147	Zeit.....	117
T-Invariante.....	176	Zusicherung.....	60, 212
Tokenspiel	184	als Zustandsmenge.....	212
totale Semantik	216	Zustand	
Trace.....	193	einer PROMELA-Spezifikation.....	49
Trace-Readiness-Äquivalenz	196	eines Programms	205
Trace-Readiness-Semantik	195	erfüllt Zusicherung.....	212
Trace-Zusicherung	71	Zustandsänderung.....	207
Transformation	15	Zustandsraum	50
Transition.....	208	Zustandsraumexplosion	244
aktiviert	166	Zustandsvektor	49
transitive Hülle	209	Zustandszusicherungen.....	125
typische Anforderungen.....	80	Zuweisungsregel.....	220
Überdeckungsgraph.....	172		

Formale Modelle der Softwareentwicklung
Model-Checking, Verifikation, Analyse und Simulation

Kleuker, S.

2009, X, 301 S. 206 Abb., Softcover

ISBN: 978-3-8348-0669-7